STANDARDIZING SOURCE CODE SECURITY AUDITS

Suzanna Schmeelk[1], Bill Mills[2] and Leif Hedstrom[3]

[1]Department of Technology Management – Cyber Security, NYU-Poly, USA

sschme01@students.poly.edu

[2]Yahoo! Sunnyvale, California, USA

wmills@yahoo-inc.com

[3]Apache Foundation

zwoop@apache.org

ABSTRACT

A source code security audit is a powerful methodology for locating and removing security vulnerabilities. An audit can be used to (1) pass potentially prioritized list of vulnerabilities to developers (2) exploit vulnerabilities or (3) provide proof-of-concepts for potential vulnerabilities. The security audit research currently remains disjoint with minor discussion of methodologies utilized in the field. This paper assembles a broad array of literature to promote standardizing source code security audits techniques. It, then, explores a case study using the aforementioned techniques.

The case study analyzes the security for a stable version of the Apache Traffic Server (ATS). The study takes a white to gray hat point of view as it reports vulnerabilities located by two popular proprietary tools, examines and connects potential vulnerabilities with a standard community-driven taxonomy, and describes consequences for exploiting the vulnerabilities. A review of other security-driven case studies concludes this research.

KEYWORDS

Cyber Security, Vulnerability Analysis, Source Code Analysis, Apache Traffic Server, C/C++, CWE

1. INTRODUCTION

A source code security audit is a powerful methodology for locating and removing security vulnerabilities [33], [29], [34], [24], [10]. An audit can be used to (1) *white hat*: pass potentially prioritized list of vulnerabilities to developers (2) *black hat*: exploit vulnerabilities and (3) *gray hat*: provide proof-of-concepts for potential vulnerabilities [22].

In this paper, we examine the security of the stable version of the Apache Traffic Server(ATS), v.2.0.1, released on September 1, 2010 and built on Ubuntu 9.10 [4]. This research is intended to exemplify and standardize the audit process and therefore takes a white to gray hat point of view.

1.1. Vulnerabilities

Vulnerabilities are, according to the Common Vulnerabilities and Exposures (CVE) [39], "a mistake in software that can be directly used by a hacker to gain access to a system or network." The CVE considers vulnerability a state in a computing system that allows an attacker: to execute other users' commands, access unprivileged data, masquerade as another entity and/or affect the

DOI : 10.5121/ijsea.2012.3101

systems service. Expanding the CVE notion of vulnerability, we can say that a vulnerability is any aspect of the software that can be used to exploit the confidentiality, integrity and/or availability (CIA) of the system.

In many cases vulnerabilities arise from programming errors and misinterpreted function usage. Vulnerabilities are faults, but not all faults are vulnerabilities. A defect, or fault, is defined by IEEE as “a product anomaly [1]." As faults are found at large, it is essential to use a standard language to discuss found faults using a standardized fault taxonomy. There are many fault taxonomies, including: Beizer [6], [7], Zitser [56], Kratkiewicz [28], SAMATE [40], this paper standardizes security faults using the community-driven taxonomy, the Common Weakness Enumeration (CWE) [41]. CWE is currently maintained by the MITRE Corporation [38] with support from the Department of Homeland Security.

Vulnerabilities are expensive and can lead to system exploitation [44]. Heffley and Meunier [23] reported growing annual costs at a half million dollars in 2002. In fact, exploited vulnerabilities can cost companies millions of dollars in data loss [22]. Organizations such as iDefense and Zero-Day Initiative (ZDI) are specifically designed to report and compensate finders for vulnerabilities. iDefense claims that since its launch in 2005, over 2K cases have been created. Subsequently, patching vulnerabilities can cost a lower-bound of $100K [23] and can leave an application vulnerable for weeks [22]. Therefore, fixing vulnerabilities pre-deployment is optimal for both software producers and consumers.

1.2. General Vulnerability Remediation

General security-vulnerability remediation follows traditional fault-removal techniques. Removal depends on detection and severity-level. Depending on detected vulnerability severity, in the worst case, the complete underlying architecture may need to be improved/re-interpreted. Code modularity and federation remain critical components to developing and maintaining robust software.

2. VULNERABILITY REMEDIATION VIA AUDITING

This section fills a literature void by compiling auditing literature and practices into a single useful process. The techniques discussed can be used as necessary on an individual project-specific basis.

2.1. Manual Analysis

Standard Manual security analysis is traditionally the first auditing phase. During this phase the questions which are addressed include: *Is there any relevant background information to examine? What is the audit purpose? What are the code base quantitative and qualitative characteristics?* and *What are the attack surfaces?*

2.1.1. Background Information

Relevant background information includes collecting relevant platform exploitation issues, collecting architectural and/or design documentation (from READMEs, wikis and external les) for potential exploitable issues and exploitable issues associated with the platforms on which the code is executed [8].

2.1.2. Audit Purpose

Determining the audit purpose is a key element into both what code will actually be further examined as well as for determining the type of analysis (white, gray or black).

2.1.3. Quantitative and Qualitative Code Characteristics

The quantitative source code characteristics are essential for understanding what is being examined. This question estimates the lines-of-code (LOC) and collects knowledge of third-party libraries, churn (size of changes between versions), McCabe's cyclomatic complexity [36], nesting complexity, and number of languages to be analyzed [51]. The qualitative source code analysis explores and notes whether annotations are present within the code.

2.1.4. Attack Surfaces

Finally, exploring the attack surfaces analyzes sources of input [22]. It also determines if security mechanisms and/or state management are already in place. Once the attack surfaces have been identified, a further qualitative analysis needs to be performed noting specific programming behaviour. Behaviours including user-supplied input assumptions, lack of sanitization of user-supplied input, lack of function return-value checks, lack of variable initialization, presence of jump or function pointers and unsanitized use of user-supplied data—all need to be examined in detail.

2.2. Auditing Tools

Auditing tools can be beneficial for flagging potential security vulnerabilities. This is the next phase in vulnerability remediation provided there are tools for the language under analysis. Source code static analysis has recently emerged as a standard means to analyze a program for potential security-related faults at compile time [55], [48], [35], [11]. Static analysis is “the process of evaluating a system or component based on its form, structure, content, or documentation [1]." Typically static analysis is divided into formal and non-formal methods. Proprietary tools such as Coverity, Klocwork, Fortify, Parfait and open source tools such as FindBugs have improved analysis tool reports. A static analysis tool can be used to find a lower bound on potential program faults. Theoretically, if one fault exists per three thousand lines of code (KLOC), then there would be over a thousand faults in a code base of three million lines of code (MLOC) [16].

2.2.1. Formal Methods

Further formal methods, or program correctness techniques, are mathematical techniques for evaluating if a program satisfies its formal specification for all its possible inputs [48]. Dwyer, et al. [19], defined formal software analysis as “a mathematically well-founded automated technique for reasoning about the semantics of software with respect to a precise specification of intended behaviour for which the sources of unsoundness are defined. “Program correctness is a system's ability to perform its job according to the specification (correctness) and to handle abnormal situations (robustness) [37]. Formal methods benefits come with some costs. First, the overhead for program specification can be quite great: 1+ hour per 300 LOC. Second, correctness proves the program design to be correct, but does not necessarily prove the program implementation to be correct [43].

Ada Tools: The Spark/Ada language was developed as a proprietary language for Praxis Critical Systems. Using Spark/Ada to specify and validate programs has been shown to reduce formal testing costs by 80% [3]. Spark is an annotated subset of Ada placed as comments within the

program. The Ada compiler ignores the annotations, but the Spark Examiner checks the annotations against the code, expediting early code analysis and fault avoidance. Spark/Ada has been a key element for building Praxis' high integrity systems. Peter Amey writes that using Spark/Ada for formal correctness techniques is pushing testing towards extinction [3].

Java Tools: The Java Modeling Language (JML) [30], [31], [9], [32], was developed in the late 1990s as a modeling language for incorporating formal specifications into Java programs. Modeling languages sit between the user (using natural language) and the coded program (written in the coding language) to specify program functions, behaviours and responsibilities and to validate program correctness. JML extends Meyers' design by contract to the Java language. Following the Spark/Ada paradigm, JML provides a language for specifying programs using design by contract. JML specifications can be verified by various tools, including ESC/Java2 [20], [13], [14] and LOOP [50], [49]. JML and its verification tools are well known techniques for specifying Java programs.

2.2.2. Non-Formal Methods

Non-formal methods are non-mathematical techniques to analyze a program's syntax and semantics for possible faults. The non-mathematical technique cannot prove fault absence and usually try to balance the following: soundness, completeness, scalability and usability [23]. Sound tools report no false negatives (i.e. “the ability to detect all possible errors of a certain class" [23]). Complete tools report no false positives (i.e. “reasonably precise" [23]). Scalable tools try to balance data flow methodologies with analysis soundness and completeness. Tool usability distinguishes how a tool works in practice.

Java Tools: FindBugs [5], [15] is a pattern-driven tool for examining Java byte code and source code. FindBugs was initially developed by a team at the University of Maryland led by William Pugh. The architecture relies on a input rule set file to locate instances of faults. Rule sets are bug patterns that “are likely to be errors” [15]. The tool has over 300 rule set patterns written in Java. Cole, et al. [15], emphasize that it finds one fault per 0.8 - 2.4 KLOC on non-commercial source code.

PMD looks for potential faults in Java source code. It takes a given rule set—written in XML—and pattern matches it with source code. PMD is basically a compiler front-end as it analyzes the input program abstract syntax tree and builds control flow graphs. PMD documentation explains that each rule traverses the abstract syntax tree to find potential errors. After analysis, violations the output can be printed in many formats.

Jlint [2], designed by Konstantin Knizhnik and Cyrille Artho, is another static analysis tool. The tool has two components: a syntax checker (AntiC) and a semantic verifier. Jlint's semantic verifier is designed to look for three categories of faults: synchronization, inheritance and data flow. The semantic checker examines data flow at two levels, locally and globally. A local data flow analysis examines a set of statements within a given scope where there is no control flow transfer [27]. A global data flow analysis, also known as intraprocedural analysis, examines method-inclusive statements [27]. At the two levels, with respect to data flow analysis, Jlint calculates ranges of values for expressions and local variables and saves the calculations into a variable descriptor. A handy feature of Jlint is the implementation of a history file, which can be stored to suppress warnings during subsequent analysis.

C/C++ Tools: Coverity is a path, flow and partially context-sensitive proprietary tool that can identify many critical faults in C, C++, C#, Java and other languages. The proprietary tool was created by a team at Stanford University led by Dawson Engler. There are over sixty individual C/C++ checkers included with the tool [18]. The documentation states that the checkers fall into

four primary categories: security, crash-causing, incorrect program behaviour, concurrency and performance degradation. The categories contain further groups of potential problems.

Fortify is a path, flow and partially context-sensitive proprietary tool geared predominately towards security. Its source code analyzer component uses fault pattern rulesets, which are “code idioms that are likely to be errors [15]." Currently, Fortify 360 examines source code for over 470 potential vulnerabilities in five main categories: data flow, control flow, semantic, structural and configuration. The partially context-sensitive tool uses global, interprocedural taint propagation data flow analysis to analyze flow between source and sink. In 2006, Fortify integrated FindBugs as a plugin for source code analysis.

Parfait is a proprietary analysis tool for C/C++ developed by Sun Microsystems and the University of Queensland, Brisbane [12]. The Parfait architecture is a tier structured hierarchy ordered from faster to slower analysis. Hence, the Parfait analysis tool can efficiently analyze 1+ MLOC. The framework is designed around a demand-driven Parfait tier paradigm. First, a worklist is created for each bug category. The worklist is initialized and populated with statements that could contain the faults for the given category. Second, in each tier, Parfait iterates over the worklist eliminating proven harmless statements. Finally, the remaining statements are reported as potential faults to be reviewed by the user. Parfait's design improves the traditional forward analysis, thus, improving traditional analysis bottlenecks.

The static analysis tool ARCHER (Array CHeckER) is strictly a C memory access checker [52] designed by a team at Stanford University. The tool can work either with or without annotations; however, the tool is “not a verifier [52]." ARCHER analyzes code through inferring a set of context-sensitive constraints. ARCHER's design includes using path-sensitive analysis and interprocedural symbolic analysis. It evaluates all known expressions at every array reference, pointer dereference or call to a function that expects a size parameter. Faults are either categorized as security concerns or standard faults. The three analysis phases include: (1) constructing abstract syntax trees using a modified version of the GNU C compiler, (2) transforming the abstract syntax into intermediate control flow graph (cfg) representation and constructing an approximate program call graph, and (3) analysing each function for potential memory errors. The tool advertises its false positive rate to be less than 35%.

2.2.3. Exploitation Testing

Exploitation testing is the next phase of a gray or black hat audit. Testing takes on many forms [26]. The goal of this phase is to demonstrate and document repeatable system compromise by a hostile entity. When live penetration tests are not mandatory, testing needs to occur off line in a safe environment [8]. Test cases need to be as simple as possible and can lead to the discovery of new or composite vulnerabilities. Due to cost and time limitations, testing exploitable code typically requires deciding on both prioritized vulnerabilities to examine and political motivations for fixing code by showing an exploit is possible.

2.2.4. Managing Fault Removal

Exploitable vulnerabilities need to be removed depending on system circumstances [22, 54]. Schmeelk et al. [47] reported a generalized prioritization schema based on fault: system impact, location, scope and design. Removing and eliminating faults based on a case-specific general schema will help reduce development costs and post-deployment patching.

3. C/C++ SECURITY CASE STUDY ANALYSES

3.1. Open Source Hardening Project Study

In early 2008, Coverity published results [17] from a contract with the U.S. Department of Homeland Security to participate in the Open Source Hardening Project designed to enhance security and quality of 250 open source software projects including Samba, FreeBSD, Linux, Perl, Python, PHP, TCL and NTP. One of the intentions of the project was to measure and compare the overall fault densities of the participating projects as determined by current static analysis tools [17].

3.2. Yahoo! Source Code Analysis Study

A study at Yahoo! during 2008 [47] found the most prevalent fault category to be null pointer dereferences, totalling 23.8% of all true faults found. Resource leaks totalled 18.3% of all true faults found. Trailing resource leaks, they found unsafe use of return value faults totalled 10.3% of all true faults. The fourth category of faults, related to concurrency concerns, totalled 7.7% of all true faults found. The fifth most prevalent fault category, unintentionally ignored expressions, totalled 4.7% of all true faults.

4. APACHE TRAFFIC SERVER SECURITY ANALYSIS

The Apache Traffic Server is an extensible HTTP/1.1 compliant caching proxy server developed originally by Inktomi, acquired by Yahoo! in 2002 and donated to Apache Software Foundation (ASF) in 2009 [53]. It handed over 400TB/day at Yahoo! as both a forward and a reverse server. Currently, it is a top-level project at the Apache Software Foundation (ASF).

A proxy server typically sits at the edge of a network and “acts as an intermediary for requests from clients seeking resources from other servers [45]." A forward proxy “takes origin connections from intranet clients and connects them to servers out on the internet [46]." Typically, a forward proxy is used for content filtering, email security, NATs and compliance reporting. A reverse proxy is an “Internet-facing proxy used as a front-end to control and protect access to a server on a private network [45]." Typically, a reverse proxy is used for load balancing, caching, SSL offloading, compression, content redirection, and as an application firewall, among other uses. A traffic proxy is designed to improve network efficiency and performance.

4.1. Manual Analysis

The first security auditing analysis phase consists of gathering relevant information. The following subsections explore the ATS in detail.

4.1.1. Background Information

The ATS has three main processes—server, manager and cop. Similarly to the Apache HTTP Web Server Project, the ATS proxy is constructed around a dynamic plug-in service architecture. The ATS provides six main API functions: HTTP header manipulation, HTTP transactions, Input/Output (IO), networking connecting, statistics and overall management. The ATS does contain a database to cache pages. As such, it can store/serve (but does not execute) Java applets, JavaScript program and VBScripts, and other executable objects from its cache. The ATS is designed to operate on Fedora, Ubuntu, FreeBSD, OpenSolaris and OSX operating systems.

4.1.2. Audit Purpose

The purpose of this research is to explore standardizing source code security auditing. We, therefore, have taken a white to gay hat approach. Black hat analyzes are highly dependent on a precise execution environment.

4.1.3. Quantitative and Qualitative Code Characteristics

There are both quantitative and qualitative code-base characteristics to consider. ATS consists of over 348,732 lines of code. The proxy is designed around a C plug-in architecture. As such, third-party plug-ins will need additional analysis. The proxy is developed in C and C++ and scans Hyper-Text Transfer Protocol (HTTP) and Multipurpose Internet Mail Extensions (MIME). The ATS does contain a database to cache pages; it can store objects from its cache. Annotations for the static analysis tool, Coverity, are present within the code. Warnings that have been analyzed have respective suppression annotations buried within the large code base to guide further analysis.

4.1.4. Attack Surfaces

The attack surfaces include a few sources. Code security also depends on the physical location of the proxy. As such, a more comprehensive audit need to be made when the proxy environment is determined. The code base consists of two public-facing libraries: input/output and caching. Vulnerabilities from these libraries need to receive more comprehensive analysis.

Security is considered within the ATS design. At a high-level, access, SSL, DNS configuration and Proxy Authentication are considered. Finer-grained security analysis needs to be examined for a more comprehensive audit.

4.2. Tool Analysis

The first pass of a code base is to gather relevant information. The following subsections explore the ATS in detail.

4.2.1. Coverity

Coverity located 266 potential vulnerabilities in the ATS show in Table 1. The analysis reported that the tool exceeded the path limit of 5000 paths in nearly 20% of the functions. Interestingly, ATS, prior to donation to Apache, had been analyzed as proprietary software by Coverity at Yahoo! As such, it already contained Coverity specific annotations to suppress unnecessary analysis output.

Many of the output vulnerabilities lay within test, configuration and backend files. The results labelled *PW* are parse warnings generated by the compiler. The results labelled *RW* are recovery errors generated by the parser from a parse error; in the case where a recovery was impossible the tool labels the problematic function as *ROUTINE NOT EMITTED* specifying it was not analyzed.

Table 1. ATS v 2.0.1 Coverity Analysis.

Quantity	Defects
8	NULL RETURNS
5	PARSE ERRORS
1	PW.ASSIGN WHERE COMPARE MEANT
2	PW.BAD RETURN VALUE TYPE
1	PW.CAST TO QUALIFIED TYPE
1	PW.CONVERSION TO POINTER ADDS BITS
6	PW.DECLARED BUT NOT REFERENCED
48	PW.INCLUDE RECURSION
9	PW.INCOMPATIBLE PARAM
62	PW.NON CONST PRINTF FORMAT STRING
7	PW.NO CORRESPONDING DELETE
4	PW.NO CORRESPONDING MEMBER DELETE
239	PW.SET BUT NOT USED
1	PW.USELESS TYPE QUALIFIER ON RETURN TYPE
5	RW.BAD INITIALIZER TYPE
1	RW.BAD RETURN VALUE TYPE
5	RW.EXPR NOT OBJECT POINTER
1	RW.ROUTINE NOT EMITTED
1	TAINTED SCALAR

4.2.2. Fortify

Fortify located 1,723 potential vulnerabilities in the ATS show in Table 2. Many vulnerabilities lay within non-exploitable files (backend and configuration files). Six flagged vulnerabilities (true and false positives) are discussed.

Table 2. ATS v 2.0.1 Fortify Analysis.

Categories	Rules
Data Flow	Privacy Violation, Integer Over Flow, Path Manipulation, System Information Leak, Setting Manipulation, System Information Leak, Setting Manipulation, String Termination Error, Resource Injection, Illegal Pointer Value
Control Flow	Null Dereference, Missing Check Against Null, Use After Free, Redundant Null Check, Insecure Temporary File, Uninitialized Variable, Double Free Memory Leak, Unreleased Resource Race Condition
Semantic	Insecure Randomness, Heap Inspection, Command Injection, Command Injection, Obsolete, Portability Flaw, Process Control, Format String, Weak, Cryptographic Hash, Insecure Compiler Optimization, Unchecked Return Value, Often Misused, Dangerous Function
Structural	Dead Code, Password Management, Code Correctness, Type Mismatch, Poor Style
Configuration	Dead Code, Password Management, Code Correctness, Type Mismatch, Poor Style
Buffer	Out-of-Bounds

Fortify Vulnerability One: Figure 1 shows potentially off-by-one buffer allocations, CWE-193, in *iocore/cluster/P ClusterInline.h* at line 301. (Note that similar reports were given for the same file on lines 102, 138, 203, 344, 387 and 466.) Fortify labels the vulnerabilities as a low risk categorized as semantic. As these vulnerabilities reside within the io-core libraries, they must be carefully examined. The memory allocation on line 301 is guaranteed to be bounded by the default length, via a guard, prior to buffer allocation. However, the report is a true positive since there is no explicit byte reserved for the terminating null for the msg buffer.

Figure 1. Off-by-One Vulnerability Report.

```
iocore/cluster/P_ClusterInline.h(301) : alloca()
---------------------------------------------------
// Allocate memory for message header
int flen = op_to_sizeof_fixedlen_msg(CACHE_REMOVE);
int len = host_len;
// Bound marshalled data
if ((flen + len) > DEFAULT_MAX_BUFFER_SIZE)
     goto err_exit;
char *msg = (char *) ALLOCA_DOUBLE(flen + len);
```

Fortify Vulnerability Two: Figure 2 shows potential buffer over flows, CWE-119, in *proxy/mgmt2/BaseRecords.cc* in *fpath* at line 130. This potential over flow cannot be exploited from external forces. Figure 3 shows a buffer over flow in proxy/hdrs/HdrToken.cc in *fpath* at line 130. Both reports are given high risk by Fortify from malicious dataflow through a Buffer Overflow via a Format String. Traditionally, exploitable buffer over flows, caused by a non-properly terminating/allocating a string, can cause the call stack to be overwritten. Then, as the function returns, control is transferred as directed by maliciously overwritten data.

Figure 2. Buffer Overflow Vulnerability Report.

```
proxy/mgmt2/BaseRecords.cc(130) :  ->snprintf(3)
   proxy/mgmt2/BaseRecords.cc(130) : <-  (system_local_state_dir)
   proxy/mgmt2/Main.cc(347) : <->snprintf(3->0)
      libinktomi++/ink_string.cc(57) : <->strncpy(1->0)
   proxy/sac.cc(104) : <->ink_strncpy(1->0)
   proxy/sac.cc(103) : <- get_ts_directory(0)
   proxy/logging/LogStandalone.cc(349) : <- fgets(0)
---------------------------------------------------
/* For now, we are using a dbm for record sharing */
snprintf(fpath, sizeof(fpath), "%s%s%s", system_local_state_dir,
    DIR_SEP, MGMT_DB_FILENAME);
```

Figure 3. Buffer Overflow Vulnerability Report.

```
proxy/hdrs/HdrToken.cc(474) :  ->snprintf(3)
   proxy/hdrs/URL.cc(101) :  ->hdrtoken_init(0)
   proxy/Main.cc(1398) :  ->url_init(0)
   proxy/Main.cc(1396) : <->snprintf(3->0)
      libinktomi++/ink_string.cc(57) : <->strncpy(1->0)
   proxy/Main.cc(424) : <->ink_strncpy(1->0)
   proxy/Main.cc(498) : <->snprintf(3->0)
      libinktomi++/ink_string.cc(57) : <->strncpy(1->0)
   proxy/sac.cc(104) : <->ink_strncpy(1->0)
   proxy/sac.cc(103) : <- get_ts_directory(0)
      libinktomi++/ink_string.cc(57) : <->strncpy(1->0)
   proxy/logging/LogStandalone.cc(346) : <->ink_strncpy(1->0)
   proxy/logging/LogStandalone.cc(345) : <=> (env_path)
   proxy/logging/LogStandalone.cc(345) : <- getenv(return)
-------------------------------------------------------------------
/* For now, we are using a dbm for record sharing */
snprintf(buf, sizeof(buf), "%s%shdrtoken.dat", path ? path : "",
      path ? DIR_SEP : "");
```

Fortify Vulnerability Three: Figure 4 shows Fortify flagging a potential path manipulation from improperly validating input before use within *proxy/mgmt2/LocalManager.cc* at line 1284. CWE-20. The tool tags the report with a high priority for malicious dataflow from Command Injection category. Careful inspection of the code, however, shows that manipulation is impossible as the variable is given an absolute (not relative) path. The report is a false positive. Traditionally, command injection can cause the attacker to gain control over the executed command and/or modify resources within the environment.

Fortify Vulnerability Four: Figure 5 shows a report for a potential string termination error, CWE-170, in *proxy/mgmt2/tools/SysAPI.cc* at line 193. Fortify labels the report as a medium risk. String termination errors are a form of buffer overflow vulnerabilities as the string may not be properly terminated.

Figure 4. Command Injection Vulnerability Report.

```
proxy/mgmt2/LocalManager.cc(1284) :  ->execv(0)
  proxy/mgmt2/Main.cc(1047) :  ->LocalManager::startProxy(this->abs_pxy_bny)
  proxy/mgmt2/Main.cc(701) : <=> (lmgmt)
  proxy/mgmt2/Main.cc(701) : <- new LocalManager(this-> abs_pxy_bny)
  proxy/mgmt2/LocalManager.cc(399) : <->snprintf(3->0)
      libinktomi++/ink_string.cc(57) : <->strncpy(1->0)
  proxy/sac.cc(104) : <->ink_strncpy(1->0)
  proxy/sac.cc(103) : <- get_ts_directory(0)
      libinktomi++/ink_string.cc(57) : <->strncpy(1->0)
    proxy/logging/LogStandalone.cc(346) : <->ink_strncpy(1->0)
    proxy/logging/LogStandalone.cc(345) : <=> (env_path)
    proxy/logging/LogStandalone.cc(345) : <- getenv(return)
-------------------------------------------------------------------
const size_t absolute_proxy_binary_size  = sizeof(char)*(strlen(system_root_dir)
      + strlen(bin_path) + strlen(proxy_binary))+2;
absolute_proxy_binary = (char *) xmalloc(absolute_proxy_binary_size);
snprintf(absolute_proxy_binary, absolute_proxy_binary_size, "%s%s%s",
       bin_path,  DIR_SEP,  proxy_binary);
res = execv(absolute_proxy_binary, options);
```

Figure 5. String Termination Vulnerability Report.

```
proxy/mgmt2/tools/SysAPI.cc(130) :  ->strcasecmp(0)
   proxy/mgmt2/tools/SysAPI.cc(129) : <- Net_GetNIC_Protocol(1)
      libinktomi++/ink_string.cc(57) : <->strncpy(1->0)
   proxy/mgmt2/tools/SysAPI.cc(437) : <->ink_strncpy(1->0)
   proxy/mgmt2/tools/SysAPI.cc(435) : <- find_value(2)
      libinktomi++/ink_string.cc(57) : <->strncpy(1->0)
   proxy/mgmt2/tools/SysAPI.cc(849) : <->ink_strncpy(1->0)
   proxy/mgmt2/tools/SysAPI.cc(829) : <=> (pos)
   proxy/mgmt2/tools/SysAPI.cc(824) : <=> (pos)
   proxy/mgmt2/tools/SysAPI.cc(824) : <->strstr(0->return)
   proxy/mgmt2/tools/SysAPI.cc(812) : <- fgets(0)
```

Fortify Vulnerability Five: Figure 6 shows a report for a potential user password privacy violation, CWE-359, in *proxy/hdrs/HdrTest.cc* at line 1180. The report is labelled by Fortify as a medium risk of privacy violation via dataflow. These are false positives, as they are passwords passed as headers within a message, not private user passwords. Traditionally, this type of privacy vulnerability occurs when private user data enters a program or written to an external location in plaintext or using trivial encoding mechanisms.

Fortify Vulnerability Six: Figure 7 shows a report of a potentially vulnerable dynamic class loading, CWE-545, in *proxy/Plugin.cc* at line 89. The vulnerability is reported as medium risk with process control via semantic usage; however, it currently is important for proxy plug-in design. It is thus a false positive. A white list of potential benign functions could be incorporated into future proxy architectures to assure sanitized plugins. Typically, an attack could be hidden within a dynamic class static initializer (if any) or other class components. This form of attack could be loaded dynamically while application is running.

Figure 6. Privacy Violation Vulnerability Report.

```
proxy/hdrs/HdrTest.cc(1180) :  ->printf(2)
   proxy/hdrs/HdrTest.cc(1156) : <- HTTPHdr::print(0)
   proxy/hdrs/HTTP.h(808) : <- http_hdr_print(2)
   proxy/hdrs/HTTP.cc(482) : <- url_print(1)
      proxy/hdrs/MIME.cc(3068) : <->memcpy(1->0)
   proxy/hdrs/URL.cc(1669) : <->mime_mem_print(0->2)
   proxy/hdrs/URL.cc(1669) : <-  (url->m_ptr_password)
```

Figure 7. Process Control Vulnerability Report.

```
proxy/Plugin.cc(89) : dlopen()
static void *
dll_open(char *fn, bool global)
{
  int global_flags = global ? RTLD_GLOBAL : 0;
  return (void *) dlopen(fn, RTLD_NOW | global_flags);
}
```

5. RELATED LITERATURE

In 2009, Walden et al. [51] examined fourteen popular open source PHP web applications developed from 2006 to 2008. Of the fourteen applications, the vulnerability density of the final product was approximately 3.3 KLOC. Of the found security vulnerabilities, Walden et al. showed that Cross-Site Scripting and SQL Injections were the most frequent type. Additionally, they found continuous static analysis of post-development software counter-productive for measuring vulnerabilities in each 100 week development samples; instead, they examined both the first and last development sample with a false-positive rate of approximately 18%. The researchers noted that the project development language impacts both the code size and vulnerability type. The research then presented a security resources indicator for the projects to measure the importance of security within a project. Their indicator was based on application-security configuring/installing documentation, dedicated security email alias, list of relevant vulnerabilities and secure development-process documentation.

In 2006, Chandra, Chess and Steven [10] described useful techniques for integrating static analysis tools into source code analysis. They brought forth the need for adopting new vulnerability-detection tools. They also discussed how to integrate the tool; specifically who uses the tool, when to use the tool and what to do with analysis results. The paper set initial guidelines into integrating static analysis tools into the development process.

In 2004, Heffley and Meunier [23] explored using auditing software for software security evaluation. Their work was motivated by development fault repetition and Mitre's growing CVE database. The research was specifically designed to evaluate multiple static analysis auditors. The research found that different auditors located different vulnerabilities and noted the tool limitations. They stated future beneficial tool enhancements.

6. CONCLUSIONS

This paper addresses a research need to assemble a broad literature array to help standardize source code security audits. It then uses components of the assembled ideas to perform a white-to-gray hat source code security analysis on the Apache Traffic Server (ATS) to exemplify the process. The analysis located a lower-bound of potentially exploitable vulnerabilities and connected the vulnerabilities with the common taxonomy, the CWE. Finally, it discussed pertinent exploitation ramifications. As the ATS can function at a high traffic rate, both as a forward and reverse proxy, any potential vulnerability compromising the CIA model (Confidentiality, Integrity and Availability) needs to be removed—perhaps, even without a proof of concept.

REFERENCES

[1] IEEE Guide to Classification for Software Anomalies. IEEE Standard 1044.1-1995, Aug 1996.

[2] Jlint - Find Bugs in Java Programs, Online. http://jlint.sourceforge.net/ 2010.

[3] Peter Amey. Correctness By Construction: Better Can also be cheaper. CrossTalk Magazine, The Journal of Defense Software Engineering. March 2002.

[4] Apache Traffic Server, Online. http://trafficserver.apache.org/ 2011.

[5] Nathaniel Ayewah, William Pugh, J. David Morgenthaler, John Penix, and YuQian Zhou. Evaluating Static Analysis Defect Warnings on Production Software. In PASTE '07: Proceedings

of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering, pages 1-8, New York, NY, USA, 2007.

[6] Boris Beizer. Software Testing Techniques. John Wiley & Sons, Inc., New York, NY, USA, 1990.

[7] Boris Beizer. Software testing techniques (2nd ed.). Van Nostrand Reinhold Co., New York, NY, USA, 1990.

[8] Matt Bishop. Introduction to Computer Security. Addison-Wesley, 1st edition, 2009.

[9] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An Overview of JML Tools and Applications. Int. J. Software Tools Technical Transformations, 7(3):212-232, 2005.

[10] P. Chandra, B. Chess, and J. Steven. Putting the Tools to Work: How to Succeed with Source Code Analysis. Security Privacy, IEEE, 4(3):80-83, May-June 2006.

[11] B. Chess and G. McGraw. Static Analysis for Security. Security and Privacy, IEEE, 2(6):76-79, Nov-Dec. 2004.

[12] Cristina Cifuentes and Bernhard Scholz. Parfait: Designing a Scalable Bug Checker. In ACM SAW '08: Proceedings of the 2008 workshop on Static analysis, pages 4-11, New York, NY, USA, 2008.

[13] D. R. Cok and J. Kiniry. ESC/Java2: Uniting ESC/Java and JML. Technical Report. University of Nijmegen. 2004.

[14] D. R. Cok and J. Kiniry. ESC/Java2: Uniting ESC/Java and JML. Lecture Notes in Computer Science, 3362: 108-128. 2005.

[15] Brian Cole, Daniel Hakim, David Hovemeyer, Reuven Lazarus, William Pugh, and Kristin Stephens. Improving Your Software using Static Analysis to Find Bugs. In OOPSLA '06: Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications, pages 673-674, New York, NY, USA, 2006.

[16] Coverity. The Next Generation of Static Analysis, 2007.

[17] Coverity. White paper: Open Source Report, 2008.

[18] Coverity Version 5.4 Reference, 2011.

[19] Matthew B. Dwyer, John Hatcliiff Robby Robby, Corina S. Pasareanu, and Willem Visser. Formal Software Analysis Emerging trends in Software Model Checking. In FOSE '07: 2007 Future of Software Engineering, pages 120-136, IEEE Computer Society. Washington, DC, USA, 2007.

[20] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended Static Checking for Java. In PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation, pages 234-245, New York, NY, USA, 2002.

[21] B. Hackett, Y. Xie, M. Naik and A. Aiken. Soundness and its Role in Bug Detection Systems. In Proceedings of the Workshop on the Evaluation of Software Defect Detection Tools, New York, NY, USA, 2005. ACM.

[22] Allen Harper, Jonathan Ness, Gideon Lenkey, Shon Harris, Chris Eagle, and Terron Williams. Gray Hat Hacking: The Ethical Hacker's Handbook (Third Edition). Mc Graw Hill, New York, NY, USA, 2011.

[23] J. Heffley and P. Meunier. Can Source Code Auditing Software Identify Common Vulnerabilities and be used to Evaluate Software Security? In System Sciences, 2004. Proceedings of the 37th Annual Hawaii International Conference on, page 10 pp., Jan. 2004.

[24] Changbok Jang, Jeongseok Kim, Hyokyung Jang, Sundo Park, Bokman Jang, Bonghoi Kim, and Euiin Choi. Rule-based Auditing System for Software Security Assurance. In Ubiquitous and Future Networks, 2009. ICUFN 2009. First International Conference on, pages 198-202, June 2009.

[25] Yiannis Kanellopoulos, Panagiotis Antonellis, Dimitris Antoniou, Christos Makris, Evangelos Theodoridis, Christos Tjortjis and Nikos Tsirakis. Code Quality Evaluation Methodology Using The ISO/IEC 9126 Standard. International Journal of Software Engineering & Applications (IJSEA), Vol.1, No.3, July 2010.

[26] Mohd. Ehmer Khan. Different Approaches To Black box Testing Technique For Finding Errors. International Journal of Software Engineering & Applications (IJSEA), Vol.2, No.4, October 2011.

[27] Uday Khedker, Amitabha Sanyal, and Bageshri Karkare. Data Flow Analysis: Theory and Practice. CRC Press, Inc., Boca Raton, FL, USA, 2009.

[28] K. Kratkiewicz. Evaluating Static Analysis Tools for Detecting Buffer Over flows in C Code. In Master's Thesis, Harvard University, Cambridge, MA, 2005.

[29] Benjamin A. Kuperman, Carla E. Brodley, Hilmi Ozdoganoglu, T. N. Vijaykumar, and Ankit Jalote. Detection and Prevention of Stack Buffer Over flow Attacks. Communications of the ACM, 48:50-56,November 2005.

[30] G. Leavens and Y. Cheon. Design by Contract with JML: An On-line Tutorial. 2004.

[31] Gary T. Leavens. Tutorial on jml, the java modeling language. In ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering, pages 573-573, New York, NY, USA, 2007.

[32] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of jml: a behavioural interface specification language for java. SIGSOFT Softw. Eng. Notes, 31(3):1-38, 2006.

[33] Moohun Lee, Sunghoon Cho, Hyokyung Chang, Junghee Jo, Hoiyoung Jung, and Euiin Choi. Auditing system using rule based reasoning in ubiquitous computing. In Computational Sciences and Its Applications, 2008. ICCSA '08. International Conference on, pages 261-266, 30 2008-july 3 2008.

[34] Felix "FX" Lindner. Software security is software reliability. Communications of the ACM, 49:57-61, June 2006.

[35] Barbara Liskov. Technical perspective safeguarding online information against failures and attacks. Communications of the ACM, 51(11):85-85, 2008.

[36] T.J. McCabe. A complexity measure. Software Engineering, IEEE Transactions on, SE-2(4):308--320, Dec. 1976.

[37] B. Meyer. Eiffel software, 2007.

[38] Mitre. Common Weakness Enumeration (CWE), Online. http://cwe.mitre.org/ 2008.

[39] Mitre. Common Vulnerabilities and Exposures (CVE). Online. http://cve.mitre.org/ 2011.

[40] NIST. Software Assurance Metrics and Tool Evaluation (SAMATE). Online. http://samate.nist.gov/Main_Page.html 2011.

[41] NIST. Common Weakness Enumeration (CWE). Online. http://nvd.nist.gov/cwe.cfm 2008.

[42] Anthon Pang. What is a bug? 2002.

[43] Shari Lawrence Pfleeger and Joanne Atlee. Software Engineering: Theory and Practice (3rd Edition). Prentice Hall PTR, Upper Saddle River, NJ, USA, 2006.

[44] R. Satya Prasad, O. Naga Raju and R. R. L Kantam. SRGM with Imperfect Debugging by Genetic Algorithms. International Journal of Software Engineering & Applications (IJSEA), Vol.1, No.2, April 2010.

[45] Proxy Server. Online. http://en.wikipedia.org/wiki/Proxy_server 2010.

[46] Reverse vs. Forward Proxy Server, 2010.

[47] S. Schmeelk, B. Mills, and R. Noonan. Managing Post-development Fault Removal. In Information Technology: New Generations, 2009. ITNG '09. Sixth International Conference on, pages 205-210, April 2009.

[48] G. Michael Schneider, Johnny Martin, and W. T. Tsai. An Experimental Study of Fault Detection in User Requirements Documents. ACM Trans. Software. Engineering Methodologies, 1(2):188-204, 1992.

[49] Van den Berg and B. Jacobs. Tools and Algorithms for the Construction and Analysis of Systems. 2001.

[50] Joachim van den Berg and Bart Jacobs. The LOOP Compiler for Java and JML. In TACAS 2001: Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, pages 299-312, Springer-Verlag. London, UK, 2001.

[51] J. Walden, M. Doyle, G.A. Welch, and M. Whelan. Security of Open Source Web Applications. In Empirical Software Engineering and Measurement, 2009. ESEM 2009. 3rd International Symposium on, pages 545-553, Oct. 2009.

[52] Yichen Xie, Andy Chou, and Dawson Engler. Archer: using Symbolic, Path-sensitive Analysis to Detect Memory Access Errors. In ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering, pages 327-336, New York, NY, USA, 2003.

[53] Traffic Server, Online. http://gigaom.com/2009/11/06/10-top-open-source-resources-for-cloud-computing/ 2009.

[54] Ghazia Zaineb and Irfan Anjum Manarvi. Identification And Analysis Of Causes For Software Bug Rejection With Their Impact Over Testing Efficiency. International Journal of Software Engineering & Applications (IJSEA), Vol.2, No.4, October 2011.

[55] J. Zheng, L. Williams, N. Nagappan, W. Snipes, J.P. Hudepohl and M.A. Vouk. On the Value of Static Analysis for Fault Detection in Software. Software Engineering, IEEE Transactions on, 32(4):240-253, April 2006.

[56] M. Zitser. Securing Software: An Evaluation of Static Source code Analyzers. Online. http://hdl.handle.net/1721.1/18025 Massachusetts Institute of Technology. 2003.