Software Analysis for Security

Spiros Mancoridis
Department of Computer Science
College of Engineering
Drexel University
Philadelphia, PA 19147, USA
E-mail: spiros@drexel.edu

Abstract

This is a survey of the processes, practices, and technologies that can help software maintenance engineers improve the security of software systems. A particular emphasis is placed on validating security architectures, verifying that the implementation of an architecture's constituent applications adhere to secure coding practices, and protecting software systems against malicious software. In addition to surveying the state-of-the-art, research challenges pertaining to software security are posed to the software maintenance research community.

1. Introduction

The goal of computer security is to protect computer assets (*e.g.,* servers, applications, web pages, data) from corruption, unauthorized access, denial of authorized access, malicious software (*i.e.,* malware), and other forms of misuse. Computer security is typically strengthened by means such as: physically limiting the access of computers to trusted users, hardware mechanisms (*e.g.,* biometrics), operating system mechanisms that impose rules on untrusted programs (*e.g.,* role-based access control), anti-virus software to detect malware, and secure coding techniques (*e.g.,* array bounds checking) to make code less vulnerable to security attacks.

The field of software security pays particular attention to the development of processes and tools to help avoid and mitigate software security faults. In general, a software security fault is a fault in the specification, implementation, or configuration of a software system whose execution can violate an explicit or implicit security policy [30].

A security architecture is used as a guide to enforce security constraints. It specifies where security mechanisms (*e.g.,* encryption, firewalls) need to be positioned in the software architecture to assure the system's security. The quality of a security architecture, however, also depends on the security of the applications that constitute the system. In low security operating environments, such as most commercial operating systems, it is assumed, albeit optimistically, that these applications are trustworthy. Therefore, it is important to articulate the practices to support the secure coding of applications and the tools for detecting malware. Secure coding involves programming practices and run-time mechanisms that can be used to limit the vulnerability of applications to dangerous malware that exploit, for example, buffer overflows, format string vulnerabilities, and integer vulnerabilities to attack computer systems.

One approach to software security is to verify that (a) the security mechanisms used are trustworthy; (b) the security architecture has been validated against the security policy; and (c) the software applications that constitute the system are trustworthy (*e.g.,* they have been developed using secure coding practices, or they are not malware). Note that trusting a mechanism or verifying that an application is trustworthy does not imply that the mechanism or the application is secure. An application may be deemed trustworthy if, for example, its code has been analyzed by a suite of code security analysis or anti-virus tools with no errors or warnings reported. However, such 'trustworthy' code is still vulnerable to security attacks caused by potential faults that were not exposed by the tool.

There are various security mechanisms available to the security software architect such as: firewalls, cryptographic techniques, authentication, rule based access control, file backups, anti-virus software, biometrics, intrusion detection systems, and honey pots [38]. A survey of these mechanisms, and a characterization of how trustworthy they are, has been documented elsewhere [2].

The primary goal of this paper is to survey processes and technologies to support security architectures, the secure coding of applications, and the protection of software systems against malware. In addition to surveying the state-of-the-art, a secondary goal of this paper is to formulate research challenges in the area of software security, especially as they relate to software maintenance.

Historically, software maintenance engineers have focused on the functionality of software rather than on its security. Most software engineering textbooks barely mention security issues, or discuss them in passing along with with other non-functional requirement 'ities' such as usability, reliability, and maintainability. As a result, security faults often escape the attention of software maintainers until the software is exploited by malware.

Software maintenance researchers, likewise, have not paid close attention to security-related research, as is evidenced by the dearth of security-related papers that have been published in the software maintenance literature. This paper aims to draw the attention of software maintenance researchers and practitioners to some notable aspects of software security where they can help advance the state-of-the-art and practice, respectively.

The structure of the rest of this paper is as follows: Section 2 and Section 3 provide overviews of security architecture and secure coding, respectively, Section 4 provides an overview of the various types of malware and the tools that can be used to limit their damage, Section 5 presents research challenges related to security analysis, and the paper ends with a synopsis in Section 6.

2. Security Architecture

Unlike a software system's functional requirements, which describe what the software should do, security requirements describe non-functional constraints on what the system should not do. A security architecture illustrates how security requirements are enforced in a software system. Specifically, a security architecture describes how security mechanisms are positioned among the design artifacts of a software system to control attributes such as confidentiality, integrity, accountability, and assurance.[1]

A security architecture features a set of architectural design diagrams that show the subsystems, the communication links between the subsystems, and the position of the security mechanisms. The subsystems may be, for example, web servers, application servers, database management systems, directories, web applications, and legacy applications. The edges that connect the subsystem nodes indicate how the subsystems communicate using, for example, local or remote function calls and protocols such as TLS, SSL, HTTPS, and LDAP. The security mechanisms are often specified as annotations on the subsystems and communication links to indicate, for example, authentication points, authorization points, application administration, encryption methods, audit, logging, monitoring, intrusion detection, registration, as well as backup and recovery.

There are many security vulnerabilities that arise from poorly designed security architectures, for example, unauthorized access to data and applications, as well as confidential and restricted data flowing as unencrypted text over network connections.

Figure 1 shows an example security architecture. The rectangles represent a variety of clients and servers, and the edges represent communications between them. The directory (center of Figure 1) maintains information about each user of the system such as: user name, password, and group membership. The application server (right side of Figure 1) contains the resources provided by the system such as web pages and application functions. The web server (left side of Figure 1) acts as a proxy for accessing applications. The policy server (bottom of Figure 1) maps the resources of the application to the roles that are authorized to access each resource. For example, an application may only be accessible to users who belong to the group of managers. The policies are specified by a policy administrator via a client application. The policy server specifies, but does not enforce, the policies. Policy enforcement is the responsibility of the authorization server, which has a copy of the policy data and allows or denies access requests at run time. This particular design reflects a performance optimization decision, as the architecture could have alternatively featured a single authorization server that was shared by all other servers and accessed via a remote, rather than a local, function call.

The end user employs a web browser (top of Figure 1) to access an application via one of two options. The first option involves the end user's browser contacting the web server via the HTTPS protocol. The web server authenticates the user by matching the user name and password against the directory server. The points in the architecture that are involved in the authentication process are indicated by a small circle labeled with an 'A'. If the authentication is successful, the web server calls the authorization server to ensure that the user is authorized to access the application. The points in the architecture that are involved in any authorization processes are indicated by a small circle labeled with a 'B'.

The second option to access an application involves the end user's browser contacting the application server directly. Similar to the first option, the user is first authenticated and subsequently authorized by the user registry and the server's local authorization server, respectively.

A security architecture is validated using a process called threat modeling. Threat modeling is typically a manual (*i.e.*, not automated) inspection process, similar to code and requirements inspection. The process involves a security review team and, possibly, members of the software testing team. The goal of this process is to assess the vul-

[1] IT Security Architecture http://www.securityarchitecture.org

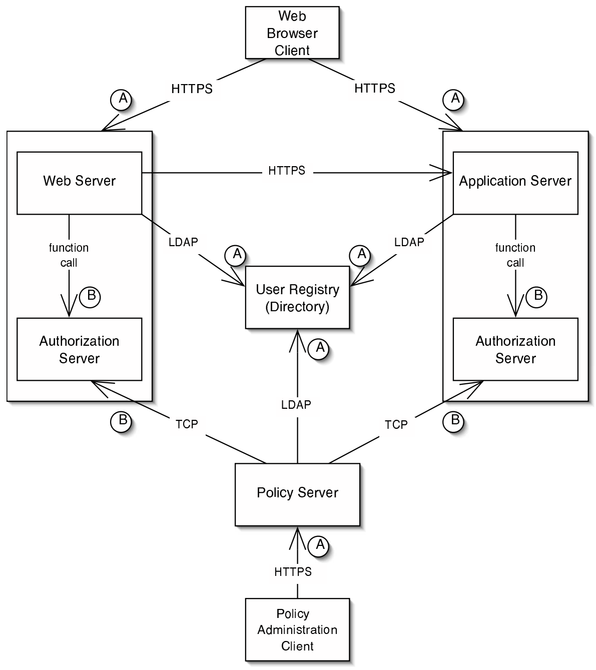

Figure 1. A Security Architecture

nerability of each feature of the software system to security attacks. The threat model [36] identifies assets such as credit card numbers, social security numbers, computing resources, trade secrets, and financial data. The model also identifies and documents threats (*e.g.,* unauthorized access or alteration of assets), as well as ranks each threat according to a scale (*e.g.,* low, medium, high).

The threat identification process is used to determine, for example, whether data can be viewed or changed, who can access the data, and what is deemed as unauthorized access of a system. The threat documentation describes each type of threat and lists counter measures to prevent an attack. Each threat is ranked according to its damage potential (*e.g.,* data, financial loss, property loss or damage), reproducibility of the attack (*i.e.,* the probability that an attempt to compromise the security of a system will succeed), exploitability or discoverability of a threat (*e.g.,* how difficult is it to break into the system), and, finally, who are the affected users (*e.g.,* number of users affected, relative importance of users). Each threat also has a description of threat mitigation factors such as security mechanisms and processes.

An additional aspect of threat identification relates to software configuration management. Although an application or a server, for example, may presently be trustworthy, an exploit in this software may be found in the future. Typically a CERT[2] (Computer Emergency Response Team) advisory, or some other vulnerability knowledge dissemination medium, is used to publicize the exploit so that users and system administrators can upgrade to a new version of the software that mitigates the exploit. However, a new exploit may escape the attention of a user or system administrator, leaving the system vulnerable to a security attack. It is common for security attackers to read the CERT advisories and target systems that run outdated software. In several cases a system may depend on software with or without the knowledge of the user. Since many software packages, especially open source software, have no commercial support, users of the software are responsible for applying security patches and for monitoring their software inventory.

For example, it has been demonstrated that an unpatched iPhone is vulnerable to the `libtiff` exploit.[3] Fortunately, this exploit has been patched by the Apple 1.1.2 update but it serves as an example of the danger of unpatched software, especially on a popular device such as the iPhone. `Libtiff` is a package of functions used to change and view TIFF (Tagged Image File Format) files. A number of vulnerabilities, including buffer overflows, described later, have been reported in `libtiff`. An attacker could create a malicious file that, if opened by a user, would crash the application that is used to open the file.

A worse scenario would involve the exploiting of a security vulnerability to gain root access to the operating system of the device. This type of access can enable attackers to install and execute arbitrary code on the device, including, for example, an application that can be used to record conversations on an exploited iPhone.

Buffer-overflow vulnerabilities, such as the one just mentioned, is one of the subjects of the following section.

3. Secure Coding

There are many security vulnerabilities that arise from coding problems. The coding problems described in this section are: buffer overflows, format string vulnerabilities, and integer vulnerabilities. The length of the description of each of these problems is commensurate to the impact the problem has had on software security. For example, a greater emphasis is placed on security attacks due to buffer overflows because they are the most common type of attack in the last couple of decades, as documented by the Lincoln Labs Intrusion Detection Evaluation and CERT advisories. Buffer overflow attacks, alone, costs the software

[2]CERT Advisories http://www.cert.org/advisories

[3]iPhone `libtiff` exploit can also be used to allow the user to "jailbreak" the iPhone and install applications that have not been approved by Apple http://www.engadget.com/2007/11/16/debunk-yes-virgina-the-iphone-libtiff-exploit-can-also-be-use

industry hundreds of millions of dollars per year. For example, the `bind` software, which is responsible for 95% of the Domain Name System, was discovered to contain a buffer overflow.[4]

The selected coding problems primarily pertain to code written in the C programming language. C code is the overwhelming target of buffer overflow, format string, and integer vulnerability attacks. This is due to the popularity of the language in the development of systems software and its poorly designed memory management and string libraries. Most of the software that is bundled with Linux and Sun Solaris, as well as some of the most popular servers on the Internet are implemented in C.

The three types of coding problems described next represent a small sample of a few of the most common and virulent problems. The reader is referred to a book by Seacord [41] on secure coding in C and C++ for a lucid account of secure coding problems and mitigation strategies.

3.1. Buffer Overflow Attacks

A buffer can be used to store the data that corresponds to, for example, a field on a web-based form to fill in a user's last name. Buffer overflows may occur when a fixed-size memory allocation (*i.e.*, buffer) is used to store a variable-size data entry (*e.g.*, a last name). Problems occur when the variable-size data entry overruns the bounds of the fixed-size memory buffer. These overflows are typically exploited by entering a string that is larger than the size of the buffer assigned to hold it. If the return address (RA) is part of the overwritten run-time stack, and the value of the RA is modified to a value of the memory address containing malicious code, an attacker may execute that code. In Figure 2, '+++++' represents valid data in the buffer and '/////' represents the attacker's data. Note that, because buffers (strings) grow in the opposite direction of the stack's growth, it is easy to overflow the string and overwrite the return addresses on the stack to jump to the malicious payload stored in the string.

To create a buffer overflow attack one must, first, arrange for suitable code to be available in the program's address space and, second, overflow the buffer so that the program jumps to that code.

The first step can be done by either injecting the code into an input buffer or using code that is already embedded in the program or in the program's address space. If the code is injected, it is likely that the attacker uses a string as input to the program and stores the string in a buffer. The string can contain bytes that are native CPU instructions for the attacked platform. The buffer can be located on the stack, the heap, or in the static data area. If the code is already

[4]NIST National Vulnerability Database http://nvd.nist.gov

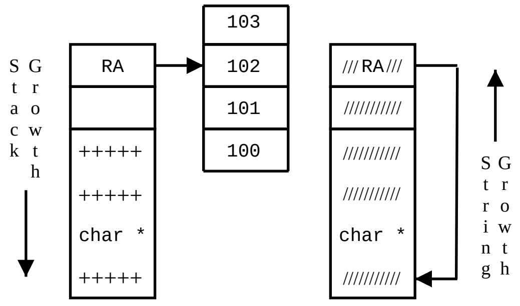

Figure 2. A stack buffer overflow

in the program, the attacker only needs to cause the program to jump to it. The hidden code might be embedded by a hostile programmer or might be code in `libc` such as `exec("/bin/sh")`. Such a hypothetical program would execute the machine instructions for `exec("/bin/sh")` after a stack-based buffer overflow. If this program runs with elevated privileges (*e.g.*, root), then the opened shell will have root privileges.

After the attack code is in the program's address space, the goal of the second step is to have the program jump to the attack code. In addition to overwriting the RA, as previously mentioned, an attacker may branch to the attack code in several other ways. For example, the buffer overflow may cause a function pointer to be re-assigned so that it points to the attack code when the action is invoked. To do this, the attacker would need to find a buffer that is adjacent to a function pointer in the stack, heap, or static data area. The buffer would then be overflowed to change the value of the function pointer so that it jumps to the attack code. Alternatively, if the code features a `setjmp(buffer)` statement to set a checkpoint and a `longjmp(buffer)` statement to go back to the checkpoint, the attacker may corrupt the state of the buffer used by these statements so that `longjmp(buffer)` jumps to the attack code instead.

Buffer overflow attacks are not limited to buffers that are allocated on the stack, attackers can also overflow heap-allocated buffers. However, unlike the stack, the heap does not contain return addresses, making it harder to change the program's control flow [45]. Overflowing a buffer on the heap will corrupt adjacent buffers, which may lead to erroneous behavior during program execution. Such behavior is unlikely to be a security vulnerability [45], but instead may cause the program to crash, hence creating a denial-of-service attack. A more potent heap-based buffer overflow attack needs to identify which variables are security-critical and cause a buffer overflow to overwrite the target variables.

Do not Use Function	Use Function
gets	fgets(buf, size, stdin)
strcpy, strcat	strncpy, strncat
sprintf, vsprintf	snprintf, vsnprintf

Table 1. Some problematic C functions

3.1.1 Buffer Overflow Mitigation

Buffer overflows can be prevented by ensuring that bounds checking is always performed on every buffer. In many C programs, strings are copied into other strings without checking if the target string has enough memory. This is less of a problem in programming languages, such as Java, that have built-in bounds checking. String copying is only one example of how a lack of bounds checking can lead to a buffer overflow attack. A simple `while` loop that reads one character at a time from user input and stores those characters in a buffer until an end-of-file character is read is one of many other possibilities. Table 1 shows some problematic C functions that do not perform bounds checking as well as corresponding preferred functions that perform an equivalent operation with bounds checking.

Some security tools, such as Splint [18] and Coverity Prevent,[5] perform static analysis to find code that is likely to be vulnerable. Splint requires programmers to annotate their source code with constraints. Not all of the existing source code analysis tools require code annotations, however. The tools Flawfinder,[6] RATS,[7] and ITS4[8] examine source code and report possible weaknesses. An overview of these tools, along with a comparison of their capabilities, can be found elsewhere [34]. In general, these tools direct the attention of software maintenance engineers to C/C++ functions that are known to be associated with security problems, such as buffer overflows, and produce a list of vulnerable code statements.

StackGuard [15] can report some buffer overflows immediately after they happen at run time. Specifically, StackGuard inserts code into the application at compile time and a canary value just before the return addresses on the run-time stack. When the function returns, the added code checks if this canary value is still in place. If the canary value is no longer present, a buffer overflow must have occurred. When this happens, the application terminates with a notification. A similar solution to StackGuard is StackGhost [19], which is a modification to the OpenBSD 2.8 kernel on the Sun Sparc architecture.

Richard Jones and Paul Kelly patched the GNU C Compiler (GCC) to perform bounds checking at run time [27]. This approach protects static, stack, and heap allocated buffers. Every buffer has extra information stored in a table, such as its bounds. This table is then referenced to determine if buffer accesses are legal. However, the overhead generated by these references can increase the execution overhead of the application by several orders of magnitude [27]. A related approach by Cordy *et al.* uses the TXL [13] transformation system to create a tool that can analyze a program and annotate it with appropriate assertions automatically. The tool uses source code assertions that are inserted before each subscript and pointer dereference to check, at run time, that the referencing expression specifies a location within the buffer [46]. Prior to this tool, the Gemini [16] tool was developed. Gemini automatically repositions stack-allocated buffers at compile time to the heap using TXL. The transformation preserves the semantics of the program with a small performance penalty. Gemini has been used to transform an entire Gentoo Linux distribution to secure it against stack-based buffer overflow attacks.

Another way to avoid the effects of an exploited stack-based buffer overflow attack to disallow the execution of the run-time stack. This prevents executable code, such as shell instructions that may have been placed on the stack during a buffer overflow, from being executed. One way to get around a non-executable, run-time stack is to perform a heap overflow, followed by a stack overflow. The heap overflow is used to insert the binary instructions for a command shell into the program's executable memory space. A stack overflow is then used to modify the return address of the current stack frame to point to the executable shell instructions in the heap. An on-line tutorial on heap overflows[9] offers an in-depth analysis of this subject.

3.2. Format String Vulnerabilities

Format string vulnerabilities target specific types of C functions such as `printf` and `sprintf`. This family of functions accepts an input string that specifies the format of the output along with an arbitrary number of arguments that correspond to that string, called the variable arguments list. The function pushes the arguments onto the run-time stack and then reads the format string, popping the arguments back off of the stack. If the programmer allows the user to specify the format string at run time, the user could request more data to be popped off the stack than the programmer had originally pushed. This unpredictable behavior could lead to a denial-of-service attack.

[5]Coverity Prevent for C/C++ http://www.coverity.com
[6]D. Wheeler. Flawfinder. http://www.dwheeleer.com/flawfinder
[7]SecureSoftware. RATS http://www.securesoft.com/rats.php
[8]Cigital. ITS4 http://www.cigital.com/its4/
[9]w00w00 on Heap overflows by Matt Conover and the w00w00 Security Team http://www.w00w00.org/files/articles/heaptut.txt

3.2.1 Format String Vulnerability Mitigation

Cqual [42] is a tool that detects format string vulnerabilities through source code analysis. It allows programmers to use two new data type modifiers, tainted (untrusted data) and untainted (trusted data), that can be applied to any data type in the source code. The tool uses a modified compiler on the annotated source code and informs the programmer when mismatches occur between tainted and untainted types.

FormatGuard [14] can be used to detect format string vulnerabilities at run time. It provides protection by using a proxy API composed of C macros that intercept predetermined vulnerable functions. These macros count the number of operands in the format string and the number of arguments passed to the function via the variable arguments list. If these numbers do not match, FormatGuard flags the program as compromised and does not invoke the vulnerable function.

3.3. Integer Vulnerabilities

The CERT advisories indicate that an integer overflow is present in the `xdr_array()` function that is distributed with the Sun Microsystems XDR library. This overflow has been shown to lead to remotely exploitable buffer overflows in several applications. Integer overflows are one of the three kinds of integer errors commonly associated with security vulnerabilities. The other two errors are integer underflow and integer truncation.

An integer overflow occurs at run time when the result of an integer expression exceeds the maximum value of its type, thus wrapping back to the minimum integer for that type. For example a 32-bit unsigned integer goes from 0 to 0xffffffff. If you add one to 0xffffffff, you get 0 again.

Likewise, an integer underflow occurs at run time when the result of an integer expression is smaller than its minimum value, thus wrapping to the maximum integer for the type. For example, the minimum value of a 32-bit signed integer is 0x80000000 (-2147483648) and the maximum value is 0x7fffffff (2147483647). If you subtract 1 from -2147483648, you get 2147483647.

A truncation occurs when one attempts to assign an integer with a larger width to a smaller width. For example, casting an `int` to a `short` disposes of the leading bits of the `int` value. Likewise, adding two 32-bit numbers results in a number that requires 33 bits. The lower 32 bits of the result are written to the destination 32-bit integer and the 33rd bit is signaled out in some other way, usually in the form of a carry flag, which is not accessible by the program.

Integer vulnerabilities can be used to cause security exploits indirectly. For example, an integer overflow can lead to allocating insufficient memory, which can result in an buffer overflow. Similarly, an integer error can lead to excessive memory allocation, which can be used to crash the program. Finally, an attacker may manipulate integer values to cause a program to enter an infinite loop, thus causing a denial-or-service attack.

3.3.1 Integer Vulnerabilities Mitigation

One way to mitigate integer vulnerabilities is via careful coding practices. For example, before adding two integers x and y ($y \geq 0$) one can check for an integer overflow by checking if $(x + y) < x$.

Alternatively, software maintenance engineers may seek the assistance of automated tools. For example, the RICH (Run-time Integer CHecking) tool [7] detects integer-based attacks against C programs at run time. The tool uses program transformation to modify C source code and generates object code that monitors the program's execution to detect integer overflows and underflows. Their experimental results show that programs often have hundreds to thousands of potential integer vulnerabilities.

4. Malware

The previous section covered some of the coding vulnerabilities that can be exploited to create malware. This section describes the most common types of malware and the anti-virus tools that are used to limit their negative impact. A more comprehensive survey of malware can be found in a book on reverse engineering by Eilam [17] and the Metasploit[10] project website.

Malware is any program that works against the interest of a system's user. Malware is developed and deployed for a variety of reasons, such as: (a) back door access, where an attacker gets unauthorized access to a computer system; (b) denial-of-service attack, where, for example, a large number of computers try to connect to a target server simultaneously with the intent to overwhelm and eventually crash it; (c) vandalism, where, for example, a web site is defaced; and (d) theft, where, for example, social security numbers, credit card numbers, or network access are stolen.

Malware appears in various forms such as viruses, worms, and Trojan horses. Viruses are self-replicating programs that must be executed by an unsuspecting user in order to cause harm and propagate. Viruses exhibit a variety of harmful behaviors, such as deleting data from a computer's disk, before they replicate themselves. They attach themselves to executable program files and then propagate into other executable files on the infected system. Examples of infamous computer viruses are: Elk Cloner (first virus in the wild), Brain, and CIH.

Worms are malware that use the network to propagate. They self-replicate like viruses but do not need human intervention to initiate the replication process. A virus would

[10]Metasploit http://www.metasploit.com

need an unsuspecting user to execute it whereas a worm can attack a computer system via an open network port automatically. As a result, worms can spread uncontrollably and rapidly to many computer systems on the same network. A worm can spread by exploiting a security vulnerability such as, for example, a buffer overflow in a software daemon. In other instances, a worm (*e.g.,* MyDoom) may spread via e-mail which is more characteristic of how a virus would spread. Examples of infamous computer worms are: Morris, Melissa, VBS/Loveletter, and Sircam.

Trojan horses are seemingly harmless applications that contain malicious code. Typically, the victim of a Trojan horse receives an infected program, which may masquerade as free software, executes the program, and remains unaware that malware has also been executed. For example, a Trojan horse might be embedded in an application that displays an image. When the application is executed, the image is displayed but the malicious code is executed in the background. Examples of infamous Trojan horses are: Bandook, Beast Trojan, ProRat, and Zlob Trojan.

Malware may immediately perform harm to a computer system by, say, erasing contents from its hard drive, but may also alter the state of the computer system so that it becomes vulnerable to a future attack. One way to alter the state of a system is via the creation of a back door. Back doors create a secret access channel that is used to interact with (*e.g.,* connect, control, spy) a victim's system. Back doors may be embedded into Trojan horses, worms, or viruses. Another way to alter the state of the system is to convert it into a zombie computer known as a bot. For example, the operator of a network of bots (botnet) may send out a worm to infect users with a malicious bot. The bot on the infected computer logs into an Internet Relay Chat (IRC) server, which acts as the command and control center for the botnet. Once the botnet is established, the botnet operator can sell access to the botnet to spammers, who may send instructions via the IRC server to the infected bot computers instructing them to send spam messages to mail servers. The most infamous botnet is the Storm worm botnet.

4.1. Malware Mitigation

Anti-virus programs search for malware by scanning for signatures in the binary code of each program's executable file on a computer system. A signature is a unique sequence of code that is found in a malware's executable file. The anti-virus program maintains a frequently updated database of known virus signatures. Two of the premier anti-virus tools are Symantec[11] and McAfee.[12]

Unfortunately, scanning executable files for malware signatures is an imperfect solution because of polymorphic and metamorphic malware.

[11] Symantec http://www.symantec.com
[12] McAfee http://www.mcafee.com

Polymorphic malware randomly encrypts the malware's executable file with a random key, in order to circumvent the anti-virus program, and later decrypts the file at run time. This strategy defeats signature-based techniques because each copy of the code is different due to the use of a random key. A decryption technique employed by polymorphic malware involves performing an exclusive-or (xor) of each byte with a randomized key that is saved by the virus. The xor operation is convenient because the encryption and decryption routine is the same. For example, $(x\ xor\ k\ =\ y)$ to encrypt byte x using key k and $(y\ xor\ k\ =\ x)$ to decrypt byte y using key k to get back byte x. Examples of infamous polymorphic malware are: 1260, and The Dark Avenger.

To protect against polymorphic malware, anti-virus programs can scan for virus signatures in memory at run time instead of on disk. This mitigates polymorphic malware because the code has to be decrypted before it can be loaded into memory and executed. Since the decryption algorithm is static, anti-virus programs can use the decryption code itself as a signature to detect the malware.

Metamorphic malware are an evolved type of polymorphic malware. Instead of encrypting the program's body and making slight alterations in the decryption engine, metamorphic malware alters, in a semantics-preserving way, the entire program each time it is replicated. This makes it impossible for signature-based anti-virus programs to identify malware. Some of the common alterations of metamorphic malware are: instruction and register randomization, NOP instruction insertion, instruction ordering, reversing (negating) conditions, insertion of "garbage" instructions, and reordering of the storage location of functions. Examples of infamous metamorphic malware are: Zmist and Simile.

5. Research Challenges

This section poses two challenges to the software maintenance research community. The first challenge suggests the creation of formal notations and tools to support the specification and analysis of security architectures. The second challenge suggests the creation of run-time mechanisms to enable software to self-diagnose and self-mitigate security attacks automatically.

5.1. Formal Security Architectures

A significant barrier to the automation of the analysis of security architectures is the absence of formalism employed by the notations that are used in practice to describe security architectures. Nevertheless, there has been considerable discussion by researchers on formalizing software security models dating back to the 1980s [31]. In the past decade, especially with the advent of model checking technology [10], there have been advances in verifying security protocols [33, 11] and proving security properties of applications [6, 5, 4].

Parenthetically, model checking is an automatic technique that explores all possible behaviors of a given model (*e.g.,* a security protocol), which is specified as a finite state machine, and checks safety or liveness properties that are typically expressed using temporal logics or assertions. Model checkers compute the state-space of the model, starting from the initial state of the model to all its reachable states via the state machine's transitions. While the state space of the model is being constructed, a model checker can verify if a given property is satisfied while also generating a counter example if the property is not satisfied by the model.

Recently, there has been considerable use of model checking to solve problems in the domain of software engineering. Specifically, model checking has been applied to models for software process [28], requirements [9], architecture [21, 22], detailed design [1, 25], and implementation [26, 12, 24].

The security architectures of industrial-strength software systems, by and large, are specified using natural language and informal diagrams such as the one shown in Figure 1. Moreover, these security architectures are validated using manual inspection processes, not unlike code review inspections.

The software maintenance community is challenged to investigate the creation of a suite of tools to model and simulate the behavior of security architectures as well as support the verification of properties that are relevant to security architectures.

In an earlier paper by Nicol *et al.* [35], a similar challenge was stated: to develop a sound model-based methodology for quantifying the security that can be anticipated from a design.

Software maintenance researchers can leverage advances in the areas of software architecture [37], software simulation and emulation [40], and model checking [10] to create tools that support security architects in the specification, simulation, testing, and verification of software security architectures. Security architecture models, unlike general software models, are especially amenable to modeling and model checking because these models are typically coarse-grained (*e.g.,* a node represents a server) and, hence, small in size. It is known that one of the primary challenges in applying model checking to models that have many transitions is dealing with the exponential state explosion problem.

Work on modeling and checking security architectures will hopefully mirror other similar specialized and highly visible successes such as the SLAM Project [3] by Microsoft Research. This project resulted in tools that automate the abstraction and checking of Microsoft Windows device drivers. The tools check that the developers of device drivers for Microsoft Windows follow the protocols for published driver APIs. The tools are now part of the Microsoft Windows Device Driver Development Kit, and all submitted Windows drivers must be checked and validated using them.

Meeting the challenge of creating formal notations and analysis tools to support security architectures will likely involve the following ingredients:

Formal modeling notations to specify the security architecture of software systems. Candidate notations would likely be visual in their syntax, to make them consistent with the types of diagrams used in practice, with a mapping to some underlying formalism that is amenable to automatic processing. The model should enable the specification of security properties

Tools to simulate the execution behavior of software security models. Security architects can use simulation modeling tools to explore architecture alternatives from a security perspective. In addition to validating architectural decisions, these tools may facilitate "what if" analyses of alternative designs to quantify the effect of architectural choices on system security. Two candidate technologies to support the modeling and simulation of security architectures are the Generic Modeling Environment (GME) [32] and Ptolemy [8].

Application of model checking tools to verify properties of security architectures. For example, one would ideally be able to prove that the security architecture in Figure 1 would not allow an authenticated but unauthorized user to access a specific function in the application server. A candidate technology to support the model checking of security architectures is Bogor [39].

Similar notations, tools, and processes have been described in other contexts, unrelated to software security, in the software maintenance literature. This puts software maintenance researchers in a favorable position to make contributions toward the solution of the stated challenge.

5.2. Security Attack Diagnosis & Mitigation

Servers and applications that were developed using secure coding practices may still not respond adequately to security attacks at run-time. The capacity of a server to diagnose security attacks automatically before it is exploited is important, especially if the server is used in a safety critical domain.

The automatic self-diagnosis of a security attack is a promising starting point, but it is insufficient for applications and servers that are deployed in contexts where patching and re-starting the software is not a practical option. Self-mitigation strategies are also needed to enable software to continue functioning, albeit in a limited capacity.

Some work related to the automatic self-diagnosis and self-mitigation of a security attack has been done that uses a framework to discover and patch vulnerabilities at run time in response to a network worm [43]. The framework defines a program that discovers when a service offered on a host is vulnerable to a network worm. The vulnerable service is placed in a sand-boxed environment to discover the specific attack the worm is using. The framework then re-engineers the source code of the service so that it is no longer vulnerable to the worm. The patched source code is automatically tested for proper inoculation to the worm, recompiled, and reinstalled.

The software maintenance community is challenged to investigate techniques that will enable software to self-diagnose security attacks and self-mitigate the effects of these attacks at run-time.

This challenge may require that researchers broaden the way they typically think of software maintenance and consider the possibility of software that is self-maintained with respect to security issues. An approach to solving this research challenge is the following:

Develop software sensors that can monitor various aspects about the execution behavior of software. For example, software sensors can measure memory usage, and runtime stack size, as well as monitor system calls for accessing the file system and network, remote calls to other software components, and queries to databases. Aspect Oriented Programming [29] can be used to weave a variety of sensors into appropriate locations in the software code automatically. A key challenge is to minimize the performance effect of executing the software while the sensors are enabled.

Create formal self-diagnostic models based on deviations from a software's expected behavior. For example, at run time, the software might detect that a function is spending an inordinate amount of time executing. This might cause the software to diagnose a buffer overflow attack because its model had previously correlated this execution behavior with that specific type of attack.

The model creation process will involve exercising the system under a variety of typical scenarios and using software sensors to gather data. This data will subsequently be provided, as input, to statistical and algebraic methods to produce formal diagnostic models.

Automatically disable software features and enable security mechanisms at run-time in response to diagnosed security attacks. For example, an application can have built-in run-time mechanisms to disable some of its features, or shut it down completely, as soon as it determines that it is being attacked. Likewise, a preliminary diagnosis of a security attack may prompt a mechanism to start or increase the logging of an application's state changes to facilitate a *post mortem* analysis.

Ideally, applications would be able to evolve their diagnostic and mitigation capabilities over time by learning from their past experiences. Research in autonomic computing [20, 23] and reinforcement learning [44] can provide insights into the solution to this problem.

6. Synopsis

This paper was written with a dual purpose: First, to provide an overview of three aspects of software security, namely security architecture, secure coding, and malware with the intent to inform software maintenance practitioners and to pique the interest of software maintenance researchers in issues that pertain to security analysis. Second, to pose research challenges that entail developing techniques that can support formal security architectures and the automatic diagnosis and mitigation of security attacks on software.

References

[1] R. Alur and M. Yannakakis. Model checking of hierarchical state machines. In *SIGSOFT FSE*, pages 175–188, 1998.

[2] R. Anderson. *Security Engineering (2nd ed): A Guide to Building Dependable Distributed Systems*. Wiley, 2008.

[3] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. Thorough static analysis of device drivers. In Y. Berbers and W. Zwaenepoel, editors, *EuroSys*, pages 73–85. ACM, 2006.

[4] F. Besson, T. Blanc, C. Fournet, and A. D. Gordon. From stack inspection to access control: A security analysis for libraries. In *CSFW*, pages 61–. IEEE Computer Society, 2004.

[5] F. Besson, T. de Grenier de Latour, and T. P. Jensen. Secure calling contexts for stack inspection. In *PPDP*, pages 76–87. ACM, 2002.

[6] F. Besson, T. P. Jensen, and D. L. Métayer. Model checking security properties of control flow graphs. *Journal of Computer Security*, 9(3):217–250, 2001.

[7] D. Brumley, T. Chieh, R. Johnson, H. Lin, and D. Song. Rich : Automatically protecting against integer-based vulnerabilities. In *NDSS (Symp. on Network and Distributed System Security)*, 2007.

[8] J. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt. Ptolemy: A framework for simulating and prototyping heterogenous systems. *Int. Journal in Computer Simulation*, 4(2):0–, 1994.

[9] W. Chan, R. J. Anderson, P. Beame, D. H. Jones, D. Notkin, and W. E. Warner. Optimizing symbolic model checking for statecharts. *IEEE Trans. Software Eng.*, 27(2):170–190, 2001.

[10] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.

[11] E. M. Clarke, S. Jha, and W. Marrero. Verifying security protocols with brutus. *ACM Trans. Softw. Eng. Methodol.*, 9(4):443–487, 2000.

[12] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, Robby, and H. Zheng. Bandera: extracting finite-state models from java source code. In *ICSE*, pages 439–448, 2000.

[13] J. R. Cordy, T. R. Dean, A. J. Malton, and K. A. Schneider. Source transformation in software engineering using the txl transformation system. *Information & Software Technology*, 44(13):827–837, 2002.

[14] C. Cowan, M. Barringer, S. Beattie, G. Kroah-Hartman, M. Frantzen, and J. Lokier. Formatguard: automatic protection from printf format string vulnerabilities. In *SSYM'01: Proceedings of the 10th conference on USENIX Security Symposium*, pages 15–15, Berkeley, CA, USA, 2001. USENIX Association.

[15] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. Stack-Guard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proc. 7th USENIX Security Conference*, pages 63–78, San Antonio, Texas, jan 1998.

[16] C. Dahn and S. Mancoridis. Using program transformation to secure c programs against buffer overflows. In *WCRE*, pages 323–333, 2003.

[17] E. Eilam. *Reversing: Secrets of Reverse Engineering*. Wiley Publishing, 2005.

[18] D. Evans and D. Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, 19(1):42–51, 2002.

[19] M. Frantzen and M. Shuey. StackGhost: Hardware facilitated stack protection. In *Proc. 10th USENIX Security Conference*, Washington, D.C., august 2001.

[20] A. G. Ganek and T. A. Corbi. The dawning of the autonomic computing era. *IBM Systems Journal*, 42(1):5–18, 2003.

[21] D. Garlan and S. Khersonsky. Model checking implicit-invocation systems. In *IWSSD*, pages 23–30. IEEE Computer Society, 2000.

[22] D. Garlan, S. Khersonsky, and J. S. Kim. Model checking publish-subscribe systems. In T. Ball and S. K. Rajamani, editors, *SPIN*, volume 2648 of *Lecture Notes in Computer Science*, pages 166–180. Springer, 2003.

[23] W. W. Gibbs. *Autonomic Computing*. Scientific American, May 2002.

[24] P. Godefroid. Model checking for programming languages using verisoft. In *POPL*, pages 174–186, 1997.

[25] J. Hatcliff, X. Deng, M. B. Dwyer, G. Jung, and V. P. Ranganath. Cadena: An integrated development, analysis, and verification environment for component-based systems. In *ICSE*, pages 160–173. IEEE Computer Society, 2003.

[26] K. Havelund and T. Pressburger. Model checking java programs using java pathfinder. *STTT*, 2(4):366–381, 2000.

[27] R. W. M. Jones and P. H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in c programs. In *AADEBUG*, pages 13–26, 1997.

[28] C. T. Karamanolis, D. Giannakopoulou, J. Magee, and S. M. Wheater. Model checking of workflow schemas. In *EDOC*, pages 170–181. IEEE Computer Society, 2000.

[29] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP*, pages 220–242, 1997.

[30] I. Krsul. *Software Vulnerability Analysis*. Ph.D. Thesis, Purdue University, 1998.

[31] C. E. Landwehr. Formal models for computer security. *ACM Computing Surveys*, 13(3):247–278, 1981.

[32] Á. Lédeczi, A. Bakay, M. Maroti, P. Völgyesi, G. Nordstrom, J. Sprinkle, and G. Karsai. Composing domain-specific design environments. *IEEE Computer*, 34(11):44–51, 2001.

[33] G. Lowe. Towards a completeness result for model checking of security protocols. *Journal of Computer Security*, 7(1), 1999.

[34] J. Nazario. Source code scanners for better code. *Linux Journal*, 2002.

[35] D. M. Nicol, W. H. Sanders, and K. S. Trivedi. Model-based evaluation: From dependability to security. *IEEE Trans. Dependable Sec. Comput.*, 1(1):48–65, 2004.

[36] R. Patton. *Software Testing (2nd ed.)*. Sams Publishing, 2006.

[37] D. E. Perry and A. L. Wolf. Foundations for study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, 1992.

[38] N. Provos and T. Holz. *Virtual Honeypots: From Botnet Tracking to Intrusion Detection*. Addison Wesley, 2007.

[39] Robby, M. B. Dwyer, and J. Hatcliff. Bogor: an extensible and highly-modular software model checking framework. In *ESEC / SIGSOFT FSE*, pages 267–276. ACM, 2003.

[40] D. C. Schmidt. Model-driven engineering. *IEEE Computer*, 39(2):41–47, 2006.

[41] R. C. Seacord. *Secure Coding in C and C++*. The SEI Series in Software Engineering. Addison-Wesley, 2005.

[42] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting format string vulnerabilities with type qualifiers. In *SSYM'01: Proceedings of the 10th conference on USENIX Security Symposium*, pages 16–16, Berkeley, CA, USA, 2001. USENIX Association.

[43] S. Sidiroglou and A. D. Keromytis. A network worm vaccine architecture. In *WETICE*, pages 220–225, 2003.

[44] S. Thrun and M. L. Littman. A review of reinforcement learning. *AI Magazine*, 21(1):103–105, 2000.

[45] J. Viega. *Building Secure Software*. Addison Wesley, 2002.

[46] L. Wang, J. R. Cordy, and T. R. Dean. Enhancing security using legality assertions. In *WCRE*, pages 35–44. IEEE Computer Society, 2005.