

Can Source Code Auditing Software Identify Common Vulnerabilities and Be Used to Evaluate Software Security?

Jon Heffley and Pascal Meunier
 Purdue University CERIAS
 pmeunier@purdue.edu

Abstract

Software vulnerabilities are a growing problem (c.f. MITRE's CVE, <http://cve.mitre.org>). Moreover, many of the mistakes leading to vulnerabilities are repeated often. Source code auditing tools could be a great help in identifying common mistakes, or in evaluating the security of software. We investigated the effectiveness of the auditing tools we could access, using the following criteria: number of false positives, false negatives by comparison to known vulnerabilities, and time required to validate the warnings related to vulnerabilities. Some of the known vulnerabilities could not be found by any code auditor, because they were fairly unusual or involved knowledge not contained or codified in the source code. The coding problems that could be identified consisted of string format vulnerabilities, buffer overflows, race conditions, memory leaks, and symlink attacks. However, we found it extremely time-consuming to validate warnings related to the latter four types, because the number of false positives was very high, and because it was not easily apparent if they were real vulnerabilities. These required that the code be audited locally, by people familiar with the code, and carefully inspected to see if the values could be manipulated in such a way as to produce malicious effects. However, the string format vulnerabilities were much easier to recognize. In small and medium scale projects, the open source program Pscan was useful in finding a mix of coding style issues that could potentially enable string format vulnerabilities, as well as actual vulnerabilities. The limitations of Pscan were more obvious in large scale projects like OpenBSD, as more false positives occurred. Clearly, auditing source code for all vulnerabilities remains a time-consuming process, even with the help of the current tools, and more research is needed in identifying and avoiding other common mistakes.

1. Introduction

The number of known vulnerabilities reported every year increases. According to NIST's ICAT, as of September 2003, the number of vulnerabilities catalogued from 1998 to 2003 has been 246, 860, 992, 1506, 1307, 798 where the count for 2003 is partial [26]. There are so many vulnerabilities, with incomplete information, that the Common Vulnerabilities and Exposures effort by MITRE, on which ICAT relies, has a processing queue such that lower priority vulnerabilities can take several months or more to get processed.

Vulnerabilities are costly. The CSI security survey reported losses of \$378 million, \$456 million and \$202 million respectively in 2001, 2002 and 2003, from 196, 223 and 252 respondents that would quantify their losses [25]. Only about 36% of vulnerabilities are due to configuration or design problems; the rest are due to programming errors [26]. The exploitation of vulnerabilities can be made less likely by shielding the vulnerable software, e.g., using firewalls (which will not be discussed in this paper); vulnerabilities can be corrected by applying patches; and some can be prevented with better software development.

The remediation of vulnerabilities by patching is expensive, each costing vendors on the order of \$100,000 (personal communication). The management of patches by customers is not only expensive but is also a complex or risky proposition, especially for mission-critical systems. Insufficient testing may result in unexpected behaviors, whereas cautiousness can increase the window of vulnerability. NIST issued a lengthy recommendation addressing this problem [27]. However, the avoidance of all these problems seems more attractive.

The irritating factor in this situation is that the same mistakes keep being repeated by programmers, and fall within well understood broad categories. Using keyword searches on the description of CVE entries, from 2000 to 2002, vulnerabilities related to buffer

overflows, directory traversal attacks, format string vulnerabilities, symlink attacks, cross-site scripting vulnerabilities and shell metacharacter issues represented respectively 20%, 11%, 9%, 4%, 4%, and 3% of all vulnerabilities. This means that 51 out of the 64% of the vulnerabilities due to programming errors, are repeated basic mistakes. In this work, we investigated the effectiveness of software auditing tools in catching common mistakes.

Many tools are available to help improve program quality. Most are concerned with coding standards enforcement, statistical quality control and computing indicators of code size, complexity and density (e.g., Lint, Cleanscape). Whereas those are useful in catching functional bugs and improving consistency and ease of maintenance, we focused on tools claiming security applications or benefits. Our stance is from a third-party perspective, i.e., external to the development process. Such a perspective could apply to people who want to help improve open source projects or who want to evaluate the quality and security of a given software package. We were unable to find a comparative study of the usefulness of such tools for that purpose; we only found self-reporting publications (i.e., publications on the tools by the authors of the tools themselves).

2. Review of typical programming errors

Buffer overflows are the most common vulnerability, making up over 60% of CERT advisories [8]. Heap and buffer overflows occur either when data is written past a point that it is intended, or when it is read past a point it is supposed to be read. Both of these are often the result of incorrect information reading or writing, or just incorrect implementation of buffers. For example, a common mistake occurs in processing loops for buffers with memory allocated for n elements or objects. The loop is run such that the index's range is 0 through n . The n^{th} element or object is accessed with the index $n-1$, so when the index reaches n , it is actually accessing a non-existent position. In this case, writing a value at index n overwrites memory following the buffer. This could cause a segmentation fault, or other unexpected behavior. This is known as an "off-by-one error," although there are unlimited possibilities for how off a pointer can be. If the index depends on arithmetic that is not properly checked, the program may go awry.

Another common buffer overflow vulnerability occurs with the string operations in the standard C library. A string is just an array of characters that ends in a null character, represented as `'\0'`. It is placed on the end of string literals, strings that begin and end

with double quotes, automatically. This character tells functions that read strings, such as `strcpy()`, `strlen()` and `printf()`, that the string is over. If the null character is missing, these functions may attempt to access or write to unallocated memory, or memory in use for other purposes. Both can cause unexpected results. This can occur because some string functions may return invalid strings; for example, under certain conditions `strcpy()` may return a string that is not null terminated.

Arithmetic errors in the calculation of the buffer size needed for a string can also produce invalid strings. For example, say we wanted to create a buffer to copy one string, X , into the new one we created, Y . The first thing we might do would be to create the buffer Y . We would want to know the size of X , which we would determine by calling `strlen()` on X . The mistake comes about when we create the buffer Y to be the size returned from the `strlen()` call; we did not take into account the null character. In this case, calling `strcpy()` to copy X to Y would result in a buffer overflow when the null character in X is written past the limit of the Y buffer. Let's say that this did not immediately cause an error; then the null character may get overwritten when that memory is used for other purposes.

Format string vulnerabilities are very dangerous because they essentially allow users to modify the program and how it is executed. Format strings are specifications such that the program understands how to represent information, and are used in functions such as `printf()`, `sprintf()` and `syslog()`. They are strings containing special formatting characters starting with `'%'`, which represent arguments to be processed. They are often used with a variable number of arguments. Because a regular string that is used as a format string is passed directly to the output, the most common misuse is passing a data string as a format string, without arguments. Vulnerabilities result if the data string is under the control of an attacker, because the attacker can then make the program process arbitrary or chosen information. They are easy to fix.

Code injection vulnerabilities result from the mixing of code and data. This typically happens when attempting to dynamically generate commands, such as SQL to a database engine, with user input. The object of code injection is tricking the program into executing code through clever input construction [3]. It can work through a command separator (e.g., `';`), or it can take advantage of buffer overflows and format string vulnerabilities as described above. Careful input validation is needed to protect a program from code injection, and this becomes a very difficult problem to detect. Code injection vulnerabilities are also frequent with function calls like `system()` or `popen()`, which interpret and execute arguments to the function calls.

These arguments are often generated dynamically by including some data that may have been compromised with malicious code.

A symbolic link (symlink) is a link that points to another file, but acts as if it were that file. Symlink vulnerabilities result from the ambiguity as to which file is affected. They can be exploited by making a program affect a file it did not expect to modify [3], or access a file and use data under the control of the attacker. Symlink attacks are often coupled with timing (race conditions) attacks, to fool a program into operating on another file. This is most often a problem when a user can make a symbolic link to a file that he may not have access to, but the program he is trying to trick does.

Some vulnerabilities may occur without any malicious or unexpected input, but because of the way the program is set up. One particular kind of vulnerability of this nature is a race condition. A race condition is “anomalous behavior due to unexpected critical dependence on the relative timing of events” [4]. This tends to occur when more than one process uses a single variable, file, or other resource. The resource is modified by one process while presumed to still be what it was last time by another process. Through certain sequences of events, a vulnerability may present itself. A user does not need to have malicious intent to cause such a problem, and the issue may be detrimental to both the user using the program, and to the organization providing the service. Race conditions can be fixed with multiple exclusion locking, or by creating a daemon that has exclusive access to the resource and shares it through interprocess communication [4]. However, it is finding the occurrence of race conditions that proves the most difficult.

Vulnerabilities may also occur in the way a program interacts with other programs; in this case the knowledge to understand the vulnerability is external to the program being examined, and source code auditing programs cannot help. For example, there was a vulnerability in the Wu-ftpd program (CVE-1999-0997) that occurred in its conversion service that “pipes a requested file through a program” [6]. In this particular case, some parameters passed to the conversion program could be used to access unforeseen functionality, in this case to execute yet another program with the privileges of the original program.

3. Code Analysis Technologies

Code auditing software can be static or dynamic; most auditing software uses static analysis. Static analysis “aims at determining properties of programs by inspecting their code, without executing them” [1]. The supposed advantage to this is that it can detect errors that can be very difficult to find through testing [2]. Code analyzers (a.k.a. auditing softwares) range from “function spotters”, hardly more sophisticated than grep, to rule-based analyzers [24] and pre-compilers. New approaches involve detecting software security vulnerabilities using constraint optimization [22], which extracts the security implications from the code.

Dynamic analysis aims at finding flaws during the execution of the program. This may require exhaustively exercising all the execution paths in a program, with all possible kinds of inputs, which can be rather complicated and time consuming. In its simplest form, random input can be used, such as that generated by the “fuzz” testing program [28] or ISIC [29]. In large or complex systems, however, the cause of a given malfunction can be difficult to pinpoint even if the random input is replayed. Using binary search to isolate the input responsible for the malfunction ignores accumulated state in the system and can result in contradictory results, such as the “critical input” (which may not exist independently) being in a set of inputs but being absent from both halves of the set (!). More sophisticated approaches involve creating a grammar describing inputs, and testing the running program with various inputs designed to find flaws; this technique has proved very powerful, but requires significant investment and deep understanding of the program’s function to bear fruit [23]. It also does not indicate exactly where in the code the problem originates.

The choice of auditing resources came from a compiled list of code auditing tools provided by the Sardonix website [5]. Its goal is “to facilitate program audits and provide a core repository for reviewing and submitting them” [5]. Most of the tools were available for free, with the exception of one. FlexeLint is a commercial product from Gimpel Software (who let us use it for free during an evaluation period). All of these code auditing applications ran on Unix/Linux platforms, and were able to analyze C source code. As such, they represent the tools that auditors have found to be most helpful, even though we are aware that there exist other tools (such as [22]). We chose to run our experiments on Red Hat Linux 8.0, and focused on C in UNIX, due to the popularity of the “C” family of languages, the ready availability of popular open

source programs of various sizes coded in C for UNIX, and the well-known vulnerabilities that are less frequent or not present in other languages (such as buffer overflows and string format vulnerabilities).

Cqual is an open source code auditor developed by UC Berkeley that is still in beta version. It is “based on the C front-end used in David Gay's Region Compiler, which is in turn based on the GNU C Compiler” [9]. This particular auditor binds with Emacs, to allow programmers to audit their source code while making changes to it. It allows for programmers to add type qualifier annotations within the C program (in the form of comments) to check to make sure the annotations are correct.

ITS4 (“It's The Software, Stupid – Security Scanner”), copyrighted by Cigital, Inc., is self-described as “a tool for statically scanning security-critical C and C++ source code for vulnerabilities” [7]. It looks for risky function calls, and determines, on five point scale (LOW_RISK, MODERATE_RISK, RISKY, VERY_RISKY, MOST_RISKY) how risky it is. In addition, it offers a description of the potential problem, and how one might go about fixing it. It is available for free to non-commercial purposes. It allows auditors to add comments to help ignore some lines (to reduce the output), add functions to be watched for, and add ways for functions it looks for to be handled. Potential vulnerabilities can also be ignored by including a file that lists previously found problems that should not be listed again.

Two open source programs were designed to mimic ITS4. Flawfinder was made by David A. Wheeler. RATS (“Rough Auditing Tool for Security”) was developed by Secure Software Inc. The author of Flawfinder and Secure Software Inc worked on their projects simultaneously, and did not realize until just prior to release that they were working on the same thing. They have agreed to mention one another's project, and to combine their efforts in the future [11]. Both use a similar five-point scale to rank the possible vulnerabilities they identify, and give programmers and auditors the ability to ignore certain lines by adding comments into the code or using a list of previously found potential problems.

Splint (Secure Programming Lint) was developed at the University of Virginia, and is built upon the concept of lint. They describe it as “a tool for statically checking C programs for security vulnerabilities and coding mistakes” [10]. It is well documented, and the site features many papers describing how Splint finds potential errors in the source code. Like Cqual, it allows programmers to annotate the code to effectively find more poor code than if used without it.

FlexeLint, the commercial software, is described as a tool to “check your C/C++ source code and find bugs, glitches, inconsistencies, non-portable constructs, redundant code... [and look] across multiple modules” [12]. It is essentially an advanced version of the program lint. It is not available to anyone, commercial or non-commercial, for free (unless for an evaluation period).

Pscan is a program by Alan Dekok that is also open source. It is designed to detect only format string vulnerabilities, but to do so effectively. It scans C/C++ files to find the greatest potential for format string vulnerabilities. It is a very simple program that is only around 100 lines long. Pscan is complemented with a program to find variable argument functions defined in the program, and helps them to be included into the search to find potential format string vulnerabilities.

4. Methods

The focus of our research was to evaluate the usefulness of the auditing applications to increase the security of the code. Within the context of the security flaws and programming errors that they were supposed to find, elements of their usefulness were a) setup time and ease of use, from the point of view of a third party (not the original developers); b) time required to make use of the output; and c) consideration of the number of false positives and false negatives. Their usefulness was mostly a consequence of how they found their respective possible bugs, which will be discussed at length later for each particular code auditor. Due to practical limits on this study, we abandoned the evaluation of some code auditing applications whenever points a) and b) were unreasonable.

We chose open source software as a benchmark, based on the number and variety of known vulnerabilities. Wu-ftpd provided 18 documented vulnerabilities in a single package, providing real examples of race conditions, code injection, memory leaks, string format vulnerabilities, and many buffer overflows. So, the code auditing applications were first run on the source of an older version of Wu-ftpd (v2.6.0). The goal was, without knowing where specifically the security vulnerabilities occurred, to use the output of the code auditing applications to try and rediscover the documented vulnerabilities, while evaluating the ease of use, setup time, and time required to make use of the output. The next steps were based on the information gathered from the first step to further analyze strengths and weaknesses. This involved creating test programs to disambiguate some results. The plan was then to benchmark the usable code auditing applications against CVE (Common

Vulnerabilities and Exposures) entries of the appropriate vulnerability types in open source software, to quantify the number of false positives and false negatives that occur in practice.

The time constraint for the evaluation was one semester, so software auditors that proved problematic, or more work than they seemed worth, were not considered past the first step of the evaluation process, and this made our evaluation more ad-hoc than we would have wished. Whereas some code auditing software might still have its benefits, especially for developers willing to put in the required flags and extra work, for the purpose of this research, they did not meet the qualifications for providing an efficient and useful method for evaluating software or auditing open source code by a third party.

5. Results and Discussion

Three auditing applications didn't progress past the first stage: Cqual, Splint, and FlexeLint. Cqual reported errors concerning pre-processor definitions that were legal. This inconvenience was compounded by the fact that it required the Emacs text-editing program. While this may serve as an advantage to auditing your own source code if you are writing it with Emacs, this does not help to serve a third party (such as ourselves) well to auditing code and compiling information, and of course doesn't work for people who do not use Emacs. Taking full advantage of Cqual required custom annotations, which require time and knowledge that a third-party code editor does not have. We found ourselves in the position of essentially having to perform a full-fledged code audit prior to using the application, which defeated the point of using an automated program to find vulnerabilities.

Splint also offered the programmer the ability to include annotation to help the auditor to understand the intent of the code. However, in the absence of annotations by the programmer, we found ourselves in the same situation as with Cqual. Moreover, Splint was complicated to use due to its desire to look through header files, including those of the C library. The exact locations of the header files were difficult to provide. The amount of work required to get Splint to function with software packages was surprisingly high; it required knowing exactly where all of the files were in the system.

Splint provided the option not to include header files. However, that limited its power; the header files for the C library help the auditor determine the problems particular to a system, without having to code system specific cases in the auditor. Despite the fact that you could opt to not include system header

files, in the end using Splint proved too time-consuming. Whereas the program was well documented, the documentation concerned mostly the annotation, and ways of auditing it that appealed more to developers.

FlexeLint had similar problems concerning header files, most likely for similar reasons. Moreover, it would audit those files in addition to the files of the software package, which is not what we wanted. Library auditing aside, FlexeLint's output did not focus on security. The majority of the results it gave concerned types, and making sure that particular variables were not being used in places that variables of different types were needed. While this strong type checking may effectively help to produce more solid code, security was, for the most part, ignored; it also fell short in FlexeLint's priority system. The priority levels were "Error," "Warning," and "Information." In test code, it reported as a warning a recursive header file, which is benign and often avoided through preprocessing. However, it reported the fact that a string did not terminate with a null character as "information", the lowest priority. This could lead to a buffer overflow, and cause a segmentation fault. So, whereas FlexeLint could be helpful to the developers, its poor reporting of security issues made it uninteresting.

Flawfinder, RATS and ITS4 were little more than "function spotters." These auditing applications go through source code and find function calls that are infamous for causing security problems (e.g., `printf()`, `open()`, `crypt()`) [7] as well as declarations for static sized buffers (which are often misused). These auditing applications would also give the line number of the function call, a level of severity (if a bug), and propose how one might modify them. They can be tuned to only report possible vulnerabilities that meet a certain severity, as well as take in a list of vulnerabilities to ignore. These auditing applications can also use annotation within the code to ignore certain lines.

ITS4 was created because the creators wanted to "immediately produce a practical, widely applicable tool that developers can use." They did not create a complete parser that understood the code, due to the difficulties of making one for C/C++ code, as well as their future goals to support real-time interactive programming environments like Emacs and Microsoft Visual C++ [7]. RATS and FlawFinder came about because their creators wanted to make a truly open source code auditor that acted like ITS4 [11].

Even though these auditing applications focused on security, it was very difficult to find actual vulnerabilities. Despite Flawfinder's theoretical capability to avoid some false positives, in practice

they both flagged each and every printf() call, regardless of whether a vulnerability was actually present or not. The analysis is described in the ITS4 documentation:

“When ITS4 first flags a function name [defined in the vulnerability database], it looks up a “handler” for the function in the vulnerability database. The handler is responsible for reporting the problem flagged by the scanner. If no handler is found in the database, the default handler is used, which merely adds the problem to the results database. However, handlers can be used to perform more sophisticated analysis on a program” [7].

This produced an unwieldy number of false positives, and the sheer number of alerts to examine and verify was discouraging. Only about half of these alerts could be disproved by casual inspection of the code. The others each required about an hour or more to validate the arguments and trace the execution paths to where the values of the arguments would be determined, and the context of the function call. We found ourselves in agreement with ITS4’s documentation which claims (or disclaims), “[ITS4] only eliminates from one quarter to one third of the time it takes to perform the manual analysis” [7]. Therefore, 67% to 75% of the work of a full code audit is still required. We would like to point out that this amount of work by the software developers translates into much more work for third party auditors, who are not as familiar with the software.

A security concern that none of these applications addressed was tracing possible vulnerabilities through custom function calls. We frequently see custom functions that enable vulnerabilities if improperly called. It is difficult to determine whether the inputs for all calls to a function have been validated in previous code or have been generated in a safe manner. Whereas this is a regular exercise in code reviews, perhaps some new code auditing software could try to identify risky custom functions and flag their uses (as an option, because it runs to risk of multiplying the number of false positives). Although ITS4 allows suspect function names to be added manually to a file (along with specific handlers), this is similar to annotation in that it is more practical for the developers, and not for third parties or for evaluation. The shell tool developed for Pscan (see below) hints at reasonable ways of approaching this problem. We conclude that as security evaluation tools or for third parties wanting to audit source code, ITS4, RATS and FlawFinder are only marginally useful.

Only the last auditing program, Pscan, could accurately find specific vulnerabilities. Pscan is a tool

that looks only for format string vulnerabilities. The Pscan documentation is very clear as to what it reports:

“IF the last parameter of the function is the format string, AND the format string is NOT a static string, THEN complain.”

False positives were caused by #define statements without format characters, such as the following, which is then used in a function that requires a format string (from article.c in innfeed, part of the INN package):

```
#define PREPARE_FAILED "ME internal
failed to prepare buffer for NNTP"
syslog (LOG_ERR,PREPARE_FAILED) ;
```

These are false positives from the point of view of exploitation. However, good practice would be to add the "%s" argument anyway; there is no guarantee that because the string is defined as a constant by the programmers, it doesn’t contain format strings which would require more arguments. So, for the purpose of evaluating coding practice, these notices are useful.

There were several kinds of false negatives. In the first kind of false negative, the last parameter is not the format string. In the second kind, the format string is a static string. The false negatives would happen with an incorrect number of variables following the format string. Pscan doesn’t determine the correct number of variables. However, the typical format string vulnerability arises from programmers too lazy to write "%s", which results in the string to output being the format string.

We used string format vulnerability entries in MITRE’s CVE to benchmark Pscan for false negatives and false positives. A search for “format” and “string” in the description of CVE entries (including candidates) returned 137 entries. However, our choice was limited by our ability to retrieve the relevant source code, such that we were able to examine only 12 entries in the available time; in addition, many of the vulnerabilities happened in firmware or closed source products. However, several entries each involved several vulnerabilities (e.g., CVE-2000-0573), so Pscan was in effect tested many more times.

There was a format string vulnerability indicated by the CVE that Pscan was not able to detect. Closer study of CVE-2002-0251 (in LICQ) revealed that it was a buffer overflow, and this is what the vendor patch fixed. In addition, the wording of the CVE was “possibly”, so we believe that it was not a string format vulnerability.

In addition to looking for the variable argument functions defined in the C library, Pscan had a shell program to look through the software package and find other such variable argument functions defined within the software package. The results were easily converted into an application-specific problem function definition file, such that with a switch in the command

line, Pscan would look for the functions and the arguments that the file described. However, Pscan and its shell program were unable to find the vulnerabilities in CVE-2000-0594. The problem was the use of variable argument functions, with a buried call to `vsprintf` in the function `logmsg` in the file `lastlog.c` (details in the appendix). After changing the shell program from:

```
grep -n '\\.\\.\\.\\. ' $@ | grep ';' | grep
'char'
to:
grep -n '\\.\\.\\.\\. ' $@ | grep 'char'
```

One of the vulnerabilities could then be found, along with 44 other misuses of string formats and variable arguments through the BitchX program. Pscan was also confused by the use of the “?” C operator, as the following (the second vulnerability in CVE-2000-0594) should have generated a warning:

```
logmsg(LOG_KILL, from, 0,
ArgList[1]?ArgList[1]:"(No Reason)");
```

The scan of the most recent version of BitchX, c19, still produced 45 misuses of string formats, although they were all only bad practice, without enabling vulnerabilities.

An interesting false negative was with a function in OpenBSD `top.c` (see CAN-2000-0999) that was defined with a fixed number of arguments and so escaped detection, yet had a format string argument and was called with a variable number of arguments. This happened also in CVE-2000-0947, where CFEngine defined a function `CfLog` with a fixed number of arguments, which then called `syslog`. However, in this case the calls to `CfLog` used the correct number of arguments but the format strings were generated from network input in an unsafe manner, so that ultimately the format strings could be controlled by attackers. Interestingly enough, the authors of CFEngine responded by hacking their logging function to filter out “%” characters instead of fixing the bad code. The “-w” option in Pscan issues warnings for format strings that are not constant and highlights the above coding practices; however it also results in many warnings for correct code, so its usefulness is limited.

Using Pscan on software that already had CVE entries for format string vulnerabilities allowed us to find new vulnerabilities. We found non-exploitable string format issues in `ssh` for OpenBSD, which the OpenBSD developers said they had just fixed a few days earlier. We found CAN-2003-0363, a probably remotely exploitable string format vulnerability in LICQ 1.2.6 and earlier (we checked 1.0.3 as well), and notified the developers. In addition, we found CAN-2003-0311 in CFEngine 2.0.6, which according to the

authors is not exploitable (see appendix). Because these software packages had already been examined for string format vulnerabilities, we believed it would be unlikely that we would find new ones. Yet, Pscan proved useful.

In practice, Pscan was able to find most, but not all, of the vulnerabilities caused by format strings in code. Emboldened by this success, we automated the generation of custom function files for Pscan such that Pscan could be run with minimal effort throughout large source code trees. However, the number of false positives and code style warnings in large, well-maintained, security-oriented projects such as OpenBSD was not acceptable to the developers.

6. Conclusions

We were able to discover several new vulnerabilities or potentially vulnerability-enabling software coding practices by using Pscan. In some software packages, the same bad programming practices remained over a broad range of software versions. While the false positives were easily manageable in small and medium-size projects, in larger projects its usefulness was limited.

Pscan should definitely be used with custom function definition files, as these reduce false negatives without increasing the false positives much. Creating the definition files was trivial with the command-line tool, but tedious and error-prone. We replaced this manual process with a script that makes using Pscan seamless and faster throughout large source trees; we will submit this script to the author of Pscan for concurrent distribution.

For the purpose of evaluating software or for trying to do quick security audits by third parties, vulnerabilities are generally undetectable because they are buried in false positives by the current auditing applications. The only exception is Pscan, which is somewhat useful for finding narrowly-defined format string vulnerabilities. We found ourselves spending much more time looking for the source code of software packages than running Pscan and interpreting the results; whereas this highlights Pscan’s usefulness, we are worried that source code will disappear with time and thus detract from attempts to systematically study vulnerabilities in practical code.

Function spotting auditing applications may be useful for programmers that are unfamiliar with security concepts. By auditing one’s own code, a programmer can get a free hands-on lesson in security. It is even beneficial for intermediate programmers who “know that a particular call may introduce problems, but do not know what the potential problems are” [7].

This allows them to get a better understanding of how security problems occur, rather than just telling them what not to do. Pscan is the only auditing application that we felt was useful for evaluating software or for third party reviewers.

We conclude that auditing applications that have few false positives, even though they may have false negatives, are more useful than applications that flag everything. Specialized tools may be more useful. However, even a specialized tool like Pscan needs to be more sophisticated in order to have no false positives and no false negatives.

Our last conclusion is that the state of the technology for (security oriented) code auditors leaves much to be desired; expensive, time-consuming security code reviews by experts are under no threat of becoming obsolete. Yet, we feel that the development of powerful code auditors could be extremely beneficial to software quality and security, while saving money for developers and customers. The gap between the latest advances being made in the field of static analysis (e.g., [30]) and the unsophisticated (security-wise) code auditors available today for third-party auditing is significant. This means that it is possible to see much better auditors being developed in the future, because the limitations highlighted in this study are not of a theoretical nature.

The next generation of code auditor needs to be at least as intelligent as a compiler, and could probably function as a compiler module, in order to understand macros, the context of the function calls and how data could be (incorrectly) used, passed, or returned. We believe that by adopting more advanced static analysis methods, potential vulnerabilities could be found more reliably. However, exploitability conditions can be very difficult to establish, even for humans, so a dual approach of static analysis and targeted dynamic analysis based on the results of the static analysis and directed towards establishing exploitability, could be highly successful. This research area requires more attention. We hope to be able to revisit this study at a later date, with much better results.

7. References

- [1] Blanchet B. "NWG1: Static Analysis - Summary (MPII)" <http://www.mpi-sb.mpg.de/units/nwg1/summary.html>
- [2] Static Analysis <http://www.ics.uci.edu/~djr/classes/ics224/lectures/08-StaticAnalysis.pdf>
- [3] CS 390S -- Code Injection and Symlinks <http://www.cs.purdue.edu/homes/cs390s/LectureNotes/cs390sIV.pdf>
- [4] Free Online Dictionary of Computing <http://foldoc.doc.ic.ac.uk/foldoc/index.html>
- [5] Sardonix Security Portal, <http://www.sardonix.org/>
- [6] Security Focus home vulns discussion: Multiple Vendor FTP Conversion Vulnerability <http://www.securityfocus.com/bid/2240/discussion/>
- [7] ITS4: A Static Vulnerability Scanner for C and C++ Code <http://www.cigital.com/papers/download/its4.pdf>
- [8] CS 390S -- Buffer Overflows <http://www.cs.purdue.edu/homes/cs390s/LectureNotes/cs390sBO.pdf>
- [9] Cqual README file <http://www.cs.berkeley.edu/~jfoster/cqual/>
- [10] Splint Homepage <http://lclint.cs.virginia.edu/>
- [11] FlawFinder Home Page <http://www.dwheeler.com/flawfinder/>
- [12] PC-lint/FlexeLint Product Information <http://www.gimpel.com/html/lintinfo.htm>
- [13] Wu-Ftpd Remote Format String Stack Overwrite Vulnerability <http://www.securityfocus.com/bid/1387/discussion/>
- [14] Wu-ftpd 2.6.0 patch for format string vulnerabilities <http://downloads.securityfocus.com/vulnerabilities/patches/wu-ftpd2.6.0.diff>
- [15] Wu-Ftpd Debug Mode Client Hostname Format String Vulnerability <http://www.securityfocus.com/bid/2296/discussion/>
- [16] Wu-ftpd patch for Debug Mode Client Hostname Format String Vulnerability ftp://ftp.wu-ftpd.org/pub/wu-ftpd/patches/apply_to_2.6.1/missing_format_strings.patch
- [17] Wu-ftpd patch for Wu-Ftpd File Globbing Heap Corruption Vulnerability <ftp://ftp.leo.org/pub/FreeBSD/ports/distfiles/wu-ftpd/ftpglob.patch>
- [18] Posadis DNS Server Logging Format String Vulnerability <http://www.securityfocus.com/bid/4378/discussion/>
- [19] Multiple Vendor BSD eeprom Format String vulnerability <http://www.securityfocus.com/bid/1752/discussion/>
- [20] Patch for fixing the Multiple Vendor BSD eeprom Format String vulnerability http://www.tux.org/pub/bsd/openbsd/patches/2.7/common/028_format_strings.patch
- [21] AutoNice Daemon Program Name Format String Vulnerability <http://www.securityfocus.com/bid/3580/discussion/>
- [22] Weber, M.; Shah, V.; Ren, C.; (2001) A case study in detecting software security vulnerabilities using constraint optimization. Proceedings of First IEEE International Workshop on Source Code Analysis and Manipulation, pp. 1-11
- [23] Oulu University Secure Programming Group (2001) PROTOS - Security Testing of Protocol Implementations <http://www.ee.oulu.fi/research/ouspg/protos/index.html>
- [24] Hill, G., (1988) A rule-based software engineering tool for code analysis. Seventh Annual International Phoenix Conference on Computers and Communications, pp. 291-295
- [25] Computer Security Institute. <http://www.gocsi.com/press/20020407.html>
- [26] ICAT statistics. <http://icat.nist.gov/icat.cfm?function=statistics>

- [27] Mell P., Tracy M.C. (2002) NIST Special Publication (SP) 800-40, Procedures for Handling Security Patches.
- [28] Miller B.P., Fredriksen L. and So B. (1990) Study of the reliability of UNIX utilities. Communications of the ACM 33, pp 32-44
- [29] Frantzen M. (1999) ISIC (IP Stack Integrity Checker), <http://www.nestonline.com/TrinuxPB/isic.txt>
- [30] Blanchet B., Cousot P., Cousot R., Feret J., Mauborgne L., Miné A., Monniaux D., and Rival X. "A Static Analyzer for Large Safety-Critical Software". In ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI'03), pages 196-207, San Diego, California, June 2003. ACM.

8. Appendix: Format String vulnerabilities investigated

8.1. CVE-2000-0573: Wu-ftpd 2.6.0

This vulnerability affected versions of Wu-ftpd 2.6.0 and earlier on most platforms. Pscan found each error in the source code of wu-ftpd 2.6.0, which was verified by the corresponding patch. There were two instances in the original code that called `Ireply` incorrectly ("`Ireply(200, cmd)`" on line 1875 and "`Ireply(200, buf)`" on line 1881) in the file `src/ftpcmd.y`. There were also two instances that called "`setproctitle(proctitle)`" on lines 2907 and 5208 of `src/ftpd.c`. The important thing to notice is that these are not C library functions; they are defined in the wu-ftpd package. The patch dictates that those above lines be removed and be replaced with "`Ireply(200, \"%s\", cmd)`", "`Ireply(200, \"%s\", buf)`", and "`setproctitle(\"%s\", proctitle)`" [14].

8.2. CVE-2001-0187: Wu-ftpd 2.6.1

Pscan found all instances of incorrect uses of `syslog` (defined in the patch). In wu-ftpd 2.6.1, these errors were found at lines 6277 and 6292 in file `src/ftpd.c`, both in the form "`syslog(LOG_DEBUG, s)`". The patch file dictates that those lines be removed and replaced with "`syslog(LOG_DEBUG, \"%s\", s)`" [16]. The patch also corrects misuses of the `reply` function, a variable argument function defined in the wu-ftpd software package. Pscan found the two misuses of `reply` corrected by the patch in the `src/ftpd.c` file. On line 6277, "`reply(530, salt)`" is to be changed to "`reply(530, \"%s\", salt)`"; on line 6438, "`reply(550, globerr)`" is to be changed to "`reply(550, \"%s\", globerr)`" [16].

8.3. CVE-2001-0550: Wu-ftpd 2.6.1

This vulnerability was related to the problem of file globbing in the `ftpcmd.y` file. Although Pscan could catch many of the lines that related to error reporting through the `reply` function (like examples above) in

CVE-2001-0187, it could not catch any of the errors that related to changing the control flow that were fixed in the patch file [17].

8.4 CAN-2002-0501: Posadis m5pre1

Posadis, "an open source implementation of a non caching DNS server" has a format string vulnerability where it reports errors in version m5pre1 [18]. There was no patch to be found to fix this problem, but the updated code was available. This gave us an interesting task of letting Pscan find the errors, and rather than checking them with the patch, checking them with the next version of the program, m5pre2. After running Pscan on all of the source code files, two errors came up in the `src/log.cpp` file. The first was on line 70, where posadis makes the call "`printf(buff)`", and the second on line 71, "`fprintf(logfile, buff)`". When comparing it to the same section (though not necessarily the same line) of the source code in m5pre2, these calls are replaced by "`puts(buff)`" and "`fputs(buff, logfile)`", respectively. These are safe substitutions for the format string vulnerability problem.

8.5. CVE-2000-0997:OpenBSD/NetBSD

The OpenBSD and NetBSD (V. 2.3 – 2.7) operating systems shipped with sparc versions have a format string vulnerability relating to its `eeprom` utility for writing to hardware [19].

Pscan caught the vulnerabilities except that it was confused and produced false negatives with things like:
"`fprintf(stdout, SSH_COM_MAGIC_BEGIN "\n")`"
which, if they had been flagged, could have been considered false positives depending on the goals of the auditor.

8.6. CVE-2001-0920: AND

The vulnerability in the AND (AutoNice Daemon) program occurs if a process name is a format string [21]. Just as in the Posadis example above, we were not given a patch to work with, but were given the code for the next version. After running Pscan, it found a line of code that was changed from "`syslog(LOG_WARNING, buffer)`" to "`syslog(LOG_WARNING, \"%s\", buffer)`", located on line 221 of the `and.c` file of the source code.

8.7. CAN-2003-0311: CFEngine 2.0.6

Line 915 of the file `src/cfexecd.c` calls "`snprintf(buf, bufsize - 1, s)`", which misuses the `sprintf` call. It should be "`snprintf(buf, bufsize - 1, \"%s\", s)`". The authors were notified and the change has been made for the next version of CFEngine. However, the string

s came from a trusted user with root access, so was not likely exploitable.

8.8. CAN-2002-0525: INN 2.2.3

“Format string vulnerabilities in (1) inews or (2) rnews for INN 2.2.3 and earlier allow local users and remote malicious NNTP servers to gain privileges via format string specifiers in NNTP responses. “ Because the entry mentions several issues, we cannot be sure that we found all of them. Pscan found many “false positives” in this software, e.g., missing “%s” that are bad practice but do not cause vulnerabilities. Misc.c in innfeed contained the function “logOrPrint“, with the following code (sic):

```
“char buffer [512] ; /* gag me */
vsprintf (buffer,fmt,ap) ;
syslog (level,buffer) ;”
```

In addition, group.c in (nnrpd) contained:

```
“sprintf(data, buffer.Name);”
```

8.9. CVE-2000-0594: BitchX

The BitchX IRC client does not properly cleanse an untrusted format string, which allows remote attackers to cause a denial of service via an invite to a channel whose name includes special formatting characters. We found the responsible function again in lastlog.c:

```
int logmsg(unsigned long log_type, char
*from, int flag, char *format, ...)
```

with the vulnerability enabling calls in parse.c:

```
logmsg(LOG_INVITE, from, 0,
invite_channel);
logmsg(LOG_KILL, from, 0,
ArgList[1]?ArgList[1]:“(No Reason)”);
```

8.10. CAN-2000-0999: OpenBSD issues

This actually represents many different vulnerabilities:

1 in ssh/auth2.c required custom pscan file
2 in ssh/auth-rsa.c required custom pscan file

2 in usr.bin/top/top.c not found!!!

1 in usr.bin/su/su.c (found)

1 in usr.bin/passwd/yp_passwd.c (found)

1 in usr.bin/fstat/fstat.c (found)

and we found “new” issues in ssh/auth-skey.c and in ssh/auth2.c, that have been fixed in OpenBSD 3.3.

The false negative occurred in top.c by calling a function called “new_message”. The reason Pscan did not catch this is that it does not present itself as a normal variable argument function. Most of them are defined in the form “void new_message(type, msgfmt, ...)”, where the ellipses at the end indicate that a variable number of arguments can follow. However, in this case, this function definition took the form “void new_message(type, msgfmt, a1, a2, a3)”. This indicates a limited number of arguments to follow (which can be seen in the display.c code in which it is defined). Although this does present a format string vulnerability, it is impossible for Pscan to find it, because it occurs within the new_message function definition due to a sprintf call.

8.11. CAN-2002-0702: ISC DHCP

Format string vulnerabilities in the logging routines for dynamic DNS code (print.c) can be found with a custom pscan file.

8.12. CVE-2000-0947: CFEngine before 1.6.0a11

Many similar false negatives; see text.