Using Malware Analysis to Improve Security Requirements on Future Systems

Nancy R. Mead
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania, U.S.A.

Jose Andre Morales
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania, U.S.A.

Abstract—**In this position paper, we propose to enhance current software development lifecycle models by including use cases, based on previous cyberattacks and their associated malware, and to propose an open research question: Are specific types of systems prone to specific classes of malware exploits? If this is the case, developers can create future systems that are more secure, from inception, by including use cases that address previous attacks.**

***Index Terms*—SDLC; malware; cyberattacks; software security**

I. Introduction

Hundreds of vulnerabilities are publicly disclosed each month [1]. Exploitable vulnerabilities typically emerge from one of two types of core flaws: code flaws and design flaws. In the past, both types of flaws have facilitated several cyberattacks, a subset of which manifested globally. We define a *code flaw* as a vulnerability in the code base that requires a highly technical, crafted exploit to compromise a system; examples are buffer overflows and command injections. *Design flaws* are weaknesses in the system that may not require a high level of technical skill to craft exploits to compromise the system. Examples include failure to validate certificates, non-authenticated access, automatic granting of root privileges to non-root accounts, lack of encryption, and weak single-factor authentication. A *malware exploit* is an attack on a system that takes advantage of a particular vulnerability.

Use cases [2] describe a scenario conducted by a legitimate user of the system. Use cases have corresponding requirements, including security requirements. A *misuse case* [3] describes use by an attacker and highlights a security risk to be mitigated. Misuse cases describe a sequence of actions that can be performed by any person or entity in order to harm the system. Exploitation scenarios are often documented more formally as misuse cases. In terms of documentation, misuse cases can be documented in diagrams alongside use cases and/or in a text format, similar to that of a use case.

Several approaches for incorporating security into the software development lifecycle (SDLC) have been documented. Most of these enhancements have focused on defining enforceable security policies in the requirements gathering phase and defining secure coding practices in the design phase. Although these practices are helpful, cyberattacks based on core flaws have persisted.

Major corporations such as Microsoft, Adobe, Oracle, and Google have made their security lifecycle practices public [4-7]. Collaborative efforts such as the Software Assurance Forum for Excellence in Code (SAFECode) [8] have also documented recommended practices. These practices have become de facto standards for incorporating security into the SDLC. These security approaches are limited by their reliance on security policies, such as access control, read/write permissions, and memory protection and on standard secure code writing practices, such as bounded memory allocations and buffer overflow avoidance. These processes are helpful in developing secure software products, but—given the number of successful exploits that occur—they fall short. For example, techniques such as design reviews, risk analysis, and threat modeling typically do not incorporate lessons learned from the vast landscape of known successful cyberattacks and their associated malware.

In this position paper, we propose that current SDLC models can be further enhanced by including misuse cases derived from malware analysis. Our focus is on the vulnerabilities resulting from design flaws. We also propose an open research question: Are specific types of systems prone to specific classes of malware exploits? If this is the case, developers can create future systems that are more secure, from inception, by including use cases that address previous attacks.

The extensive and well-documented history of known cyberattacks can be used to enhance current SDLC models. More specifically, a known malware sample can be analyzed to determine if it exploits a vulnerability. The vulnerability can be studied to determine whether it results from a code flaw or a design flaw. For design flaws, we can attempt to determine the overlooked requirements that resulted in the vulnerability. We do this by documenting the misuse case corresponding to the exploit scenario and creating the corresponding use case. Such use cases represent overlooked security requirements that should be applied to future development to avoid similar design flaws leading to exploitable vulnerabilities. This process of applying malware analysis to ultimately create new use cases and their corresponding security requirements can help enhance the security of future systems.

II. Code and Design Flaw Vulnerabilities

As previously discussed, there are two types of flaws that lead to exploitable vulnerabilities: code flaws and design flaws. A code flaw is a weakness in the code base that requires specifically crafted code-based exploits to compromise a system; examples are buffer overflows and command injections. More specifically, code flaws result from source code being written without implementing secure coding techniques. Design flaws result in weaknesses that do not necessarily require code-based exploits to compromise the system. More specifically, a design flaw can result from overlooked security requirements. Examples are failure to validate certificates, non-authenticated access, root privileges granted to non-root accounts, lack of encryption, and weak single-factor authentication. Some design flaws can be exploited with minimal technical skill, leading to more probable system compromise. The process leading to a vulnerability exploit or remediation via software update is shown in Figure 1.

In this position paper, we focus on vulnerabilities resulting from a design flaw. In these cases, the overlooked requirements can be converted to a use case applicable to future SDLC cycles.

A large corpus of well-documented and studied cyberattacks is available to the public via multiple sources. We will describe here some publicly disclosed cases of exploited vulnerabilities that facilitated cyberattacks and arose from a design flaw. For each case, we will describe the vulnerability and the exploit used by malware. We will also present the overlooked requirement(s) that led to the design flaw that created the vulnerability. By learning from these cases and analyzing associated malware, we can create use cases that can be included in SDLC models.

1. In October 2013, a vulnerability was discovered that granted unauthenticated access to a backdoor of the administrative panel of several D-Link routers [9, 10]. The router runs as a web server, and a username and password is required for access. The router's firmware was reverse engineered, and the web server's authentication logic code revealed that a string comparison with "xmlset_roodkcableoj28840ybtide" granted access to the administration panel. A user could be granted access by simply changing his/her web browser's user-agent string to "xmlset_roodkcableoj28840ybtide". Interestingly, the string in reverse partially reads "editby04882joelbackdoor". It was later determined that this string was used to automatically authenticate configuration utilities stored within the router. The utilities needed to automatically reconfigure various settings and required a username and password (which could be changed by a user) to access the administration panel. The hardcoded string comparison was implemented to ensure these utilities accessed and reconfigured the router via the web server whenever needed, without requiring a username and password. The internally stored configuration utilities should not have been required to access the administration panels via the router's web server, which is typically used to grant access to external users. A non-web-server-based communication channel between internally stored proprietary configuration utilities and the router's firmware could have avoided the specific exploit described here, although further analysis could have led to a more general solution.

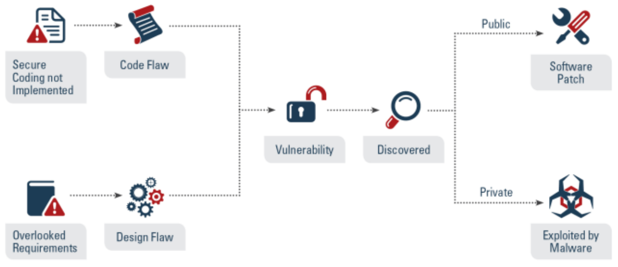

Fig. 1. Creation of a Vulnerability

2. In 2014, Xing et al. [11] discovered critical vulnerabilities in the Android operating system (OS) that allowed an unprivileged malicious application to acquire privileges and attributes without user awareness. The vulnerabilities were discovered in the Android Package Manager and were automatically exploited when the operating system was upgraded to a newer version. A malicious application already installed in a lower version of Android OS would claim specific privileges and attributes that were available only in a higher version of the Android OS. When the OS was upgraded to the higher version, the claimed privileges and attributes were automatically granted to the application without user awareness. The overlooked requirement in this case was to specify that during an upgrade of the Android OS, previously installed applications should not be granted privileges and attributes introduced in the higher OS without user authorization.
3. In March 2013, analysts discovered malware authors were creating legitimate companies for the sole purpose of acquiring verifiable digital certificates [12]. These certificates are used by malware to be authenticated and allowed to execute on a system since they each possess a valid digital signature. When an executable file starts running on an operating system such as Windows, a check for a valid digital signature is performed. If the signature is invalid, the user receives a warning that advises him/her not to allow the program to execute on the system. By possessing a valid digital signature, malware can execute on a system without generating any warnings to the user. The reliance on digital signature to allow execution of binaries on a system is no longer sufficient to avoid malware infection. The overlooked requirement in this case was to carry out multiple

security checks along with verifying the digital signature, such as (1) scanning the file for known malware, (2) querying as to whether this file has ever been executed in this system before, and (3) checking on whether the digital signature has been previously seen in other legitimate files executed on this system, before granting execution privilege to a file.

In each of the cases described above, the vulnerabilities could have been avoided had they been identified during requirements elicitation. Using risk analysis and/or good software engineering techniques, teams can identify all circumstances of use and craft appropriate responses for each instance. The cases above can be generalized and applied as needed. The following abstracts can lead to requirements statements:

- *Case 1:* Identify all possible communication channels. Designate valid communication channels, and do not permit other communication channels to gain privileges.
- *Case 2:* Do not automatically transfer privileges during an upgrade. Request validation from the user that the application or additional user should be granted privileges.
- *Case 3:* Require multiple methods of validation on executable files. In addition, as a default, consider asking for user confirmation prior to running an executable file.

In addition to the specific cases of exploited vulnerabilities discussed above, there have been several other cyberattacks on software systems, using one or more exploited vulnerabilities resulting from either a code flaw or a design flaw. These vulnerabilities are defined in a hierarchical structure using the Common Weakness Enumeration (CWE) [12-13]. The CWE provides a common language to discuss, identify, and handle causes of software security vulnerabilities. These vulnerabilities can be found in source code, system design, or system architecture. An individual CWE represents a single vulnerability type. A subset of CWEs can be attributed to design flaws resulting from overlooked requirements; some of the design flaw CWEs pertinent to the cases above are listed in Table I. An explanation follows.

TABLE I. SAMPLING OF DESIGN FLAW CWES

CWE Identifier	**Description**
CWE-306	Missing Authentication for Critical Function
CWE-654	Reliance on Single Factor in Security Decision
CWE-295	Improper Certificate Validation
CWE-326	Inadequate Encryption Strength
CWE-357	Insufficient UI Warning of Dangerous Operations

CWE-306: The software failed to perform authentication to allow access to a function that requires either a provable user or a significant amount of resources. The overlooked requirement was to enforce adequate authentication, ensuring only appropriate users are granted access.

CWE-654: The security check used to grant a user access to a restricted resource of functionality relied on a singular condition or object. If the singularity is weak—that is, if it can be identified, falsified, or modified—an attacker can gain unwarranted access. The overlooked requirement was to realize the sensitivity of the resource or function and the need to provide appropriate multi-factor authentication.

CWE-295: The software failed to adequately validate a certificate that can be leveraged by an attacker to undermine the validation and gain access to a system. An overlooked requirement was the certificate validation stage—the software trusted other aspects of the object requesting access and skipped the certificate validation stage.

CWE-326: The software used either weak or inadequate encryption for the required level of protection. The overlooked requirement was to realize the high level of security needed to access a specific function or resource; overlooking this requirement led to selecting an inadequate encryption scheme.

CWE-357: The user interface (UI) was poorly designed and did not emphasize warning messages telling the user to avoid a known dangerous or sensitive operation. The overlooked requirement was the need to clearly warn the user before granting access for such an operation. The lessons learned from previous cyberattacks and the underlying CWEs can be used to better understand overlooked requirements and resulting security implications. Analyzed and publicly disclosed cyberattacks provide details about how attackers implemented an exploit on a specific vulnerability. CWEs provide a better understanding of security vulnerabilities underlying the cyberattack. Combining information from these two sources facilitates the creation and inclusion of use cases that capture the overlooked requirements that lead to design flaws.

III. MALWARE-ANALYSIS-DRIVEN USE CASES

Malware exploits vulnerabilities to compromise a system. Vulnerabilities are normally identified by analyzing a software system or a malware sample. When a vulnerability is identified in a software system, it is documented and remedied via a software update. Vendors inform the public of vulnerabilities that are considered critical and that impact a large user base; the OpenSSL Heartbleed vulnerability is an example of such a vulnerability [14]. A vulnerability is usually identified via malware analysis after the malware has entered the wild and compromised systems. Sometimes the discovered exploited vulnerability in the analyzed malware is a zero-day vulnerability. Zero-day [15] vulnerabilities are one of the biggest threats to cybersecurity today because they are discovered in private. They are typically kept private and exploited by malware for long periods of time. Zero-days afford malware authors time to craft exploits. Zero-days are not guaranteed to be detected by conventional security measures, making their threat much more serious.

One approach to avoid creating vulnerabilities is by implementing secure lifecycle models. These models can be enhanced with the inclusion of use cases that are derived from previously discovered vulnerabilities that resulted from design flaws. Analyzing malware that exploits a vulnerability provides details of the vulnerability itself and, more importantly, provides details of the exploit implementation. The exploit details can offer additional insight into the vulnerability and the underlying design flaw.

The examples shown in the prior sections indicate that standard secure lifecycle practices may not be adequate to identify all avenues for potential attacks. Potential attacks must be addressed at requirements time (and at every subsequent phase of the lifecycle). The selected case studies illustrate that malware analysis can reveal needed security requirements that may not be identified in the normal course of development, even when secure lifecycle practices are used. At present, malware discovery is often used to develop patches or address coding errors, but not necessarily to inform future security requirements specification. We believe that failure to exercise this feedback loop is a serious flaw in security requirements engineering, which tends to start with a blank slate rather than using lessons learned from prior successful attacks. We recommend that secure lifecycle practices be modified to benefit from malware analysis.

Examining techniques that are focused on the early security development lifecycle activities can reveal how malware analysis might be applied.

Security requirements engineering techniques, such as Security Quality Requirements Engineering (SQUARE) [16] and the Secure and Usable Requirements Engineering approach [17], are silent on the use of malware analysis to inform security requirements engineering. The Core Artifacts work [18] does an excellent job of articulating a security requirements engineering framework, but it is also silent on the use of malware analysis as an input. On the other hand, earlier work on extensions to i* to account for vulnerabilities is related to this idea and will be examined carefully [19]. The CAPEC effort [20] is noteworthy in a general way. CAPEC and other languages used for expressing attack patterns may provide lessons learned so that requirements engineers specifying misuse cases can learn from history.

Maturity models such as the Software Assurance Maturity Model (SAMM) [21] hint at the use of malware analysis in earlier lifecycle phases when they discuss root-cause analysis of vulnerabilities and feedback into proactive planning. In its discussion of attack models, Building Security In Maturity Model (BSIMM) [22] also refers to the concept of using actual attack information in future development and mentions maintaining an attack database that is relevant to the organization. Since the data from participating organizations is abstracted into a set of principles, and since BSIMM is process agnostic, it is difficult to assess exactly what data is captured and how the organizations are using it.

We recommend a process for creating malware-analysis-driven use cases that incorporates malware analysis into a feedback loop for security requirements engineering on future projects, and not just into patch development for current systems. Such a process can be implemented in the following steps and is illustrated in Figure 2.

1. A malicious code sample is analyzed both statically and dynamically.
2. Analysis reveals the malware is exploiting a vulnerability that results from either a code flaw or a design flaw.

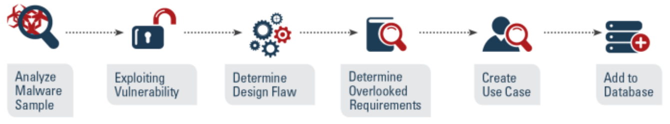

Fig. 2. Malware-Analysis-Driven Use-Case Creation

3. In the case of a design flaw, the exploitation scenario corresponds to a misuse case that should be described. The misuse is analyzed to determine the overlooked use case.
4. The overlooked use case corresponds to an overlooked security requirement.
5. The use case and corresponding requirements statement is added to a requirements database.
6. The requirements database is used in future software development projects.

Steps 1 and 2 include standard approaches to analyzing a malicious code sample. The specific analysis techniques used in Steps 1 and 2 are beyond the scope of this paper. In Step 2, the analysis is used to determine whether the exploited vulnerability is the result of a code flaw or a design flaw. Typically this can be determined by detailed analysis of the exploit code. Of course, this presumes that the malware is detected, which in itself is a challenge. Step 2 illustrates the advantage of malware analysis by leveraging the exploit code to determine the flaw type. Standard vulnerability discovery and analysis without malware analysis excludes exploit code and may make flaw type identification less straightforward. Step 3 details how the exploit was carried out in the form of a misuse case, which provides the needed information to determine the overlooked use case that led to the design flaw. In Step 4, the overlooked use case is the basis for deciding what may have been the overlooked requirement(s) at the time the software system was created that led to the design flaw. These are the requirements that should have been included in the original SDLC of the software system, which would have prevented creation of the design flaw that led to the exploited vulnerability. Steps 5 and 6 record the overlooked use case and corresponding requirement(s) for use in future SDLC cycles. This process is meant to enhance future SDLC cycles in a simplified manner by providing known overlooked requirements that led to exploited vulnerabilities. By including these requirements in future SDLC cycles, the resulting software systems can be more secure by helping avoid the creation of exploitable vulnerabilities.

IV. Conclusions and Future Work

At present, malware analysis is primarily used to identify exploits and spurs development of patches to mitigate the exploits for existing systems. Some forward-thinking organizations may use this information to help specify requirements for future software systems. We recommend incorporating this feedback loop into the secure software development process as a standard practice. We have described the process steps to support such a feedback loop.

Exploit kits [23] are often used by malware in a plug-and-play fashion to infect a system. An exploit kit is a piece of software that contains working exploits for several vulnerabilities. It is designed to run primarily on a server (exploit server) to which victim machines are redirected after clicking a malicious link, either in a webpage or email. The victim machine is scanned for vulnerabilities. If a vulnerability is identified, the exploit kit will automatically execute any applicable exploit to compromise the machine and infect it with malware. In general, malware either has exploits built into its binary or relies on exploit kits to initially compromise a machine. The implications of exploit kits is another area of exploration that may provide code samples for use in the process.

An open question for consideration: Do specific types of malware exist that are likely to occur in specific kinds of critical systems, such as control systems? Analysis of historical and current malware incidents may help to identify exploits that target specific types of applications. Knowing these exploit types in advance could help requirements engineers to identify standard misuse cases and the needed countermeasures for their specific application types. These misuse cases would again lead to corresponding use cases and security requirements. It may be wishful thinking to expect all developers of future systems to give priority to security requirements and apply the methods that we are postulating. However, developers of mission-critical systems, financial systems, and other essential systems, such as critical infrastructure systems, recognize the importance of security and should be willing to invest in it throughout the development process.

Acknowledgment

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. This material has been approved for public release and unlimited distribution. DM-0001247

References

[1] NIST. (2014). National Vulnerability Database. [Online]. Available: https://nvd.nist.gov/

[2] I. Jacobson, Object-Oriented Software Engineering: A Use Case Driven Approach. Boston, MA: Addison-Wesley, 1992.

[3] Alexander, I. "Misuse Cases: Use Cases with Hostile Intent." *IEEE Software 20*, pp 58-66, January-February 2003.

[4] S. Lipner and M. Howard. (2005, Mar.). The Trustworthy Computing Security Development Lifecycle. [Online]. Available: http://msdn.microsoft.com/en-us/library/ms995349.aspx

[5] Oracle. (2014). Software Security Assurance / Secure Development / Secure Coding Standards. [Online]. Available: http://www.oracle.com/us/support/assurance/development/secure-coding-standards/index.html

[6] Adobe Systems, Inc. (2014). Security / Proactive Efforts. [Online]. Available: http://www.adobe.com/security/proactive-efforts.html

[7] Google. (2012). Google's Approach to IT Security: A Google White Paper. [Online]. Available: https://cloud.google.com/files/Google-CommonSecurity-WhitePaper-v1.4.pdf

[8] S. Simpson (ed). (2008, Oct.). "Fundamental practices for secure software development: a guide to the most effective secure development practices in use today." SAFECode. [Online]. Available: http://www.safecode.org/publications/SAFECode_Dev_Practices1108.pdf

[9] ShyWriter. (2013, Oct. 14). "Security alert: back door found in D-Link routers." Malwarebytes.org forum. [Online]. Available: https://forums.malwarebytes.org/index.php?showtopic=134875

[10] Craig. (2013, Oct. 12). "Reverse engineering a D-Link backdoor." /dev/ttys0.com blog. [Online]. Available: http://www.devttys0.com/2013/10/reverse-engineering-a-d-link-backdoor/

[11] L. Xing, X. Pan, R. Wang, K. Yuan, and X. Wang. (2014, May). "Upgrading your Android, elevating my malware: privilege escalation through mobile OS updating." Presented at 2014 IEEE Symposium on Security and Privacy (to be published). [Online]. Available: http://www.informatics.indiana.edu/xw7/papers/privilegescalationthroughandroidupdating.pdf

[12] T. Kitten. (2013, March 11). "Digital certificates hide malware." BankInfoSecurity.com. [Online]. Available: http://www.bankinfosecurity.com/digital-certificates-hide-malware-a-5592/op-1

[13] MITRE. (2014). Common Weakness Enumeration: A community-developed dictionary of software weakness types. [Online] Available: http://cwe.mitre.org/

[14] Wikipedia contributors. (2014, April). Heartbleed. [Online]. Available: http://en.wikipedia.org/wiki/Heartbleed

[15] Wikipedia contributors. (2014, April). Zero-day Attack. [Online]. Available: http://en.wikipedia.org/wiki/Zero-day_attack

[16] N. Mead, E. Hough, and T. Stehney II. (2005, Nov.) "Security Quality Requirements Engineering." Software Engineering Institute, Carnegie Mellon University. Pittsburgh, PA. [Online]. Available: http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=7657

[17] J. Romero-Mariona, "Secure and usable requirements engineering," in Proc. of 24th IEEE/ACM International Conference on Automated Software Engineering, 2009, pp.703-706.

[18] C.B. Haley, R. Laney, J.D. Moffett, and B. Nuseibeh, "Security requirements engineering: a framework for representation and analysis," in IEEE Transactions on Software Engineering, vol. 34, no.1, pp.133-153, January 2008.

[19] G. Elahi, E. Yu, and N. Zannone, "A vulnerability-centric requirements engineering framework: analyzing security attacks, countermeasures, and requirements based on vulnerabilities" in *Requirements Engineering Journal*, vol. 15, issue 1, pp 41-62, March 2010.

[20] Common Attack Pattern Enumeration and Classification (2014). https://capec.mitre.org/

[21] Open Web Application Security Project. (2009, Mar.). Software Assurance Maturity Model: A guide to building security into software development, Version - 1.0. [Online]. Available: http://www.opensamm.org/downloads/SAMM-1.0.pdf

[22] G. McGraw. (October 2013). Building Security In Maturity Model V, Release 5.1.2. [Online]. Available: http://bsimm.com/

[23] Malwarebytes Unpacked. Tools of the Trade Exploit Kits. (2013) http://blog.malwarebytes.org/intelligence/2013/02/tools-of-the-trade-exploit-kit