Using Variability Modeling to Support Security Evaluations: Virtualizing the Right Attack Scenarios

Andy Kenner
METOP GmbH
Magdeburg, Germany
Andy.Kenner@metop.de

Stephan Dassow
METOP GmbH
Magdeburg, Germany
Stephan.Dassow@metop.de

Christian Lausberger
METOP GmbH
Magdeburg, Germany
Christian.Lausberger@metop.de

Jacob Krüger
Otto-von-Guericke University
Magdeburg, Germany
jkrueger@ovgu.de

Thomas Leich
Harz University & METOP GmbH
Wernigerode & Magdeburg, Germany
tleich@hs-harz.de

ABSTRACT

A software system's security is constantly threatened by vulnerabilities that result from faults in the system's design (e.g., unintended feature interactions) and which can be exploited with attacks. While various databases summarize information on vulnerabilities and other security issues for many software systems, these databases face severe limitations. For example, the information's quality is unclear, often only semi-structured, and barely connected to other information. Consequently, it can be challenging for any security-related stakeholder to extract and understand what information is relevant, considering that most systems exist in different variants and versions. To tackle this problem, we propose to design *vulnerability feature models* that represent the vulnerabilities of a system and enable developers to virtualize corresponding attack scenarios. In this paper, we report a first case study on Mozilla Firefox for which we extracted vulnerabilities and used them to virtualize vulnerable instances in Docker. To this end, we focused on extracting information from available databases and on evaluating the usability of the results. Our findings indicate several problems with the extraction that complicate modeling, understanding, and testing of vulnerabilities. Nonetheless, the databases provide a valuable foundation for our technique, which we aim to extend with automatic synthesis and analyses of feature models, as well as virtualization for attack scenarios in future work.

CCS CONCEPTS

• Security and privacy → Penetration testing; **• Software and its engineering →** *Software product lines.*

KEYWORDS

Vulnerability, Exploit, Attack Scenarios, Software Architecture, Docker-Container, Variability Model, Feature Model

ACM Reference Format:
Andy Kenner, Stephan Dassow, Christian Lausberger, Jacob Krüger, and Thomas Leich. 2020. Using Variability Modeling to Support Security Evaluations: Virtualizing the Right Attack Scenarios. In *Proceedings of the 14th International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS '20), February 5–7, 2020, Magdeburg, Germany.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3377024.3377026

1 INTRODUCTION

Software systems are constantly threatened by their vulnerabilities and exploits that are, besides other factors, used to describe the threat level of a system [20, 29]. Various organizations, governments, and researchers maintain databases to provide an overview of vulnerabilities, exploits, and other security issues, for example:

- National Vulnerability Databases (e.g., the US NVD[1]),
- Exploit Database,[2]
- Offensive Security Exploit Database,[3]
- rapid7 Vulnerability and Exploit Database,[4] and
- rapid7 open research datasets.[5]

All of these sources can comprise valuable information on a software's security problems. More precisely, vulnerability databases allow any security-related stakeholder to search for vulnerabilities that have been reported for a software system. These problems may relate to the software itself (e.g., Firefox), a component of the software (e.g., Firefox plug-ins) or its integration into a larger infrastructure (e.g., tools that rely on Firefox to visualize data). So, the information is highly important for software users, developers, and researchers to be aware of and tackle security problems. Moreover, many additional stakeholders (e.g., administrators, security auditors) who are involved in the security management of organizations, software infrastructures, and software systems do benefit from this information.

However, already for a single system, many pieces of information may or may not be available in varying granularity, for example, Common Vulnerabilities and Exposures (CVE)[6] identifiers, descriptions of the vulnerability as well as attack scenarios including the

[1]https://nvd.nist.gov/
[2]https://www.exploit-db.com/
[3]https://www.offensive-security.com/community-projects/the-exploit-database/
[4]https://www.rapid7.com/db/
[5]https://opendata.rapid7.com/
[6]http://cve.mitre.org/cve/identifiers/index.html

concrete exploit, and system specifications. The system specifications can include, for instance, operating systems, platform information, versions, and configurations (e.g., what combination of software causes a vulnerability). For the purpose of identifying vulnerabilities for a specific system under test, we can consider all of these specifics as variability. To manage such variability, researchers in the software-product-line engineering domain have proposed variability models, and especially feature models have become established in academia and industry [1, 6, 12, 19].

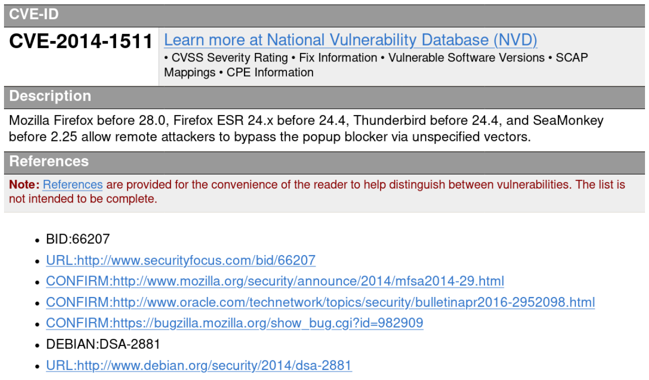
In this paper, we report our first findings of designing our technique and creating vulnerability feature models. More precisely, we conducted an exploratory case study on Mozilla Firefox during which we identified what information we can extract from existing databases to describe vulnerabilities and exploits. In addition, we used the information to design a vulnerability feature model by hand. To prove and verify the extracted information, we virtualized vulnerable systems and attacked them. We conducted this exploratory case study manually to understand the problems we may have to tackle with our technique. The results indicate that we can utilize the available information, but automatically extracting and synthesizing information into a vulnerability feature model requires adapted techniques to consider varying quality and availability of information, and its links throughout different databases. Still, the model allows us to document threats and to receive the list of vulnerabilities for a provided system configuration.

2 BACKGROUND

Vulnerabilities. A vulnerability is usually caused by errors or design flaws in a software that allow attackers to gain access to the system and manipulate it [16, 22]. For example, the execution of malicious code on a local system (e.g., via SQL injections [7]) or remote attacks through compromised links in mails may both allow an attacker to gain access to the software, its data, or the system beyond. The most import prerequisite for an attack is the existence of an exploit, of which more and more are identified every day. An exploit is a systematic way to make use of vulnerabilities in a system's architecture [16], forcing a break at the vulnerable part to gain access to resources, to penetrate the system or to harm the users in any other way. In this paper, we focus on analyzing the available information in vulnerability databases and on virtualizing attackable systems based on Docker containers [4].

CVE Identifiers. Vulnerabilities and exploits are the central elements that we aim to evaluate and to model for a software system or landscape. Both are collected in specialized databases to gather all existing information, which requires a unique identifier to map the same content throughout the databases. For such an identifier, the industrial CVE[6] standard became the default and is also established in academia—with The MITRE Corporation operating the corresponding information system. We display an excerpt of a CVE entry in Figure 1. A CVE identifier (i.e., CVE-2014-1511[7]) is assigned to each vulnerability after its discovery and an additional audit [17]. In addition to the identifier, each CVE entry also comprises at least a brief description and a list with links to additional information. Such links can include, for instance, vulnerability databases, community websites, the developers' issue- and bug-trackers, and GitHub.

[7]http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-1511


CVE-ID

CVE-2014-1511 Learn more at National Vulnerability Database (NVD)
• CVSS Severity Rating • Fix Information • Vulnerable Software Versions • SCAP Mappings • CPE Information

Description

Mozilla Firefox before 28.0, Firefox ESR 24.x before 24.4, Thunderbird before 24.4, and SeaMonkey before 2.25 allow remote attackers to bypass the popup blocker via unspecified vectors.

References

Note: References are provided for the convenience of the reader to help distinguish between vulnerabilities. The list is not intended to be complete.

- BID:66207
- URL:http://www.securityfocus.com/bid/66207
- CONFIRM:http://www.mozilla.org/security/announce/2014/mfsa2014-29.html
- CONFIRM:http://www.oracle.com/technetwork/topics/security/bulletinapr2016-2952098.html
- CONFIRM:https://bugzilla.mozilla.org/show_bug.cgi?id=982909
- DEBIAN:DSA-2881
- URL:http://www.debian.org/security/2014/dsa-2881

Figure 1: Excerpt of CVE-2014-1511.[7]

Vulnerability Databases. For each vulnerability, corresponding databases, such as the national vulnerability databases, report the weak spot of a system in detail. The provided information include the required software variant, version, and operating system—also with its variant and version. However, in most cases, this information is in pure textual form.

Exploit Databases. CVE identifiers are also used within exploit databases. Such databases report how to exploit a vulnerabilty and provide the malicious code or even detailed guides. The Exploit Database[2] and the rapid7 Vulnerability and Exploit Database[4] do belong to the most extensive of such databases.

Attack Scenario Dataset and Framework. A distinct amount of entries in exploit databases belongs to datasets, which are embedded into frameworks that allow to automatically simulate attack scenarios for a given vulnerability. One prominent example is the MetaSploit Framework (MSF) [15], which comprises more than 3,800 modules that allow to run attack scenarios on a software system. We use the MSF in combination with Docker containers to virtualize and attack a system, and rely on the aforementioned databases to verify and enrich the information we identified.

3 THE BIG PICTURE

The different databases provide a variety of information sources to describe the vulnerabilities of a software system and assess the security risks it is exposed to. We are only aware of the National Institute of Standards and Technology to provide a collection of vulnerabilities for software configurations called Common Configuration Enumeration (CCE).[8] However, this includes only a limited number of systems, seems to be out of date (last updated in 2013 as of 2019), is arguably not flexible enough for highly-configurable systems or infrastructures, and requires considerable manual effort to analyze. To overcome these problems, we intend to automatically analyze vulnerability and exploit databases to extract a *vulnerability feature model*. With a vulnerability feature model, we intend to describe the vulnerabilities of a software system and their dependencies, similar to a feature model that describes the variability in a software product line.

To create such a vulnerability feature model, we need to recover and synthesize the information that is provided in different

[8]https://nvd.nist.gov/config/cce/index

information sources (i.e., vulnerability databases and the linked references). Few pieces of information are rather explicit and can be easily mapped (e.g., CVE identifiers, references). Unfortunately, most of the actually interesting information (e.g., the actual attack scenarios, version numbers, configurations) are usually described in natural language (cf. Figure 1). Consequently, we need to adopt advanced analysis techniques to not only model and document explicit information, but also details of the vulnerabilities and to find side effects (e.g., a vulnerability only mapped to a Firefox plug-in, but not to Firefox itself).

In contrast to the scattered information provided in the vulnerability databases, a complete vulnerability feature model provides a homogeneous view on the threat level of a software, facilitates documentation, and reduces manual analysis efforts. The resulting model and its analysis enable security-related stakeholders, for example, to create and evaluate testing systems, to identify vulnerabilities of a system by providing its configuration, and to perform automated analyses of threats. Depending on a stakeholder's intent and perspective (e.g., assess the security of the infrastructure), our model is the basis for customized views and analyses that provide additional information to the stakeholder.

In summary, our overarching goal is to develop a technique that: i) automatically extracts and links information from vulnerability databases; ii) semi-automatically connects the information and synthesizes a vulnerability feature model; and iii) provides all security-related stakeholders the ability to identify the relevant vulnerabilities and attack scenarios for their purpose. This technique helps, for instance, to document and model vulnerabilities, understand and address potential security threats in a software system and infrastructure, and enable penetration testing by virtualizing the configured system to attack it. So, our technique will be an asset for researchers and practitioners alike.

4 STUDY DESIGN

The threat level of a software system is defined by the number of its identified vulnerabilities and potential attack scenarios. By analyzing vulnerabilities, exploits, and attack scenarios, we can describe connections between a system and possible attacks. To assess to what extent we can utilize vulnerability databases and synthesize the information, we conducted a case study. In the following, we report our study design to extract information from the aforementioned databases and to simulate attacks. Based on the results, we plan to extend and automate the information extraction to generate our envisioned vulnerability feature models.

To get an impression on the information provided in vulnerability databases, we used them to create Docker containers that comprise an attackable system. Currently, we focused on the MSF [15] to attack the virtualized system, as this framework comprises a database of directly usable attack scenarios. During our analysis, we found that most other databases provide additional information, but it is usually not readily reusable. So, we first aimed to evaluate the usability of our technique before integrating cost-intensive data analysis, cleansing, and synthesizing tools.

Extracting Information. We started by analyzing the attack-scenario modules in the MSF and manually extracted information on versions, variants, platforms, and dependencies that are affected by a vulnerability. This information is available through a module's description and its additional meta-information. To refine the information on versions, variants, and platforms, we analyzed the CVE identifiers as well as referenced vulnerability and exploit databases.

Creating a Vulnerability Feature Model. We used the identified and synthesized information that we extracted in the first step to manually create a vulnerability feature model. For this purpose, we relied only on the basic concepts of feature models and included all vulnerabilities we identified. The resulting model allows to configure a system and, for now, can be used to automatically create a list of all relevant vulnerabilities for that configuration.

Defining Docker Images. Based on the extracted information, we defined an attackable system. We used the platform information to select an image that is provided either by the Docker community or the developer of the operating system. Furthermore, we determined what sources provided the correct versions and variants of a system. We manually analyzed additional dependencies for each system and integrated them into the Docker image.

Executing Attack Scenarios. Every attack scenario in the MSF is a separate module with specific parameters, such as an IP address and a web-link that contains the malicious code. We needed to execute the code while the MSF documented the interaction between our attackable system and attacker. The documentations showed whether the scenarios were successful, and provided hints on potential sources for faults. To execute attacks, we virtualized the system that we defined as a Docker image and analyzed its behavior. This allowed us to check whether we investigated all needed properties and to verify the reliability of the extracted information.

5 CASE STUDY

As an initial proof for our technique, we used Mozilla Firefox as victim and explored suitable attack scenarios in the MSF. To avoid technical and legal restrictions, we decided to select Linux-based operating systems for our case study. In particular, the Docker community already provides several basic images for Linux that facilitate our setup and the installation of Firefox. Based on those requirements, we systematically identified 18 exploit modules of the MSF by following the folder structure and by searching for the phrase "Firefox" in the GitHub repository.[9] Each of these modules encapsulates the definition of a single attack scenario. We used this as a starting point for our investigations to create vulnerable Docker images based on the extracted information. In the remaining paper, we name these modules S01 to S18 (**S**cenarios). We provide a summary of the most important facts of the attack scenarios in Table 1, including the relevant software, version numbers, operating systems, and CVE identifiers that correspond to each scenario. As success (✓), we classified each vulnerability that we were able to exploit while running the attack scenario. Otherwise, we classified the attack scenario as a failure (✗). A failure may be caused in any step of the working process of a scenario, from the definition of the Docker image to the exploitation within the attack script of the MSF. Whatever the reason, the result remained that we could not successfully replicate the attack scenario.

[9]https://github.com/rapid7/metasploit-framework

Table 1: Attack scenarios and their characteristics that we extracted for Mozilla Firefox.

ID	Version	Operating system	CVE	Success
Firefox-related				
S01	35.0 - 36.0.4 (FF)	Linux (Ubuntu 12.04 - 14.10)	CVE-2015-0816 CVE-2015-0802	✓
S02	5.0 - 15.0.1 (FF)	undefined (Ubuntu 10.04 - 12.04)	CVE-2012-3993	✓
S03	31.0 - 34.0.5 (FF)	undefined (Ubuntu 12.04 - 14.04)	CVE-2014-8636 CVE-2015-0802	✓
S04	1.5.0 (FF)	Linux (Ubuntu 4.1 - 5.10)	CVE-2006-0295	✗
S05	17.0 - 17.0.1 (FF)	undefined (Ubuntu 10.04 - 12.10)	CVE-2013-0757	✓
S06	15.0 - 22.0 (FF)	undefined (Ubuntu 12.04)	CVE-2013-1710	✓
S07	22.0 - 27.0.1 (FF)	undefined (Ubuntu 12.04 - 13.10)	CVE-2014-1510 CVE-2014-1511	✓
S08	≤ 42.0† (FF)	undefined (Ubuntu)	No identifier	✓
Adobe Flash-related				
S09	33.0 (FF), 11.2.202.468 (F)	Linux Mint 17.1 "Rebecca"	CVE-2015-5119	✓
S10.1	33.0 (FF), 11.2.202.466 (F)	Linux Mint 17.1 "Rebecca"	CVE-2015-3043 CVE-2015-3113	✓
S10.2	35.01 (FF), 11.2.202.466 (F)	Ubuntu 14.04	CVE-2015-3043 CVE-2015-3113	✓
S11.1	33.0 (FF), 11.2.202.424 (F)	Linux Mint 17.1 "Rebecca"	CVE-2015-0336	✗
S11.2	33.0 (FF), 11.2.202.442 (F)	Ubuntu 14.04	CVE-2015-0336	✗
S12	33.0 (FF), 11.2.202.350 (F)	Linux Mint 17.1 "Rebecca"	CVE-2015-3043 CVE-2015-0515	✓
S13	33.0 (FF), 11.2.202.460 (F)	Linux Mint 17.1 "Rebecca"	CVE-2015-3043 CVE-2015-3105	✓
S14	33.0 (FF), 11.2.202.457 (F)	Linux Mint 17.1 "Rebecca"	CVE-2015-3043 CVE-2015-3090	✓
S15	33.0 (FF), 11.2.202.424 (F)	Linux Mint 17.1 "Rebecca"	CVE-2015-3043 CVE-2015-0311	✓
Java-related				
S16	undefined (FF), 7 (J), ≤ 6U27 (J)	Linux (x86) (Ubuntu 10.04 - 11.10)	CVE-2011-3544	✗
S17	undefined (FF), 7 (J)	Linux (x86) (not Ubuntu/Mint)✜	CVE-2012-4681	✗
S18	1.5.0 - 1.5.0.4 (FF), undefined (J)	Linux (x86) (Ubuntu 5.10 - 6.06)	CVE-2006-3677	✓/✗★

(FF) - Mozilla Firefox; (F) - Adobe Flash; (J) - Java
† Manually identified range 3.6.16 - 42.0
✜ Not vulnerable on Ubuntu or Mint, examplary tested by reusing S16.
★ - Firefox crashes successfully, Docker restriction closes container, too

In the first section of Table 1, we describe the scenarios S01 to S08, which comprise exploits that address Firefox itself as the victim and try to use the vulnerability of a specific version or a range of versions as a gateway for attacks. Within the second and third section, we display scenarios with exploits that occur due to interactions of Firefox with its plug-ins: The scenarios S09 to S15 describe exploits that are based on Abobe Flash, which is installed as a Firefox extension for Flash-related content of websites. S16 to S18 cover vulnerabilities that are caused by the run-time environment for Java code and the specific Firefox extension through which Java code is executed and embedded in web-pages as an applet.

Overall, we manually defined 48 Docker images. We needed this number of Docker images to test the specified version as well as the following and previous ones to assess the borders of the version ranges. In the following, we report the details of our case study.

5.1 Extracting Information

For browser-related attacks, each MSF entry defines a structured part for data and meta-data that stores, among others, the following:

- **Description:** A textual description of the vulnerability in a software system, including the affected versions, the underlying operating system, and the activities that are needed to execute the attack; and thus exploit the vulnerability.
- **References:** Links to additional information sources, such as one or multiple CVE identifiers and other sources that document the vulnerability.
- **Targets:** Defines the platform in detail as a listing, usually the operating systems (e.g., Windows, Linux, MacOS) including specific derivatives of Linux (e.g., Ubuntu, Mint), on which the attack scenario is executed, comprising version numbers, variants (e.g., x86, x64), and sometimes patch versions (e.g., Service Pack 1 for Windows).
- **Browser Requirements:** Defines an MSF specific function, which is used to check that all requirements of the script are fulfilled. These requirements include the browser (i.e., Firefox), supported versions (e.g., minimum, maximum), and versions of additional dependencies (e.g., Adobe Flash, Java).

In addition to the MSF, we also investigated the referenced CVE identifiers and vulnerability databases to verify and synthesize the information we extracted.

We show all this extracted information for our case study in Table 1. In the column Version, we display the version numbers for Firefox (FF), Adobe Flash (F), and Java (J). The column Operating System comprises the version numbers and variants for the Linux operating system or its distributions. To allow traceability and replicability, we also present the corresponding CVE identifiers.

Insights. For our manual extraction from different information sources, we experienced the following:

- **Analyzing MSF Entries:** An MSF entry encapsulates the attack scenario, the attack script as a main part, and a predefined set of meta-information. The fields description, targets, and browser requirements comprise various types of information. We found plain text for the description, a free-selection list of supported OS-targets, and a ruby-based code fragment in the browser requirements. This code fragment is executed at the beginning of a script and restricts the execution to defined versions of the operating system, Firefox, and optionally for the additional Adobe Flash or Java.
  In ideal cases (e.g., S10.1, S10.2), the provided information is detailed and didn't require any further investigation. However, in other cases, the information about Firefox and/or

the operating system was missing. As we show in the column Version in Table 1, we were able to extract the correct versions for most scenarios, except for S16–18. Here, the information for versions of Firefox (S16, S17) or Java (S18) was missing. For the column Operating System in particular, we can see a high variance in the details documented in an MSF entry, for example, we found no specification for six scenarios (S02, S03, S05–S08). Five other entries contained only a general description (e.g., Linux) as requirement for the operating system (S01, S04, S16–S18).

- **Missing, Imprecise Information:** We needed to perform additional, manual analyses of referenced sources for supplementary information. In particular, we followed the CVE identifiers for each scenario and analyzed the main entries contained in the US NVD.[1] We used the referenced links (cf. Figure 1), which guide to other websites or archives focusing on the same topic, to fill most of the information gaps.
  During our investigation, we identified a tendency towards Ubuntu and Mint as the underlying operating system, with other candidates like SUSE Linux, Red Hat Linux or Gentoo appearing, too. Since this study is about general feasibility, we have restricted the OS to Ubuntu and Mint. This helped us in the next steps of our case study, as this restriction facilitated the definition of Docker images.
  For six scenarios, the information of the operating system was not described, another five entries provided only general values. In Table 1, we show the manually gathered information as additional data about the Ubuntu versions in brackets in the column Operating System. S17 represents a special case: We aimed to identify more precise information than the basic entry Linux (x86), but we were not able to find a reference to Ubuntu or Mint, only to other Linux-based operating systems. Due to a strong similarity to S16, we tested whether we could use the same setup as for S17.
- **Additional Investigations:** After we were able to collect almost all information in a sufficiently concrete form, additional investigations were still necessary. For the scenarios S08, S16, and S17, we had to test Firefox versions manually. Considering S08, we identified a range from 3.6.16 up to 42.0 by sampling the versions using the Docker definition of other scenarios within a range lower than Firefox version 42.0. In the cases of S16 and S17, we considered versions 5.0 until 35.0.1 the same way as for S08. As mentioned before, we reused the setup from S16 in S17 to check whether the vulnerability was existing.

As we describe in the next sections, we used this information to design a vulnerability feature model and to derive Docker images.

5.2 Vulnerability Feature Model

We used the information we extracted manually from the MSF and vulnerability databases to model and document all attack scenarios into a vulnerability feature model. In Figure 2, we show an initial version of our model (excluding any cross-tree constraints) that describes the threats we summarize in Table 1. As the attack scenarios exploit vulnerabilities caused by specific versions of the operating system, Firefox, and its plug-ins, these are our primary entities in

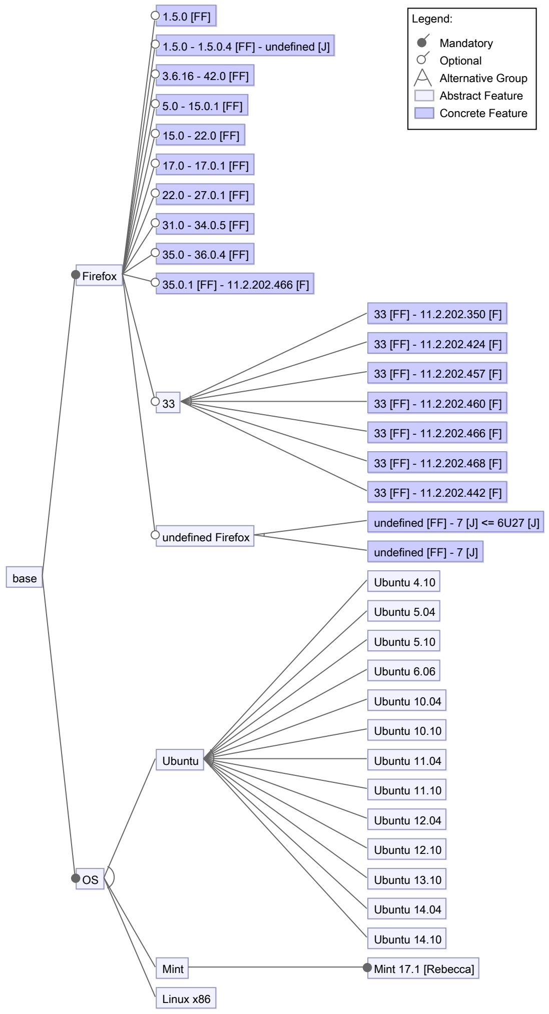


Figure 2: Vulnerability feature model for our case study.

the model at this point. We remark that this model is an initial concept to illustrate the usability of the information we extracted in our case study: Other information may be incorporated (e.g., modeling the attacks) and will influence a model's form, and potentially the modeling technique we use (e.g., extended feature models, decision models) [3, 6, 27]. This strongly depends on the information an organization wants to include at the end, and is influenced by the range of applications included. Also, we did not follow guidelines or principles to derive, model, and organize features, so far [12, 19].

We use the leaf features and 24 cross-tree constraints to model the relations between operating systems and Firefox. The constraints restrict the variation space in order to allow only those configurations that we found to be vulnerable (cf. Table 1). Most constraints are implications to model the intersections of operating systems and Firefox in the attack scenarios. As we omit the cross-tree constraints in Figure 2 to improve readability, we provide a fully functioning model in an open-source repository (cf. Section 8).

Based on the vulnerability feature model, we may be able to configure systems in such a style that they are vulnerable according to the attack scenarios in Table 1. For instance, almost all child features of feature "33" can only be selected if the feature "Mint 17.1 Rebecca" is also selected. This covers the scenarios S09–S15, which all require this particular configuration. The concrete features—which comprise the related CVE identifiers—below the feature "Firefox" are selected automatically, due to their constraints. We can then create a list of vulnerabilities that are relevant for our configuration.

Insights. As we can see in Figure 2, we used mainly the features "Firefox" and "OS" to structure the feature model. Currently, we use the feature "Firefox" to actually represent all attack scenarios, but do not add features for Adobe Flash or Java. Moreover, the features in this branch of the model represent the column Version in Table 1, which may not be ideal. Nonetheless, we can configure the feature model to obtain a list of attack scenarios, and thus of all CVE identifiers, that are relevant for that configuration. So, we argue that our vulnerability feature model is a helpful means to document and analyze vulnerabilities for a configurable system, but its final representation will most likely vary from the one we selected for this case study. For example, the branch below the feature "OS" groups the variants and versions of the operating systems defined in the attack scenarios. While, for now, we separated between individual versions, attributed feature models may be a helpful means to improve the structure.

5.3 Defining Docker Images

To define the Docker images, we first selected and set up the operating system's version—or an equally old one if the original one was not available. We relied on images that are provided by the Docker community. For most scenarios, we were able to use either Ubuntu 12.04 or Ubuntu 14.04 as basic image for the Docker definition. The scenarios S04, S08, S16, and S18 require an Ubuntu version that has been released earlier or has not been defined in detail. Since there were no basic images for earlier versions ($\leq$ 11.10) in Docker, we tried to implement them with Ubuntu 12.04, which seems to be suitable for S08. An image for Mint 17.1 was also not available in the Docker community. Using guided modifications, we rebuilt a fully valid image based on an existing Ubuntu image.

In order to install the correct Firefox version for the scenario, we used external sources that allowed us to derive the corresponding installation packages. Precisely, Firefox versions are available as Debian packages in special archiving version databases.[10] While defining the Docker images, we used `dkpg` (Debian package manager) to install the specific Firefox version manually. To install the Debian package successfully, we had to determine every additional library, which is a necessary requirement for the browser. Despite the usual way to install Firefox in Ubuntu with the help of `apt`, which installs a software including all of its dependencies, at this point we had to choose the manual installation. That way, we managed to install older versions of Firefox as well as Adobe Flash and Java, independent from the repositories behind `apt`. Those repositories are restricted to newer versions to prevent the installation of software releases that are known to be vulnerable.

We had to manually determine all dependencies and, with additional effort in debugging, resolve installation errors. A repetitive, step-wise procedure of adding a required library to the installation and retrying the manual installation of Firefox led us to a complete list of first-order dependencies. To be sure to meet every other entity of the dependency tree, we used an `apt`-based installation of the list we determined before. We were able to reuse this part of the Docker definition with minimal adjustments for all scenarios.

By using the Docker framework, we aimed to reach a high level of automation. This should result in Docker images that are fully defined and may be used directly as an attack victim without any further activities. To reach this goal, for S09–S15, we also needed installation packages for Adobe Flash, more precisely the Adobe Flash plug-in. Unfortunately, we are not aware of any centralized databases providing different versions of the plug-in, but we were able to retrieve them from other sources as shared object files (`.so`). In order to install the required plug-in in an older version, we had to investigate and perform the following steps:

- Identify how to integrate plug-ins in the user folder.
- Create the required folder structure within the user profile.
- Manually install the plug-in as `.so` file.

After performing these steps, we successfully installed the Adobe Flash plug-in and could automatically load it.

For the Java plug-in, we had to install the Java Runtime Environment (JRE) first. To this end, we manually installed the required package by using a `deb` package in the `dkpg` manager. The Java plug-in is a part of the JRE installation and has to be embedded into the Firefox folder structure, identical to the aforementioned Adobe Flash plug-in. Based on these steps, we created a fully specified Docker image, including all components needed to execute the attack scenarios we identified.

Insights. For defining the Docker images for our attack scenarios, we experienced the following:

- **Installation Packages:** We had to retrieve the installation packages for Firefox, Adobe Flash, and Java manually, which partly required a lot of effort. The sources containing these packages differ in many ways and are neither standardized nor centralized.
- **Standard Libraries:** Software packages have dependencies to other packages, which are standard libraries in most cases. We had to identify dependencies to standard libraries and resolve them appropriately. For our case study, we chose the manual way of repeatedly creating and testing the Docker images with a step-wise procedure.
- **Additional steps for Adobe Flash and Java:** Software interactions (e.g., plug-ins) demand additional activities that vary depending on the application. The scenarios S09–S18 need additional plug-ins that are integrated into Firefox and, in the case of Java, the installation of a complete runtime environment. For the integration into Firefox, we had to perform additional steps to install the corresponding plug-ins in a proper way.

These problems highlight that it is not only challenging to extract the information needed to define a virtualization for testing, but also to instantiate that virtualization. Despite these problems, we were able to create Docker images for each scenario, albeit in a slightly modified form, to attack them in the next step.

[10] https://sourceforge.net/projects/ubuntuzilla/files/mozilla/apt/pool/main/f/firefox-mozilla-build/

5.4 Executing Attack Scenarios

We parameterized the MSF scripts according to the scenario and attacked each virtualized system, instantiated as a Docker container, to exploit the specific vulnerability. All scenarios of our case study describe remote-based exploits that enable an attacker to manipulate the attacked system after access has been gained. The parameters themselves included setting the attack target, defining the payload, and providing the manipulated data on the framework's internal server. For our case study, we used Linux-specific remote access via reverse shells as payload, through which we had a connection to the attacked Docker container after the exploit was successful. After gaining full access to a system, an attacker can potentially execute any action. The MSF integrates simple mechanisms, for example, reading the browser history or stored passwords. We did not perform any additional actions, as we were focusing on the usability of our technique.

As we show in Table 1 (cf. column Success), we could not execute all attack scenarios successfully. 13 of the 18 attacks we performed did exploit the defined vulnerability. We identified the following reasons causing failures:

- S04 / S16 / S17: We were not able to provide the required Ubuntu version (S04/S16) or limited our case study regarding other Linux variants (S17).
- S11.1 / S11.2: The Adobe Flash plug-in was rejected by the attack script, due to an incorrect version.
- S18: This scenario illustrated an additional problem with the virtualization and the definition of the Docker image. The attack was successful, as the scenario did crash Firefox, and thus executed code. However, the crash also resulted in the Docker container closing and only a manual repetition of the scenario revealed its actual success—but the expected code execution was missing. Restructuring the definition of the Docker image and choosing a different way to create the container instance may resolve this problem, but using a newer Ubuntu version has apparently no influence.

Besides these cases, we did not experience particular problems in executing the attack scenarios.

Insights. Some reasons for failing attack scenarios (i.e., we could not exploit a vulnerability) seem to be the aforementioned problems with imprecise or incomplete information, technical limitations, missing packages and libraries, as well as run-time errors in the attack scenarios. Additional investigations seem necessary to evaluate what information was missing and from where we could recover it in order to reach a more complete and reliable information base. Still, we were able to show that the available information can be used to successfully virtualize attack scenarios for evaluating a system's security. Due to the identified problems and variability, we argue that our vulnerability feature model is a step in the right direction to facilitate security evaluations.

6 PROSPECTS

In this paper, we showed our first findings of designing our technique and creating vulnerability feature models. As we described in Section 3, many information sources that summarize vulnerabilities of software systems exist. Our vulnerability feature model is intended to unify all existing perspectives arising from these information sources. Due to the diversity and the strategic directions of the sources and security-related stakeholders, these perspectives vary heavily. Based on our vulnerability feature models, we see many possible prospects in research and practice for the concept and ideas we presented in this paper. In the following we summarize some of them.

Determining the threat level. The mapping of a vulnerability feature model to a software system or landscape, for example, of an organization, allows an assessment of the threat level that the system is currently exposed to. This allows to develop strategies in order to deal with the resulting risk. A software system or landscape can be secured or at least protected from known vulnerabilities by concrete measures, such as installing patches or switching to alternative software and operating systems. In addition, our vulnerability feature model could help to detect further dependencies and vulnerabilities between different software systems, which do not emerge from a pure separate consideration of a single system.

Dealing with disclosures of system and landscape information. In order to use a vulnerability feature model to estimate the threat level, an organization needs to partly disclose its system and landscape information. We expect real-world systems and landscapes to provide further input to improve the model. However, disclosure is an important aspect that will definitely be of particular relevance for future use. For example, highly critical infrastructures or the application of principles like *defense in depth* may contradict such disclosures without certain security assurances [13, 28].

Analyzing risks and viability. For every vulnerability, the Common Vulnerability Scoring System (CVSS) defines two metrics to rate them [9, 18]. This leads to a very detailed description of a vulnerability's nature, containing a vector that defines an attack's details and its effect on the confidentiality, integrity, and availability of the attacked system. In addition, the CVSS metrics rate the severity of the weak spot based on a ranking from zero to ten, with ten being the highest rank and the scale reaching from none to critical. This information can also be included into the modeling process to assign additional properties to the features, for example, using extended feature models [3]. Risk estimations as well as viability analyses based on the CVSS can be addressed by using such extensions. Optimization problems arise in this area as we need to consider non-functional properties. Existing research on extended feature models and non-functional properties seems a good opportunity to advance in this direction [2, 8].

Optimizing through alternative configurations. An existing configuration of a software system or landscape that is derived from a vulnerability feature model may be optimized from different points of view. In addition to simply switching to a newer version of the software system, other changes and optimizations can be highly valuable, for example, if updates are not possible or the risk is not sufficiently reduced. Alternatives may include changing the operating system (e.g., Linux instead of Windows), replacing or removing parts of the software stack (e.g., HTML5 instead of Adobe Flash), and switching to alternative software systems in the same application spectrum (e.g., Google Chrome instead of Mozilla Firefox). Extensions to the basic feature-modeling concepts and research on optimization techniques of configurations form the fundamentals that we could reuse [21, 26].

Tailoring penetration testing. Configuring a software system or landscape based on a vulnerability feature model allows to create penetration tests that are customized for this use case. Software systems and landscapes that have been hardened according to the threat level can be subsequently investigated for the success of the security hardening. Automatically generated attack scenarios can be used to determine the real effects and to include them in the risk assessment. So far, only known attacks are considered, unknown vulnerabilities and attacks remain unnoticed. At this point, many techniques can help to create more advanced penetration tests. For example, considering the history and evolution of vulnerabilities and the corresponding vulnerability feature model, investigating dependencies or side effects that have only been determined through the model, and even artificial mutations. With such techniques, the existence of previously unknown, related vulnerabilities may be detected early on in other software systems or landscapes.

Tailoring vulnerable victim systems. Similar to the tailoring of penetration tests, we can support the creation of intentionally vulnerable systems. This may be in the form of a simple list or guideline for the manual implementation of a victim system, for instance, based on the Docker framework we used in our case study. A fully automated technique is quite conceivable and also presents a multitude of challenges in many areas, including those we highlighted in this paper concerning data collection and synthesis as well as appropriate means and techniques for variant generation.

Managing model evolution. Entries of vulnerability databases arise with a high frequency and are subject to constant changes and extensions. This defines a life cycle that affects the vulnerability feature model and causes changes. In addition, systems and processes based on it are influenced. To be aware of the effects of edits in the vulnerability feature model, a suitable tracing and management technique for changes has to be established.

7 RELATED WORK

We are aware of related works, but none of them aims to model the variations in a system's design to evaluate its security.

Vulnerable Virtual Machines. In the research areas of security and penetration testing, several virtual machines (VMs) have been developed to analyze older software and vulnerabilities. The most prominent ones may be the MSF 2/3 VM, on which we relied, or the VMs of the `vulnhub` platform that provide numerous prepared virtualized systems [15, 23]. To create variations within such vulnerable VMs, Schreuders and Ardern [23] define a generator. This shall prevent that usual VMs have always the same attack surface.

Cyber-Security Information Extraction. Currently, we started with a manual extraction of information that we plan to automate in the future to assess the threat level of a system in its whole variability. Arnav et al. [10] describe an architecture that aims to extract and connect several datasets on cyber-security. A prototypical assessment framework that focuses on cloud computing is defined by Kamongi et al. [11]. They limit their work only on the extraction and classification of cloud-specific systems and their vulnerability information. In contrast to these works, we plan to model vulnerabilities of any system and provide capabilities for automated analysis as well as virtualization for testing.

Reverse Engineering. Numerous techniques have been proposed to reverse engineer information from software systems [5]. Some techniques, especially on reverse engineering and synthesizing feature models [24, 25] and natural-language processing of documents comprising variability information [14], are closely related and relevant for our technique. We intend to build on such techniques to facilitate and improve the technique we proposed in this paper. However, none of the works we are aware of does tackle the analysis and synthesis of models from various information sources to evaluate the security of a software system. So, we can reuse existing techniques, but need to adopt them to automate our technique and assess how well they solve the problems we identified.

8 CONCLUSION

Security vulnerabilities in software systems are extensively stored in several sources, providing detailed information on the vulnerabilities' characteristics, how to exploit them, and attack scenarios. Our case study showed that the information is suitable to virtualize vulnerable systems in Docker containers. With these containers, we have been able to evaluate and analyze the Firefox architecture under attack. However, our extraction and analysis also showed that further automation and the inclusion of other information sources are necessary. In particular, we argue that it can be highly beneficial for the evaluation of system architectures and infrastructures to model the variations of the existing information in a vulnerability feature model. This would allow every security-related stakeholder, such as administrators, to assess their actual setup, including variability and versions.

For our future work, we will focus on the process of extracting and processing the information in order to build vulnerability feature models. In the long run, we plan to define and automatically extract vulnerability feature models to describe existing threats of systems, allowing their analysis and assessment. Moreover, we aim to automatically perform risk and viability assessments, and to recommend architectural restructurings to address vulnerabilities, for example, by exchanging servers or operating systems. We also intend to consider patch behavior, risk evolution, and the impact of architectural variability. Moreover, we will consider various optimization potentials as well as the automation for generating penetration tests as well as vulnerable victim systems. As part of our overarching goal of managing vulnerabilities based on our technique, we need to effectively and comprehensibly trace changes and their consequences.

Replication. To enable others to replicate this study, we provided the full list of attack scenarios we considered and their details in Table 1. Moreover, we created an open-access repository that comprises our complete vulnerability feature model, example Docker containers, and an attack scenario.[11]

ACKNOWLEDGMENTS

This research is partly supported by the German Federal Ministry of Education and Research (16KIS0526) and the German Research Foundation (LE 3382/2-3, SA 465/49-3).

[11]https://bitbucket.org/akenner/vamos-2020

REFERENCES

[1] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2016. *Feature-Oriented Software Product Lines*. Springer.

[2] Mohsen Asadi, Samaneh Soltani, Dragan Gasevic, Marek Hatala, and Ebrahim Bagheri. 2014. Toward Automated Feature Model Configuration with Optimizing Non-Functional Requirements. *Information and Software Technology* 56, 9 (2014), 1144–1165.

[3] David Benavides, Pablo Trinidad, and Antonio Ruiz-Cortés. 2005. Automated Reasoning on Feature Models. In *International Conference on Advanced Information Systems Engineering (CAiSE)*. Springer, 491–503.

[4] David Bernstein. 2014. Containers and Cloud: From Lxc to Docker to Kubernetes. *IEEE Cloud Computing* 1, 2 (2014), 81–84.

[5] Gerardo Canfora, Massimiliano Di Penta, and Luigi Cerulo. 2011. Achievements and Challenges in Software Reverse Engineering. *Communications of the ACM* 54, 4 (2011), 142–151.

[6] Krzysztof Czarnecki, Paul Grünbacher, Rick Rabiser, Klaus Schmid, and Andrzej Wąsowski. 2012. Cool Features and Tough Decisions: A Comparison of Variability Modeling Approaches. In *International Workshop on Variability Modeling of Software-Intensive Systems (VaMoS)*. ACM, 173–182.

[7] William G Halfond, Jeremy Viegas, and Alessandro Orso. 2006. A Classification of SQL-Injection Attacks and Countermeasures. In *International Symposium on Secure Software Engineering (SSSE)*. IEEE, 13–15.

[8] Fatima Z Hammani. 2014. Survey of Non-Functional Requirements Modeling and Verification of Software Product Lines. In *International Conference on Research Challenges in Information Science (RCIS)*. IEEE, 1–6.

[9] Siv H Houmb, Virginia N L Franqueira, and Erlend A Engum. 2010. Quantifying Security Risk Level from CVSS Estimates of Frequency and Impact. *Journal of Systems and Software* 83, 9 (2010), 1622–1634.

[10] Arnav Joshi, Ravendar Lal, Tim Finin, and Anupam Joshi. 2013. Extracting Cybersecurity Related Linked Data from Text. In *International Conference on Semantic Computing (ICSC)*. IEEE, 252–259.

[11] Patrick Kamongi, Srujan Kotikela, Krishna Kavi, Mahadevan Gomathisankaran, and Anoop Singhal. 2013. VULCAN: Vulnerability Assessment Framework for Cloud Computing. In *International Conference on Software Security and Reliability (SERE)*. IEEE, 218–226.

[12] Kyo C Kang, Sholom G Cohen, James A Hess, William E Novak, and A Spencer Peterson. 1990. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Technical Report. Carnegie-Mellon University.

[13] David Kuipers and Mark Fabro. 2006. *Control Systems Cyber Security: Defense in Depth Strategies*. Technical Report. Idaho National Laboratory.

[14] Yang Li, Sandro Schulze, and Gunter Saake. 2017. Reverse Engineering Variability from Natural Language Documents: A Systematic Literature Review. In *International Systems and Software Product Line Conference*. ACM, 133–142.

[15] David Maynor. 2011. *Metasploit Toolkit for Penetration Testing, Exploit Development, and Vulnerability Research*. Elsevier.

[16] Gary McGraw. 2006. *Software Security: Building Security in*. Addison-Wesley.

[17] Peter Mell, Karen Scarfone, and Sasha Romanosky. 2006. Common Vulnerability Scoring System. *IEEE Security and Privacy Magazine* 4, 6 (2006), 85–89.

[18] Peter Mell, Karen Scarfone, and Sasha Romanosky. 2007. *A Complete Guide to the Common Vulnerability Scoring System Version 2.0*. Technical Report.

[19] Damir Nešić, Jacob Krüger, Stefan Stănciulescu, and Thorsten Berger. 2019. Principles of Feature Modeling. In *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 62–73.

[20] Hiroyuki Okamura, Masataka Tokuzane, and Tadashi Dohi. 2013. Quantitative Security Evaluation for Software System from Vulnerability Database. *Journal of Software Engineering and Applications* 6, 4 (2013), 15–23.

[21] Rafael Olaechea, Derek Rayside, Jianmei Guo, and Krzysztof Czarnecki. 2014. Comparison of Exact and Approximate Multi-Objective Optimization for Software Product Lines. In *International Software Product Line Conference (SPLC)*. ACM, 92–101.

[22] Charles P Pfleeger and Shari L Pfleeger. 2002. *Security in Computing*. Prentice Hall.

[23] Z Cliffe Schreuders and Lewis Ardern. 2015. Generating Randomised Virtualised Scenarios for Ethical Hacking and Computer Security Education: SecGen Implementation and Deployment. In *UK Workshop on Cybersecurity Training & Education*.

[24] Steven She, Rafael Lotufo, Thorsten Berger, Andrzej Wąsowski, and Krzysztof Czarnecki. 2011. Reverse Engineering Feature Models. In *International Conference on Software Engineering (ICSE)*. ACM, 461–470.

[25] Steven She, Uwe Ryssel, Nele Andersen, Andrzej Wąsowski, and Krzysztof Czarnecki. 2014. Efficient Synthesis of Feature Models. *Information and Software Technology* 56, 9 (2014), 1122–1143.

[26] Norbert Siegmund. 2019. Challenges and Insights from Optimizing Configurable Software Systems. In *International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 2:1–2:2.

[27] Marco Sinnema and Sybren Deelstra. 2007. Classifying Variability Modeling Techniques. *Information and Software Technology* 49, 7 (2007), 717–739.

[28] Martin R Stytz. 2004. Considering Defense in Depth for Software Applications. *IEEE Security & Privacy* 2, 1 (2004), 72–75.

[29] Su Zhang, Doina Caragea, and Xinming Ou. 2011. An Empirical Study on Using the National Vulnerability Database to Predict Software Vulnerabilities. In *International Conference on Database and Expert Systems Applications (DEXA)*. Springer, 217–231.