Characterizing the 'Security Vulnerability Likelihood' of Software Functions

Dan DaCosta, Christopher Dahn, Spiros Mancoridis, Vassilis Prevelakis
Department of Computer Science
Drexel University, Philadelphia, PA, USA
{Daniel.William.DaCosta, Christopher.Stephen.Dahn,
Spiros.Mancoridis, Vassilis.Prevelakis}@drexel.edu

Abstract

Software maintainers and auditors would benefit from a tool to help them focus their attention on functions that are likely to be the source of security vulnerabilities. However, the existence of such a tool is predicated on the ability to characterize a function's 'security vulnerability likelihood.'

Our hypothesis is that functions near a source of input are most likely to contain a security vulnerability. These functions should be a small percentage of the total number of functions in the system. To validate this hypothesis, we performed an experiment involving thirty one vulnerabilities in open source software. This paper describes the experiment, its outcome, and the tools used to conduct it. It also describes the FLF Finder, which is a tool that was developed using knowledge gathered from the outcome of the experiment. This tool automates the detection of high risk functions. To demonstrate the effectiveness of the FLF Finder, three open source applications with known vulnerabilities were tested. In addition to this test, a case study was performed on the privilege separation code in the OpenSSH server daemon.

1. Introduction

A *software vulnerability* is a fault in the specification, implementation, or configuration of a software system whose execution can violate an explicit or implicit security policy [15]. Software maintainers typically focus on the functionality of software rather than on its security posture. Hence, vulnerabilities often escape their attention until the software is exploited by specially written malicious code.

A large percentage of software is developed using unsafe programming languages (e.g., C and C++) in the name of cost effectiveness, programmer familiarity, and performance. Being unable to influence how others develop new software, we must find ways to improve the maintenance process to secure software against possible attacks.

Code audits are one aspect of the maintenance process that can focus on security vulnerabilities, and have been tried, with some success, on systems such as the OpenBSD operating system [23]. Unfortunately, audits are expensive and reoccurring. Each audit requires many man hours, and each software revision requires re-examination to verify that new faults have not been introduced.

The quantity of code in many systems makes large-scale auditing infeasible. In the case of OpenBSD, the auditing effort only focuses on software that is enabled in the default installation. This decision has resulted in overlooked vulnerabilities in other often used components of the distribution, such as `telnetd`.

Beizer states that good source code will have one to three faults for every one hundred lines of code [4]. However, it is not known which of those faults is a security vulnerability. Auditors would benefit from a tool that can reduce the amount of code that needs to be studied; enabling them to focus their attention on areas of likely vulnerability.

Our hypothesis is that a small percentage of functions near a source of input (e.g., file I/O) are the most likely to contain a security vulnerability. Exactly what we mean by 'near' is described in Section 3.3. We refer to these functions as *FLFs* (Front Line Functions), and the percentage of functions likely to contain a security vulnerability as the *FLF density*. We validate our hypothesis with an experiment that involves thirty one vulnerabilities in open source software using two tools that were developed for this purpose. The results of this experiment are summarized in Table 1.

Based on the validation of the hypothesis, the FLF Finder tool was developed to identify areas of high vulnerability likelihood automatically. The effectiveness of the FLF Finder is demonstrated in two ways. First, it is applied to three open source software systems, micq, elm, and dhcpd, with known (documented) vulnerabilities. Second, the FLF Finder is applied to the OpenSSH daemon software, which does not have known vulnerabilities but has recently undergone a widely-publicized restructuring, called privilege

separation, aimed at minimizing the amount of code that runs with elevated privileges. By minimizing the amount of privileged code, it reduced the risk of a security vulnerability occurring within that code. Although the restructuring was done manually, our case study shows that the results produced by the FLF Finder are consistent with the design decisions made by the security auditors.

The remainder of this paper is structured as follows: Section 2 outlines related research, Section 3 presents the experiment, Section 4 describes how the experiment's outcome was used to develop the FLF Finder, Section 5 details the OpenSSH case study, and Section 6 outlines the limitations of this work and our future plans. The paper concludes in Section 7.

2. Related Research

This work is related to two major areas of research. The first is software security, specifically as it applies to security vulnerabilities in code. The second is the use of source code analysis for software maintenance.

2.1. Software Security

2.1.1. Common Security Vulnerabilities

There are many classifications of software security vulnerabilities [2, 14, 5]. This work considers only the categories of vulnerabilities that involve obtaining external input data that causes a security exploit.

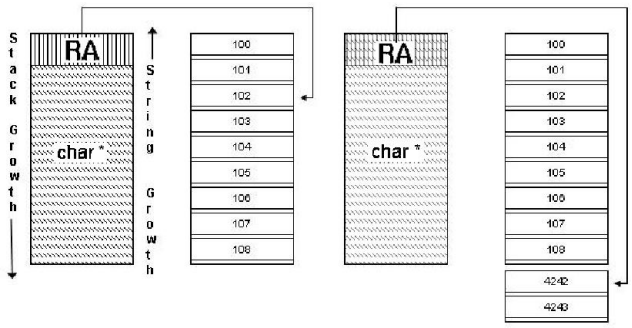

Figure 1. Buffer Overflow

Buffer overflows may occur when a fixed size memory allocation is used to store a variable-size data entry. There are conflicts when the variable size data entry overruns the bounds of the fixed-size memory, as show in Figure 1. These overflows are typically exploited by entering a string which is larger than the buffer assigned to hold it. If the return address (RA) is part of the overwritten run-time stack, the user may execute arbitrary code [22].

Format string vulnerabilities target specific types of C and C++ function calls such as `printf` and `sprintf`. This family of functions accepts an input string that specifies the format of the output along with an arbitrary number of arguments that correspond to that string, called the variable arguments list. The function pushes the arguments onto the run-time stack and then reads the format string, popping the arguments back off of the stack. If the programmer allows the user to specify the format string at run-time, the user could request more data to be popped off the stack than the programmer had originally pushed. This unpredictable behavior could lead to a denial of service attack, or a buffer overflow [20].

Denial of service attacks involve giving the application an unexpected input that causes it to disallow service to any other user. For example, the input may cause the application to enter an infinite loop or to crash. Many denial of service attacks target the network infrastructure [17], but this work only pertains to attacks on applications.

2.1.2. Technologies to Detect & Prevent Vulnerabilities

Our effort is not the first to address the problems of vulnerabilities in software. Some of the existing security tools, such as Splint and cqual, perform static analysis to find code that is likely to be vulnerable. Unlike our tools, however, they require programmers to annotate their source code with constraints. Not all of the existing source code analysis tools require code annotations, however. Flawfinder, RATS, ITS4, and Mops are all tools that analyze source code and report possible weaknesses.

Other tools, such as StackGuard and FormatGuard, attempt to detect and handle vulnerabilities at run-time. Finally, some vulnerabilities can be prevented in hardware and operating systems by using non-executable memory pages.

Flawfinder [28], RATS [26], and ITS4 [8] are all tools that examine source code and report possible weaknesses. An overview of these tools, along with a comparison of their capabilities, can be found in an Linux Journal article [19]. In general, these tools direct the attention of code auditors to C/C++ functions that are known to be associated with security problems (i.e., buffer overflows, format string problems, temporary file race conditions) and produce a list of vulnerable code statements.

The Splint utility [16] allows source code annotations to be inserted to determine if a buffer overflow is possible for certain portions of the code. The tool assumes that the developer will add all of the necessary annotations correctly.

Cqual [27] is a tool that detects format string vulnerabilities through source code analysis. It allows programmers to use two new data type modifiers, `tainted` (untrusted data) and `untainted` (trusted data), that can be applied to

any data type in the source code. The tool uses a modified compiler on the annotated source code and informs the programmer when mismatches occur between `tainted` and `untainted` types.

Our tool should be used in conjunction with these five tools. Specifically, the FLF Finder reduces the number of functions the auditor should look at. The auditor may then apply any of the other tools to find specific lines of code that may be vulnerable.

Model checking has been used with some success to detect possible vulnerabilities in software. For example, MOPS [6], uses state machines to detect patterns of function call sequences that commonly result in security vulnerabilities.

StackGuard [11] has been reasonably successful at reporting buffer overflows immediately after they happen. Specifically, StackGuard inserts code into the application at compile time and a 'canary' value just before the return address on the run-time stack. When the function returns, the added code checks if this canary value is still in place. If the canary value is no longer present, then a buffer overflow must have occurred that overwrote the value. When this happens, the application terminates with a notification.

FormatGuard [10] is used to detect format string vulnerabilities. It provides protection by using a proxy API composed of C macros that intercept predetermined vulnerable functions. These macros count the number of operands in the format string and the number of arguments passed to the function via the variable arguments list. If these numbers do not match, FormatGuard flags the program as compromised and does not invoke the vulnerable function.

An effective way to avoid the security problems of an exploited vulnerability is to disallow the execution of the run-time stack altogether. By allowing memory pages to be set as non-executable, it prevents executable code that may have been placed on the run-time stack during a buffer overflow from being executed. This measure can still be defeated in some cases and also dictates that programs must be reentrant.

2.2. Source Code Analysis

Code analysis tools parse source and produce a data repository of source code facts. Maintainers can then formulate queries against the data repository when trying to understand a program.

A C and C++ code analyzer has been developed at AT&T called Acacia [7, 3]. Acacia uses the Edison Design Group's (EDG) compiler to parse the source code, but only accepts code written in ISO C/C++ or K&R C. This is a limitation for our research since we are studying open source systems, which often contain third-party GNU extensions that do not conform to ISO or K&R standards.

A similar tool is the GNU C Compiler (GCC) XML Code Introspector [12]. Introspector converts the GCC abstract syntax tree (AST) into either XML or a Postgres relational database.

Other examples of code analyzers are Reprise [25] for C++, Chave for Java [**?**] and Cobol/SRE [21] for COBOL.

Some of the mentioned tools display their output as simple text or as a graph using visualization tools from the Graphviz [3] package.

3. FLF Experiment

This section describes the experiment to validate the FLF hypothesis. The experiment involves a suite of thirty one vulnerabilities in open source software written in C that have known (documented) security vulnerabilities. The first column of Table 1 shows the name of each open source system. If a system contains more than one vulnerability, the name of the system is followed by a '-' and the vulnerability number (e.g., zgv-1, zgv-2).

3.1 Inputs and Targets

For the purpose of the experiment, we refer to the functions that accept input as *Inputs* and the functions with known vulnerabilities as *Targets*.

An example of an Input is a user defined function that contains a call to `read` in `unistd.h`. The function `read` stores data from a potentially untrusted source into a buffer. Our analysis revealed that the most common sources of input data are direct user input, indirect user input via command line arguments, and input from environment variables. The FLF Finder supplies a list of common Inputs such as `read`. However, any function could be a potential Input, so the user may specify what other functions in the software system should be considered an input.

A Target is any function that contains a known vulnerability. These functions typically use a global buffer or a variable parameter that contains data from an Input. For example, a Target could be a function that calls `printf` using a user supplied buffer as the first argument.

All of the open source systems used in the experiment have at least one known vulnerability and one patch file for repairing the vulnerability. The patch files are created by maintainers using the Unix `diff` tool. The experiment uses patch files to identify the Targets in each system automatically.

Figure 2 shows a typical patch file that repairs a security vulnerability. The *subtractive* lines (i.e., lines that begin with a '-') lines in the patch file specify the original source code that must be removed to repair the vulnerability. We consider the functions that contain the subtractive lines to be Targets.

```
--- channels.c  Thu Mar  7 15:01:32 2002
+++ channels.c  Thu Mar  7 15:11:55 2002
@@ -145,7 +145,7 @@
{
    Channel *c;

-   if (id < 0 || id > channels_alloc) {
+   if (id < 0 || id >= channels_alloc) {
      log("channel_lookup: %d: bad id",
          id);
      return NULL;
    }
```

Figure 2. An OpenSSH patch file

3.2. Tools

The experiment uses two tools we developed. GAST-MP (GNU Abstract Syntax Tree Manipulation Program) takes pre-processed C source code and generates a database of code facts for each system in the experiment. SGA (System Graph Analyzer) discovers Targets in the source code and creates function call graphs that are necessary.

3.2.1. GAST-MP

The GNU C Compiler (GCC) version 3.21 is able to output the AST (abstract syntax tree) that it produces into an ASCII text file when given the `-fdump-tree-original` flag. GAST-MP parses this file and produces a relational database of code facts.

3.2.2. SGA

The SGA tool has a dual purpose. It functions as a vulnerability patch file analyzer to identify Target functions and as a function call graph generator that is used to trace how potentially dangerous data could flow from Inputs to Targets.

For each vulnerability patch file, SGA determines the line number of each subtractive line in the corresponding source code. It then uses the GAST-MP database to find the function that contains that line of code. Once the function is determined, it is marked as a Target.

3.3. Function Invocation Path Model

Recall that the FLF hypothesis states that a small percentage of functions, specifically those near a source of input, are most likely to contain a security vulnerability. We will show that thirty one vulnerabilities in open source software occur within close proximity to an Input. The proximity is measured as the number of function invocations that occur between the Input and Target.

The FLF density k for each Input and Target pair is the ratio p/m, where p is the number of functions on the longest achievable call path between the Input and Target, and m is the total number of functions in the system.

Assuming that asynchronous code is not taken into account (see Section 6), there are three possilble invocation paths from an Input to a Target that will contain all data flow paths. Given an Input node i and a Target node t, one can have a invocation path $i \rightsquigarrow t$. Given the same pair of nodes, one can also have a invocation path $t \rightsquigarrow i$. Finally, given a graph $G = (V, E)$ and a node $n \in V$ such that n is neither an Input nor a Target , there can exist a data flow such that $n \rightsquigarrow i$ and $n \rightsquigarrow t$. Between these three case, under our assumption, all possible data flow paths must be contained.

We know that the call graph contains all of the possible data flow paths between Input and Target. Hence, all data flow paths between Input and Target must not be contained in any invocation path longer than the longest invocation path between Input and Target on the call graph. As shown in Section 3.4, we always choose the longest invocation path. By choosing the longest invocation path, we are make conservative estimate of the length of the data flow path that is actually executed.

3.4. FLF Hypothesis Validation

The validation consists of four stages. The first stage is to search for software systems with known vulnerabilities and patch files for those vulnerabilities. In general, it is difficult to find patch files that only pertain to security vulnerabilities since maintainers make one general patch file that contains fixes for both regular faults and security vulnerabilities. Fortunately, some Linux distributions provide software in the form of Source Red Hat Package Manager (SRPM) files. SRPMs contain unaltered source code and a set of patches that address specific faults in the source. SRPM packages comprise much of the test suite.

The second stage is to pre-process each software system in the test suite with GCC to resolve macros and compile time dependencies. GAST-MP is then used to generate a database of code facts for each system.

Finally, SGA is used to calculate the FLF density of each system. The process to calculate FLF density is as follows:

1. Create the entire call graph G for the system
2. Transform G into a directed acyclic graph (DAG) with a root node. Call this new graph G'
3. Label Input nodes in G'

4. Use the subtractive lines in the vulnerability patch files and code facts from the GAST-MP database to determine and label Target nodes in G'
5. Discover the set of common ancestors, C, for each Input-Target pair in G'
6. Calculate p for each Input-Target pair, which is the number of functions along the longest path from Input to Target through a member of C
7. Calculate the total number of functions, m, in G'
8. For each Input-Target pair, compute the FLF density, $k \leftarrow p/m$
9. Select the largest k as the FLF density for that system

It is important to transform G into a rooted DAG since the common ancestors algorithm [13] requires a DAG to perform correctly. By choosing the largest result, we obtain a conservative estimate of the number of functions that had access to data from the Input. A conservative estimate will reduce the likelihood of false negatives (functions that have access to data from an Input, but are not chosen for audit) created by the FLF finder.

Our experiment, computed the FLF density for each system. The sample mean FLF density ($\bar{x}$) across all systems is 2.87% with a standard deviation (δ) of 1.83%. This means that, on average, 2.87% of the functions in each system were involved in the security vulnerability documented by the patch files. We can say, with 95% confidence, that the true mean FLF density (μ) is within the interval of 2.23% to 3.51%.

4. FLF Finder

The FLF Finder discovers those functions in the code that are at higher risk of vulnerability. The tool is not intended to find faults, only to show which functions are at risk. The FLF Finder requires two pieces of information to be provided by the user. The first is the source code to be analyzed, and the second is a list of Inputs (besides those provided by the FLF Finder). Table 2 lists the default Inputs that the FLF Finder looks for. Each column lists functions from a specific system library.

The process the FLF Finder uses is as follows:

1. Create the entire call graph G for the system
2. Label Input nodes in G
3. Compute the total number of functions, m, in G
4. Given the FLF density result of 2.87%, solve for $p \leftarrow km$
5. Label the nodes in G as follows:

```
StopDepth ← p;
InEdgeDepth ← 0;
while Input has unlabeled ancestor ∧
InEdgeDepth < StopDepth do
    traverse incoming edge;
    label node;
    increment InEdgeDepth;
    OutEdgeDepth ← InEdgeDepth;
    while node has unlabeled predecessor ∧
    OutEdgeDepth < StopDepth do
        traverse outgoing edge;
        label node;
        increment OutEdgeDepth;
    end
end
print labeled nodes;
```

This process is identical to that used in Section 3.4, except that it might produce false positives. The false positives are introduced because Targets are not known ahead of time. In the experiment, the resulting FLF density was based on one path through one common ancestor. Since the Targets are not known ahead of time when the FLF finder is used, every function invocation path of length p through every common ancestor is suspect.

To test the effectiveness of the FLF Finder, we applied it to three open source software systems with known vulnerabilities that were not used in the experiment, micq, elm, and dhcpd. These systems were found in the same manner as discussed in Section 3.4. The results of these tests are shown in Table 3.

The table show three pieces of information. The first, *Function Coverage*, is the percentage of the functions in the system that were labeled as FLFs. This number is much larger than the experimental amount (i.e., 2.87%) due to the introduction of false positives described previously. Next, the table lists how many known vulnerabilities exist in the test system, and how many of the functions that contained vulnerabilities were identified using the FLF Finder. Only `dhcpd` failed to identify all of its known vulnerabilities. It failed to find one vulnerability because there is no known path to the function in `dhcpd` which contains the vulnerability.

One of our objectives was to supply the maintainer with a tool that eases the process of performing a security audit. The FLF Finder accomplishes this by eliminating most of the system's functions from consideration.

The next section describes how the FLF Finder was applied to the OpenSSH secure shell daemon software.

5. OpenSSH Case Study

This case study test the validity of our tools, concepts, and observations by using them against a software system and then verifying the results against the decisions made by the developers who audited the code for security. It is our opinion that if the FLF Finder notes those functions suspect that the developers also noted as suspect, then our FLF hypothesis is one which can have an impact on the software security community.

5.1. OpenSSH

OpenSSH is a freely available suite of network connectivity tools that a growing portion of the Internet is relying on. `Telnet`, `rlogin`, `ftp`, and other such programs transmit unencrypted password information during authentication. OpenSSH encrypts all traffic (including passwords) to eliminate eavesdropping, connection hijacking, and other network-level attacks.

OpenSSH includes `sshd`, a secure alternative to `telnetd` and `sftp`, a secure alternative to `ftp`. Privilege separation was originally an optional part of the OpenSSH architecture. However, the OpenSSH team made it mandatory as of version 3.2.3 [1].

5.2. Privilege Separation

The principle behind privilege separation is to minimize the amount of code that runs with elevated privileges without limiting the functionality of the program. The result is that the separated code(which runs with elevated privileges) can now be audited thoroughly due to its small size, and therefore should be more secure. After separation, the number of lines of `sshd` code that needed to be audited shrunk from 27,000 to 2,500.

The remaining lines of unprivileged code execute in what is known as a *jail*. The jail is a specially constructed directory which works with a user that only has access to this directory and has no other privileges. Any attempts to exploit code within this jail should result in either a denial of service to the attacker or arbitrary code execution as the restricted user in an environment isolated from the rest of the operating system [24].

5.3. Case Study

The goal of the following case study is to demonstrate that the FLF concept and its associated tools can be used to increase the efficiency of a source code auditor. This will be done by comparing the FLF Finder results of OpenSSH 3.1(3.1) against OpenSSH 3.2.3(3.2.3). 3.1 was the last version that did not run with mandatory privilege separation, and 3.2.3 was the most recent version as of the time the case study was performed. The criteria for a successful study will be the agreement of the set of FLF's with the changes made in 3.2.3. More explicitly, we wish to show that those FLF's noted by the FLF Finder contain those functions modified by the experience developers in 3.2.3

The changes made in 3.2.3 add 78 functions which specify a message passing API. These 78 functions are used primarily by legacy functions to pass data between the privileged and unprivileged segments of the system. It is our belief that the developers were implicitly enforcing FLF bounds on 3.1 by choosing where to place the new 78 functions. Therefore, we will run our FLF finder on 3.1 to study how many of these legacy functions the FLF finder also notes as high risk functions.

Those functions which we are interested in finding are located in the following manner, we let $LF_{3.2.3}$ be the set of all functions in 3.2.3 which use the message passing API and let $F_{3.1}$ be the set of all functions in 3.1. Then let $F_{intersects} \leftarrow LF_{3.2.3} \cap F_{3.1}$ such that $F_{intersects}$ now contains only those functions in 3.1 which where changed to use the message passing API in 3.2.3. We now apply the FLF finder to 3.1 to generate FLF, the set of FLFs in $F_{3.1}$. Finally we let $F_{success} \leftarrow FLF \cap F_{intersects}$, $F_{success}$ is the set of all functions which were found by the FLF finder and also were used in 3.2.3 to call message passing functions. Therefore we can determine our correctness by $|F_{success}|/|F_{intersects}|$. When this procedure was applied we were $\%82(14/17)$ successful at locating those functions which the openssh developers also implicitly defined as FLF's. Of the total 1029 functions openssh 3.1 the FLF finder identified 374 or $\%34$ as potential targets.

This case study presents an example of how the FLF concepts and tools can be used to aid code auditors in finding high risk areas of code in an efficient manner. We say this, because we were able to remove and estimated $16,448$ lines of code with very little effort while maintaining a high degree of accuracy. As our tools mature and our experimental set becomes larger we can assume that our ability to trim those unimportant segments of a system(in terms of security) will only improve. Preliminary versions of this process which use data flow as a method of further reducing extraneous function paths have shown even more dramatic results maximizing our success rate while minimizing lines of code. This implies that code auditors will be able to spend thier time on those functions that are in the most need of attention.

6. Limitations & Future Work

This section describes limitations associated with our current work as well as proposed solutions and future im-

provements.

6.1 Function Pointers

Function pointers cause call-paths to be disconnected because function pointers are resolved at run-time. In some cases, it is possible to discover the set of functions that a function pointer can resolve to, and include that branching in the call-graph.

6.2 Asynchronous Executable Code

Static analysis of asynchronously executable code, such as threads and signal handlers, is non-deterministic. Call graphs that contain these constructs appear disconnected because it is impossible to guarantee that an association will be found to connect the data flow paths and thereby the invocation paths.

6.3 Future Improvements

The first improvement is to support C++ in addition to C. This will enable us to use a much larger experimental test suite.

The integration of the tools described in this paper into our reverse engineering portal, called *REportal* [18], would widen the impact of the work significantly. The web portal enables developers to upload source code and perform specialized analysis online using a web browser.

Currently, all macros are lost during the pre-processing phase of the source code analysis. Modification to the preprocessor will allow macro associations to be included in the database and, hence, used in our analysis.

The querying mechanism supported by SGA is primitive. In the future, we would like to enhance SGA with a more expressive query language that would support, among other things, transitive closure.

7. Conclusions

Our hypothesis states that a relatively small percentage of functions near a source of input are the most likely to contain a security vulnerability. Described in this paper is an experiment to validate this hypothesis. The results of this experiment, shown in Table 1, support our hypothesis.

In Section 3 we computed the FLF density of each system in the test suite. Recall that the FLF density k for each Input and Target pair is the ratio p/m. p is the number of functions on the longest achievable call path between the Input and Target, and m is the total number of functions in the system. We found that the sample mean FLF density was2.87% with a standard deviation of1.83%. From this we can generate a 95% confidence interval for the true mean FLF density from 2.23% to3.51%.

These results were tested against several open source software systems not included in the experiment, as well as the OpenSSH server daemon. The case study showed that the design decisions made by the OpenSSH team concur with the results our FLF Finder produced.

By using the FLF Finder, code auditors can focus their attention on the most vulnerable functions in the system. This would allow them to spend more of their time searching for less obvious security flaws in systems, leading to more secure applications.

Acknowledgments

Effort sponsored by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-01-2-0534. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon.

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA), the Air Force Research Laboratory, or the U.S. Government.

Dr. Angelos Keromytis was especially helpful in the validation of the FLF Finder tool against the OpenSSH system.

Special thanks to Jeffrey White for helping us set up the FLF experiment.

References

[1] OpenSSH website. http://www.openssh.org.

[2] T. Aslam. A Taxonomy of Security Faults in the Unix Operating System. Master's thesis, Purdue University, 1995.

[3] AT&T Labs - Research Tools http://www.research.att.com/sw/tools.

[4] B. Beizer. *Software Testing Techniques*. International Thomson Computer Press, 1990.

[5] M. Bishop. A Taxonomy of UNIX System and Network Vulnerabilities. Technical report, Department of Computer Science, University of California at Davis, May 1995.

[6] H. Chen and D. Wagner. Mops: An infrastructure for examining security properties of software. http://www.cs.berkeley.edu/ hchen/publication/ccs02.pdf.

[7] Y. Chen, E. R. Gansner, and E. Koutsofios. A C++ Data Model Supporting Reachability Analysis and Dead Code Detection. In *Proceedings of the European Conference on Software Engineering/Foundations of Software Engineering*, 1997.

[8] Cigital. ITS4 http://www.cigital.com/its4/.
[9] CoSAK Case Studies Page, http://serg.mcs.drexel.edu/cosak/case_study/.
[10] C. Cowan, M. Barringer, S. Beattie, and G. Kroah-Hartman. FormatGuard: Automatic Protection From printf Format String Vulnerabilities. In *Proceedings of the 10th USENIX Security Symposium*, August 2001.
[11] C. Cowan, C. Pu, D. Maier, H. Hinton, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. Automatic Detection and Prevention of Buffer-Overflow Attacks. In *Proceedings of the 7th USENIX Security Symposium*, January 1998.
[12] GCC XML Code Introspector. http://introspector.sourceforge.net/.
[13] D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal on Computing*, 13(2):338–355, May 1984.
[14] I. Krsul. *Computer Vulnerability Analysis Thesis Proposal*. PhD thesis, Purdue University, 1997.
[15] I. Krsul. *Software Vulnerability Analysis*. PhD thesis, Purdue University, 1998.
[16] D. Larochelle and D. Evans. Statically Detecting Likely Buffer Overflow Vulnerabilites. In *Proceedings of the 10th USENIX Security Symposium*, August 2001.
[17] R. Mahajan, S. M. Bellovin, S. Floyd, J. Ioannidis, V. Paxson, and S. Shenker. Controlling High-Bandwidth Aggregates in the Network (Extended Version). Technical report, ACIRI and AT&T Labs Research, July 2001.
[18] S. Mancoridis, T. Souder, Y.-F. Chen, J. Korn, and E. Gansner. Reportal: A web-based portal site for reverse engineering. In *Proceedings of the Working Conference on Reverse Engineering (WCRE'01)*, October 2001.
[19] J. Nazario. Source code scanners for better code. *Linux Journal*, January 2002. http://www.linuxjournal.com/article.php?sid=5673.
[20] T. Newsham. Format string attacks, September 2000. http://www.lava.net/ newsham/format-string-attacks.pdf.
[21] J. Q. Ning, A. Engberts, and W. Kozaczynski. Automated Support for Legacy Code Understanding. *Communications of the ACM*, 37(5):50–57, 1994.
[22] A. One. Smashing The Stack For Fun and Profit. *Phrack Magazine*, 7(49), November 1996.
[23] OpenBSD Homepage, http://www.openbsd.org/.
[24] N. Provos. Preventing privilege escalation. Technical Report 02-2, CITI, Center for Information Technology Integration, August 2002.
[25] D. Rosenblum and A. Wolf. Representing semantically analyzed c++ code with reprise. In *USENIX C++ Conference Proceedings*, pages 119–134, 1991.
[26] SecureSoftware. RATS http://www.securesoft.com/rats.php/.
[27] U. Shankar, K. Talwar, J. Foster, and D. Wagner. Detecting Format String Vulnerabilities with Type Qualifiers. In *Proceedings of the 10th USENIX Security Symposium*, August 2001.
[28] D. Wheeler. Flawfinder http://www.dwheeleer.com/flawfinder/.

System Name	FLF Density (%)	Longest Path	Total Functions	Lines of Code
bash	2.18	18	824	67,141
crond	2.50	3	120	3,646
elm	1.28	6	468	95,921
exim	1.82	10	549	61,210
fetchmail	3.13	11	351	24,201
gnupg	1.16	6	517	73,274
inn	.73	3	407	81,429
joe	4.04	26	644	20,639
lukemftp	3.04	17	558	7,995
lynx	1.00	12	1206	12,9420
mailx	3.33	10	300	9,351
man	3.98	9	226	23,581
minicom	3.91	10	256	11,571
mutt	1.32	15	1139	62,824
netkit-ftp	2.97	7	236	76,695
netkit-inetd	5.50	5	91	1,351
netkit-ping	2.38	1	42	835
netkit-tftpd	1.79	1	56	1,020
nmh	1.53	12	785	52,356
radius-client	4.43	7	158	15,872
screen	.94	4	424	24,796
sharutils	6.12	3	49	9,271
stunnel	3.52	8	227	3,820
sysklogd	7.58	10	132	6,115
tcpdump	.48	3	627	27,738
telnetd	7.14	8	227	16,480
webalizer	1.33	2	150	6,450
wu-ftpd	1.12	4	358	67,755
wwwoffle	2.85	11	386	44,498
zgv-1	3.32	9	271	8,607
zgv-2	2.58	7	271	8,607

Table 1. Thirty one open source systems make up the test suite [9]

Function Name	Function Coverage (%)	Number Vuln.	Number Found
micq	35.58	3	3
elm	39.82	1	1
dhcpd	18.75	2	1

Table 3. The results of the FLF Finder validation

unistd.h	stdio.h	getopt.h	stdlib.h	socket.h	
read	fscanf	getopt	getenv	recv	main
pread	scanf	getopt_long	canonicalize_file_name	recv_from	
pread64	vfscanf	getopt_long_only	realpath	recvmesg	
getcwd	vscanf	_getopt_internal	__secure_getenv		
getwd	fgetc				
get_current_dir_name	getc				
ttyname	getchar				
ttyname_r	getc_unlocked				
readlink	getchar_unlocked				
getlogin	fgetc_unlocked				
getlogin_r	getw				
getpass	fgets				
ctermid	fgets_unlocked				
	gets				
	getdelim				
	getline				
	fread				
	fread_unlocked				
	ctermid				

Table 2. The default Inputs provided by the FLF Finder