Static and Dynamic Analysis Tools for Improving Software Security

Project Report Version 1.3

Submitted on: November 5, 2004
Course Project: CS4235A

Members:
Forkner, Kyle
Macwan, Heena
Wierzbowski, Paul

Table of Contents

Introduction....................3
Static Analysis of Source Code....................4
Goals in Static Code Analyzer Design for Secure Software Development.................... 5
Detecting Invalid Pointer Dereferences in C.................... 5
Tradeoffs for This Pointer Dereference Static Code Analyzer.................... 7
MOPS: an Infrastructure for Examining Security Properties of Software....................7
Tradeoffs for This Model Checking Static Code Analyzer.................... 9
CSSV: A Tool to Uncover String Manipulation Errors....................10
Tradeoffs for This Contract Checking Static Code Analyzer.................... 11
Other Forms of Software Security Analysis.................... 11
The Binary Object File Format.................... 12
Static Binary Analysis through Disassemblers.................... 14
Dynamic Binary Analysis Using Debuggers....................16
Dynamic Binary Analysis through Simulators and Emulators.................... 17
Integrating Software Security Tools into the Development Process....................18
Conclusion.................... 20
Works Cited.................... 22

Introduction

When building creating software, developers face the difficult challenge of creating bug-free, security-conscious code. Since software flaws form the base of security vulnerabilities, tools and education are obviously needed to help programmers with their difficult task. Many exploits result from common coding errors are easy to discover and correct if found before release. Buffer overflow exploits and string format bugs are just two common vulnerabilities that can be fixed if code review processes discover them. Unfortunately, many organizations leave security as an afterthought, placing functionality and project deadlines ahead of security reviews that may catch problems before they can be exploited. In an attempt to remedy this problem, researchers have created a set of tools that will analyze both source and binary files for software vulnerabilities. When used alongside sound software development practices, these tools can significantly reduce security flaws in programs before they reach production status. Static source code analysis tools review source files in high level languages such as C or C++ in an attempt to find coding flaws which lead to security vulnerabilities. Similar to a compiler's warning messages, these tools alert developers to problem areas, and can even suggest solutions. Binary analysis tools can diagnose already created object files when source code may not be available. These tools must disassemble the object file before they can analyze instructions or flow control to identify possible vulnerabilities. By using source and binary file analysis tools during the development process, software developers can create more secure code that will result in fewer exploits and improved overall security.

This paper will describe static analysis tools that review source code to discover flaws. For each tool described, an overview of the approach, as well as any drawbacks, will be discussed. This paper will also define and discuss static and dynamic binary analysis tools to identify flaws in already compiled programs. Finally, this paper will highlight how the incorporation of these tools into the software development process will create more secure products.

Static Analysis of Source Code

Over the past several years, there has been a lot of research and development in static code analysis tools for ensuring software security. These analyzers review program source code to discover any flaws that might lead to security vulnerabilities and attempt to either: prevent and correct these errors, or alert the programmer of their presence. Just as there are many ways to produce software security flaws through poor coding, there are also many methods and approaches for static code analyzers to try and correct them. Consequently, there are innumerable amounts of research and products for static code analyzers that claim to fix certain software security flaws through differing technical means. Since a survey of these analysis techniques would be simply too much to cover with any reasonable depth, this paper will present a few different approaches for solving various sections of software security flaws. These important approaches are protecting C programs from invalid pointer dereferences, identifying variances from rules of safe programming practices, and identifying possible unsafe string manipulation in C.

Goals in Static Code Analyzer Design for Secure Software Development

Before describing the static code analyzer projects chosen for this paper, the various goals that they are trying to achieve should be established first. These goals will help compare the effectiveness of the various methods of a particular static code analysis approach. A static code analyzer should have the following features or characteristics:

- An ability to cover a wide variety of software security flaws in coding.
- Compatibility over various environments and programming languages.
- Little overhead in program execution, if the analyzer makes changes in the code.
- Require only minimal manual effort from the programmer.

In reality, no static code analyzer can cover all of these aspects completely. Each of the following projects and approaches had to make some tradeoff decisions to ensure their effectiveness in observing and correcting security software flaws in source code.

Detecting Invalid Pointer Dereferences in C

In 2003, Suan His Yong and Susan Horwitz from the University of Wisconsin-Madison presented a research paper describing their static code analysis tool. The tool identifies and tracks pointer dereferences and memory locations within C programs (Horwitz 307). From the results of this static analysis, checks are inserted into the program. At runtime, if the program runs into an unsafe dereference, then the potential security violation is reported, and the program halts (Horwitz 308). These researchers felt that focusing on pointer dereferences can address a large scope of software security vulnerabilities in C programs (Horwitz 307). They cite that by ensuring proper pointer dereferencing, one can prevent exploitation of vulnerabilities such as buffer overruns,

changing system-call arguments, or corruption of data to cause a denial of service (Horwitz 307).

This security tool essentially goes through two major phases: a static code analysis in order to identify unsafe pointers and memory locations to track; and an implementation of runtime check routines into the C source code that will halt the program in the event of an unsafe pointer dereference (Horwitz 308, 310). The static analysis itself goes through three steps. In the first step, the security tool does a "points-to" analysis for each variable present in the program (Horwitz 309). In this step, the goal is to determine the set of memory locations that each variable can point to at any time during program execution (Horwitz 309). Because the C language allows casting, any value can be copied into a pointer and will be reviewed by this "points-to" analysis (Horwitz 309). The second step is to identify the unsafe pointers defined as: being able to refer to invalid memory at runtime, or being dereferenced for writing or memory deallocation (Horwitz 310). This identification is to help the program monitor only relevant pointers and save on overhead (Horwitz 310). The final step is where tracked memory locations are identified for the second phase of the approach (Horwitz 310).

During the second phase, each tracked memory location's allocation site is marked as *appropriate* while each tracked memory location's deallocation site is marked as *inappropriate*. Prior to each write through an unsafe pointer dereference, or each call to deallocate memory, the "pointed-to" location is checked, and if marked as *inappropriate* then the program is halted with an error message (Horwitz 310). These runtime checking routines are implemented in the source as special C macros and functions (Horwitz 310).

Tradeoffs for This Pointer Dereference Static Code Analyzer

This project is an example of a static code analysis approach that focuses on a particular language operation that is known to be handled poorly. In this case, the operation is dereferencing a pointer in C, a prevalent and powerful operation with little safeguards. While this approach does not allow for a great deal of compatibility among various environments and languages, it does cover a substantial amount of security flaws in a widely used programming language. Furthermore, according to these researchers' results, there is little overhead in executing time and speed, and the security tool does not require much effort for the programmer to run.

However, there are several downsides to this approach. According to the researchers' report, when a C program uses library functions, the tool cannot guarantee that it will run its algorithms correctly unless the programmer makes the effort to write wrappers and static models for the C library functions. Unfortunately, many C programs rely on library functions, presenting major hurdles for security-conscious programmers. Also, the static analysis is only used to spot potential security flaws, but does not necessarily correct them in the code. It only provides the means to stop exploitation of such vulnerabilities.

MOPS: an Infrastructure for Examining Security Properties of Software

In 2002, Hao Chen and David Wagner from University of California at Berkeley presented a research paper describing a static code analysis tool called MOPS, which stands for *MOdelchecking Programs for Security properties*. This project is far more technically complicated than the previous project from Wisconsin. This security tool

focuses on identifying rules of safe programming practices, and then verifying whether these safety properties are being obeyed within a program (Chen 235).

This security tool represents known security properties as finite state automata, and uses model checking techniques to identify whether there are any violations of the security properties that are reachable in the program (Chen 235). These rules or security properties are already defined in a database that the security tool will check during analysis (Chen 237). These security properties are defined to be the order of a sequence of security-relevant operations, which tend to be system calls from the operating system (Chen 237). An example can be seen in the following figure.

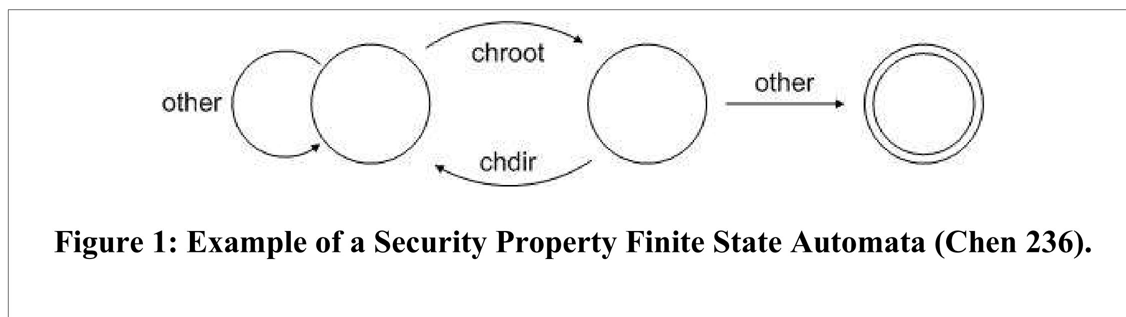


Figure 1: Example of a Security Property Finite State Automata (Chen 236).

In this example, the *chroot* call is used by a system to confine access to a sub file system. The *chdir* call should be immediately made afterwards to change its working directory to the root of the created sub file system. If the *chdir* call is not immediately made, then a user can gain root access to other parts beyond the sub file system by using commands listed in the diagram as "other." This unsafe sequence possibility is noted in the automata by the double-circled state.

The pushdown automaton model for the program is inspired by how a program runs in an operating system with a pointer and a stack, where the pointer points to the program position of the next executable statement and the stack records the return

addresses of all unfinished function calls (Chen 239). A pushdown automaton can be realized if the pointer and the stack are merged by regarding the pointer as the top element on the stack (Chen 239). Once these two automaton models are formed, model checking algorithms are used to see if a security violation state, as defined by the security properties automata, is reached within the program (Chen 237). If such a violation is reachable, a notice is made to the programmer who is using this tool to audit his program (Chen 237).

Tradeoffs for This Model Checking Static Code Analyzer

This project represents an approach in static code analysis that focuses on modeling a program after operating system calls that are known to be prone to security vulnerabilities. More specifically, this approach relies on pattern recognition of safe and unsafe coding practices in security-critical programs. In this case, what is safe or unsafe is the order of input parameters for certain operating system calls in a program.

The researchers of this program are hoping that the project will have an extensible security property database that will be provided to programmers who want to use this code-auditing tool (MOPS). This approach actually reduces the effort a programmer needs to expend in ensuring their program's security, since they would only have to look over the analysis made by this tool and make the proper corrections. Furthermore, since this project focuses on defining various rules of secure programming, the analysis will be helpful to the programmer alerting them to their common mistakes.

In terms of coverage of vulnerabilities and compatibility across platforms, this project might have some serious issues to consider. Its effectiveness is completely dependent on its database of security rules. If a programmer is not developing on a

specific language platform supported by this tool, they are forced to review their own code for security flaws.

CSSV: A Tool to Uncover String Manipulation Errors

In 2003, Nurit Dor, Michael Rodeh, and Mooly Sagiv, a group of researchers from Tel-Aviv University and IBM Research, have developed a tool called CSSV, which stands for *C String Static Verifyer* (Dor 155). These researchers decided to focus on string manipulation as it is a common source of software defects and security vulnerabilities, namely buffer overflows (Dor 155). They claim that this tool is able to uncover all string manipulation errors in a C program while producing few false positives (Dor 155).

The tool uses static code analysis in conjunction with a contract system that includes information on a procedure's precondition, post-condition, and potential side-effects (Dor 155). A contract can either be automatically derived by the security tool or manually entered by the programmer in CoreC, a simplified subset of the C programming language (Dor 155). With these contracts, CSSV goes through four phases in its static code analysis. In the first phase, automatic and manual contracts for each procedure are integrated into the code (Dor 155). In the second phase, the security tool runs a string pointer analysis (Dor 156). The results of both the first and second phase are used to create an integer analysis program in the third phase that checks the contracts against the pointer analysis results (Dor 156). The fourth phase then uses this integer analysis program to determine string manipulation errors resulting from broken contracts (Dor 156). If a string manipulation error occurs, the fourth phase will note it in a list of potential errors for the programmer to review (Dor 156).

Tradeoffs for This Contract Checking Static Code Analyzer

This project is representative of an approach that has a focal scope on procedures within programs. By using the contract concept, this approach is able to capture the intent of the programmer and verify it against the actual programming of the procedure. In this specific case, the researchers decided to keep the focus of these contracts to a specific class of functions and data types, namely strings. There are some advantages and disadvantages to this approach. Of course, by focusing on strings, this project can work on detecting a large class of security vulnerabilities, but at the same time, it makes no effort to analyze other possible software security problems. Another concern is the reported fact that this tool allows some false positives in the attempt to detect all string manipulation errors. While from a security point of view this may seem acceptable, there is a serious issue to consider from a development point of view. Since this tool only reports potential error, it still places the burden of manually changing the program back on the programmer. Therefore, any false alarm is only going to place an unnecessary burden to the software development cycle.

Other Forms of Software Security Analysis

Despite these advances in source code analysis tools, many flaws may still find their way into production releases. Because of these problems, researchers have created tools to analyze software security in binary files where source code may not be present. Many researches or software developers are comfortable with both source files, and their binary counterparts. Unfortunately, many people may only be familiar with applications that are distributed and run from a binary format. Many corporate or home users would

not understand program source files, but can execute their binary results. Microsoft and other software companies distribute their products exclusively in binary format. With a business plan based on software, they are forced to release binary only versions to maintain their intellectual property. Many open source distributions of UNIX also choose binary distribution of their software for its ease of use and speed during setup. Compiling certain features such as window managers from source can take days on lower end machines that are common homes for these free distributions. Binary files, once compiled for a particular architecture, are easily distributable via CD installation, or setup programs which extract the needed files. Since the binary will contain needed libraries, routines, and files to work properly, companies can produce a master that will be copied for distribution.

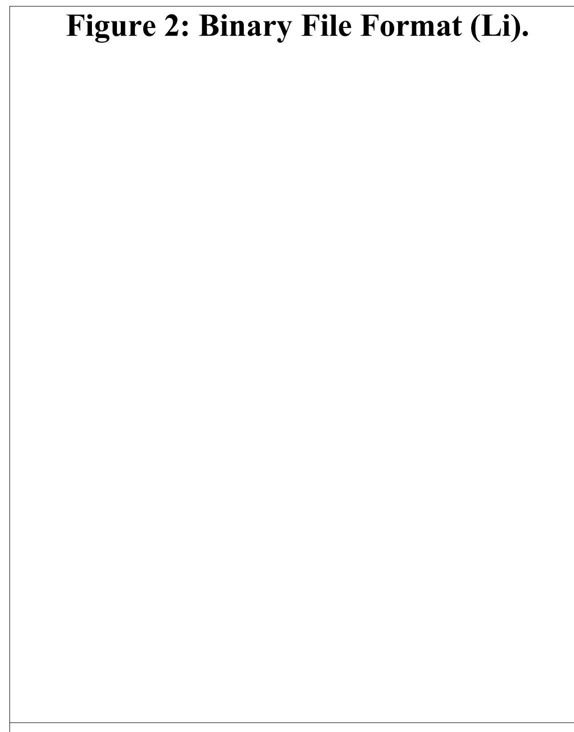
Unfortunately, these features that make binary files so easy to use also present them as an excellent target for viruses, trojans, and spyware. The executable nature of binary files helps malware products launch their installation packages that imbed the pesky code deep into a system. As with most binary products, viruses and other malware do not provide the user with source code that can be analyzed for security flaws. To combat these problems, researchers have developed static and dynamic binary analysis tools to diagnose the security problems which may be present in binary files. The most important tools analyzed here include disassemblers, debuggers, and emulators.

The Binary Object File Format

Before attempting to create these analysis tools, researchers had to become familiar with the binary object file format. Luckily, though binary object files differ

depending on system architecture, all contain five basic types of information as described by the following figure.

Figure 2: Binary File Format (Li).



First, header information describes the object file, often containing the name, size, and creation date of the file (Li). Second, relocation information provides a list of addresses that will need to be translated when the object is loaded into memory at a different location than originally intended (Li). This information can include important jumps to subroutines and libraries created by the developer. Third, symbol information (used primarily by the linker or loader) describes import or export modules that work with the systems kernel during operation (Li). Fourth, debugging information includes symbols, descriptions of data structures, and source information to be used by debuggers or reporting programs in case of an application crash (Li). Finally, many sections of code

and data are included which provide the actual instructions generated from source files (Li).

Common binary formats which many users may be familiar with include the MS-DOS .COM and .EXE format, the UNIX a.out and ELF formats, and the Portable Executable (PE) format used by the Windows NT family (Li). Focusing on the Windows PE binary format, a popular home for malware, Li describes seven predefined sections, . text, .bss, .rdata, .data, .rsrc, .edata, .and idata (Li). As described in the Microsoft Developer Network (MSDN) documentation, the .text section contains "all general-purpose code emitted by the compiler or assembler." (Pietrek) The .bss section stores uninitialized static or global variables, while the .rdata section contains a resource tree that can include cursors and images, and a description string for the executable (Pietrek). The .data section holds all initialized global and static variables including defined strings and messages (Pietrek). The .rsrc section provides links to all required resources for the program including all images, menus, or dialogs for user interaction (Pietrek). The .edata section defines all functions or data that must be exported to other modules, while the . idata section defines a list of imports that must be present during runtime (Pietrek). Based on the knowledge of binary file formats, researchers have been able to create binary analysis tools to inspect files for security flaws.

Static Binary Analysis through Disassemblers

Disassemblers are used to statically analyze binary object files. This reverse assembly process attempts to decompose the binary object file back into human readable assembly or high-level language code. This static analysis approach, used by

disassemblers such as the Win32 Program Disassembler*, has several advantages over dynamic methods described below. Most importantly, the time of disassembly is proportional to the number of code lines, whereas dynamic analysis can be slowed extensively by iterations over large data sets. Also, disassemblers provide the user with a global view of the program whereas debuggers can only look at a single section or module (Li). Despite these advantages, correct disassembly of a binary object remains a difficult problem to solve. The most difficult section of the disassembly process is distinguishing code from data segments. This problem arises because of performance improvements generated by modern compilers. During the build process, many compilers compact data and instructions into the same space to save on overall file size. This process, though providing an important benefit, makes disassembly difficult because of disguised instruction boundaries (Li). Additionally, a disassembler cannot access dynamic instructions that may only be present at runtime. Because of this problem, disassemblers fail to correctly analyze the control flow of algorithms. Finally, disassembly is difficult because of variable instruction size present in most architectures (Li). Optional parameters for instructions, coupled with intermixed data from optimizations, form a difficult hurdle for disassemblers to overcome. To solve this problem, two algorithms have been created: linear sweeping, and recursive traversal (Li). The linear sweeping approach sequentially reads in binary bytes and attempts to match them to instructions (Li). This method is particularly vulnerable to the data and instruction mixing problem described above because of its simple scanning approach. Once an error in disassembly has occurred, the scanner will continue to report incorrect instructions until a series of bytes fails to match any possible instruction (Li). The

* Win32 Program Disassembler - http://www.geocities.com/~sangcho/disasm.html

weaknesses of the linear sweeping algorithm stem from its failure to account for the control flow information present in the binary. In an attempt to use this information from the program, the recursive traversal algorithm attempts to disassemble all possible instructions along branches of a program (Li). The recursive traversal approach, though slower than linear sweeping, can often provide a better overall result for the disassembly process. This algorithm attempts to completely follow the instructions on both sides of a branch to provide a complete view of the parsed function (Li). Based on the short comings of static disassemblers that stem from difficulties in properly identifying instructions, researchers have created debuggers and simulators to study dynamic running processes for security flaws.

Dynamic Binary Analysis Using Debuggers

Dynamic binary analysis tools such as debuggers have the advantage of being able to see context information available only during runtime. This information includes memory addresses, register data, and stack or heap areas, which can be observed and used to fill gaps during analysis. Debuggers instruction stepping process allows execution to be interrupted by a user, who can then control function execution on a line by line basis (Li). This stepping process allows for diagnosis before and after each instruction that can pinpoint problems or flaws. The basic functionality of a debugger, such as GDB[*], is to set single or multiple break points at which a special interrupt command is inserted. During execution, the program runs as normal until the interrupt is reached and pauses execution. After the interrupt has been caught by the debugger, the user directly controls execution of each instruction, or steps over certain troubling code sections (Li). At each

[*] GDB – The GNU Project Debugger - http://www.gnu.org/software/gdb/gdb.html

new instruction, the user can view the function context and all variables before execution, execute the next instruction, and view any resulting changes to variables or settings. To provide accurate context information, many debuggers trace execution back through procedure calls and stack frame data. By searching the names of procedures in the stack, and analyzing register data, the debugger can match a name to each instruction to help the user during the analysis (Li). Despite the advantages of a debugger, some problems or flaws may go unnoticed. Some flaws, such as logic bombs, may not execute at the time of the analysis. During the testing process, users may fail to properly scan the code, and execution continues as expected until specified conditions are met and the flaw is revealed. Users may also fail to find variables that have been overwritten with incorrect data that still follows data integrity constraints (Li). A different price for a shopping cart item could go undetected unless users closely monitor program variables. Despite these problems, debuggers provide a useful instruction level analysis of binary files.

Dynamic Binary Analysis through Simulators and Emulators

Another approach to binary analysis is the use of simulators or emulators to mimic instructions, and their effect on particular architectures. Li, a researcher in binary code analysis tools, states, "A code simulation tool is a software program that simulates the hardware execution of the test program by fetching, decoding, and emulating the operation of each instruction." Emulators provide similar features but attempt to run programs designed for a different architecture than the current system. Proper simulators and emulators, such as SimOS*, can mimic the hardware, processor, register, memory, and I/O operations for their specific architecture (Li). Simulators can be run in safe

* SimOS – The Complete Machine Simulator - http://simos.stanford.edu/

environments to analyze a program and its interactions with the system. Using this approach, researchers analyze malware interaction and spreading method to inform public users. One problem common to most simulators or emulators is speed of overall execution. Hardware vendors, like software vendors, rarely provide creators with the specifications needed to properly implement their simulators. Because of these problems, simulators often chain known instructions together to create results that match a single instruction on proprietary hardware. This speed problem, combined with the expense of creating a simulator for each architecture, continues to be a major problem for simulator creators and users.

Integrating Software Security Tools into the Development Process

In order to minimize software security problems, developers must incorporate these analysis tools into their development cycle. The following diagram shows how security practices should be applied to different phases in the software security life cycle.

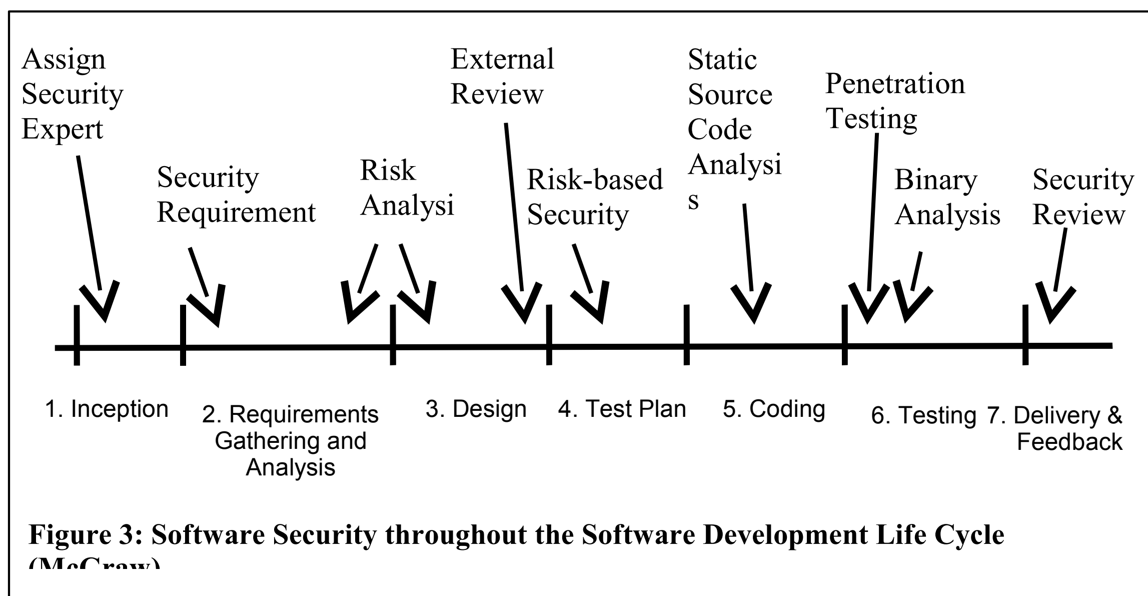


Figure 3: Software Security throughout the Software Development Life Cycle (McGraw)

Phase 1: Inception

This phase is typically used to identify the problem that will be solved by the software application. Security is rarely a topic of discussion unless the problem is a security protocol itself. The assignment of a security expert during this phase can prove beneficial by identifying security concerns in later stages (McGraw).

Phase 2: Requirements Gathering and Analysis

During this phase, the identified problem is studied thoroughly and aspects such as user expectations and system features are identified and clearly defined. Any security requirements should be identified in this section as well. Risk analysis couples strongly with the requirement gathering phase in order to track the various decisions being made. Security considerations during this phase will save developers time and companies money when compared with the constant patch and re-release process that is often used today (McGraw).

Phase 3: Design

In this phase, important decisions are made that influences the quality of the product. Risk analysis is also important during this phase. All the system and security requirements must be considered during this stage (McGraw). After completion, an external review of the design may discover problems not considered by the development team. Again, the cost changes before later steps will save the company money in maintenance and modification costs.

Phase 4: Test Plan

Based on the software design, a test plan is generated to formalize the procedures and resources that will be used in testing the software components and complete system (McGraw). Plans for security testing should be created by the team's security expert and incorporate the source and binary analysis tools discussed above.

Phase 5: Coding

During this phase, actual code for modules and sections is developed. Static source code analysis tools should be integrated with the development environment to help the developers locate possible security issues before testing. The automated static analysis tools can quickly identify problems, allowing developers to fix bugs before they reach the testing phase. Again, any changes before a production release will save companies money they would have spent to maintain or modify flawed programs (McGraw).

Phase 6: Testing

Along with the regular software testing, a special team should be assigned to perform security and penetration tests on the developed software (McGraw). A structured repository of all the application vulnerabilities should be maintained and used to develop future test cases for similar products. During this testing phase, both source and binary file analyzers can be used to automate the testing process. By speeding up this step, management can achieve important deadlines while still producing secure, quality code.

Phase 7: Delivery & Feedback

The final software product is delivered and thereafter it is exposed to many kinds of attacks. Any successful attack on the system should be recorded so that a patch or a new version of the software can be developed and delivered. A post exploit report should be created so that developers can reflect and identify the flaw.

Conclusion

Throughout the software development process, creators must keep software security in mind. By integrating common security practices into the development cycle, fewer errors will be created leading to improved overall security. However, before software products can improve their security, companies must emphasize it as much as product features or overall quality. Repeatable development processes and developer education will help companies and teams create consistent products that are both high quality and provide excellent security. By using the static analysis tools to identify common problems such as buffer overflows and string format exceptions in source files, companies can catch problems before attackers get the opportunity to exploit the product. Similarly, using binary analysis tools on software packages can identify flaws that need to be patched before their exploitation compromises important systems. By integrating these tools into the development cycle, companies will create better software, which will increase security to both systems and their users.

Works Cited

Chen, H. and D. Wagner. “MOPS: an Infrastructure for Examining Security Properties of Software.” Proceedings of the 9th ACM Conference on Computer and Communications Security, Novermber 18-22, 2002, Washington, D.C., USA. New York, NY: ACM Press, 2002. 235-244.

Chen, Hao, and David Wagner. MOPS. University of California at Berkeley. 1 Nov. 2004 <http://www.cs.berkeley.edu/~daw/mops/>.

Dor, N., Rodeh, M., and M. Sagiv. “CSSV: Towards a Realistic Tool for Statically Detecting All Buffer Overflows in C.” Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation, June 9-11, 2003, San Diego, California, USA. New York, NY: ACM Press, 2003. 155-167.

Horwitz, S. and S.H. Yong. “Protecting C Programs from Attacks via Invalid Pointer Dereferences.” Proceedings of the 9th European Software Engineering Conference Held Jointly with 10th ACM SIGSOFT International Symposium on Foundations of Software Engineering, September 1-5, 2003, Helsinki, Finland. New York, NY: ACM Press, 2003. 307-316.

Li, Shengying. A Survey of Tools for Binary Code Analysis. Stoney Brook University. 24 August, 2004. <http://www.ecsl.cs.sunysb.edu/tr/BinaryAnalysis.doc>.

McGraw, Gary. “46.1 Exploiting Embedded Software.” Session 46 of the 41st Annual Design Automation Conference. June 7, 2004. <http://videos.dac.com/41st/slides/46-2.ppt>.

Pietrek, Matt. “Peering Inside the PE: A Tour of the Win32 Portable Executable File Format”. MSDN. March, 1994. November 1, 2004. <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndebug/html/msdn_peeringpe.asp>.