

PREPRINT

An Automated Approach for Identifying Potential Vulnerabilities in Software*

Anup K. Ghosh, Tom O'Connor, & Gary McGraw
Reliable Software Technologies Corporation
21515 Ridgetop Circle, #250,
Sterling, VA 20166
{aghosh,toconnor,gem}@rstcorp.com
<http://www.rstcorp.com>

Abstract

This paper presents results from analyzing the vulnerability of security-critical software applications to malicious threats and anomalous events using an automated fault injection analysis approach. The work is based on the well-understood premise that a large proportion of security violations result from errors in software source code and configuration. The methodology employs software fault injection to force anomalous program states during the execution of software and observes their corresponding effects on system security. If insecure behavior is detected, the perturbed location that resulted in the violation is isolated for further analysis and possibly retrofitting with fault-tolerant mechanisms.

1 Analyzing the behavior of software

It is now well understood that a vast majority of security intrusions are made possible by flaws in software. One need only look at the annals of Bugtraq for empirical evidence of this assertion.¹ To address this problem, computer security researchers and practitioners have created mature software engineering processes such as the TCSEC and the Systems Security Engineering Capability Maturity Model (SSECMM) [13] to improve the likelihood of producing more secure systems.

Another body of research focuses on producing secure protocols for transporting and accessing confidential data across insecure networks and in shared

databases. Process maturity models and formally verified protocols play a necessary and important role in developing secure systems. It is important to note, however, that even the most rigorous processes can produce poor quality software [17]. Likewise, even the most rigorously and formally analyzed protocol specification can be poorly implemented. In practice, market pressures tend to dominate the engineering and development of software, often at the expense of formal verification and even testing activities. This is especially true of commercial grade software for use by consumers. The result is a software product employed in security-critical applications (such as Internet clients and servers) whose behavioral attributes in relation to security are largely unknown.

The objective of the approach presented here is to provide the capability to analyze software programs for potential vulnerabilities that can be leveraged into security intrusions. Software developers currently have at their disposal a number of techniques for aiding in the development of quality software, including: program debugging, configuration management, memory leak detection, performance profiling, load testing, analysis of structural metrics, test case generation, and code coverage analysis. While all of these tools if properly used can result in higher quality software, none are specifically oriented toward analysis of security properties. The analysis technique presented in this paper is specifically oriented towards *identifying portions of software that if flawed can result in security violations*. This paper presents an automated approach and tool for simulating program flaws to determine their potential effect on system security. Results from applying the automated fault injection analysis on five common network service daemons are presented.

The approach described here focuses on the *behav-*

*This work is sponsored under the Defense Advanced Research Projects Agency (DARPA) Contract F30602-95-C-0282. THE VIEWS AND CONCLUSIONS CONTAINED IN THIS DOCUMENT ARE THOSE OF THE AUTHORS AND SHOULD NOT BE INTERPRETED AS REPRESENTING THE OFFICIAL POLICIES, EITHER EXPRESSED OR IMPLIED, OF THE DEFENSE ADVANCED RESEARCH PROJECTS AGENCY OR THE U.S. GOVERNMENT.

¹Bugtraq can be viewed on-line at <http://www.netSPACE.org/lsv-archive/bugtraq.html>

PREPRINT

ior of the software when executing under anomalous circumstances. The tool provides the developer or analyst with some assurance for how badly the software can behave when subjected to unexpected events. Until software has been exercised thoroughly under anomalous or malicious scenarios, the security of its behavior will remain unknown. Cheswick and Bellovin summed it up nicely in (page 7, [4]):

...any program, no matter how innocuous it seems, can harbor security holes. (Who would have guessed that on some machines integer divide exceptions could lead to system penetrations?) We thus have a firm belief that *everything is guilty until proven innocent*. [emphasis added]

Finding flaws in software that can be leveraged into full-scale security intrusions has been likened to finding a needle in a haystack. This approach is practiced regularly by computer “crackers” seeking to leverage software flaws into higher levels of privilege. The approach described in this paper does not purport to find the needle in the haystack, but rather to reduce the size of the haystack significantly by using automated software analysis methods. Knowing the locations of all flaws in any non-trivial piece of software requires omnipotence. Rather than searching for flaws in software, the approach simulates the effect of flaws in software by using data perturbation functions better known as fault injection functions. If the simulated flaw violates the security of the system, then the location where the flaw was introduced is identified for further investigation and possibly retrofitting of fault-tolerant mechanisms.

This approach has been successfully applied in other areas of critical software systems where flaws in software can result in critical losses [14, 15, 16]. The common ground between the analysis of safety-critical and security-critical software is that flaws in either type of application can lead to intolerable losses. The difference is in the usage of these systems (including malicious threats) and the types of failures that result in intolerable losses. The fault-injection-based approach is not a silver bullet solution to finding critical flaws as much as a means for identifying potential vulnerabilities in security-critical systems. Given this vulnerability information, software developers can harden these systems against potential attacks.

With a tool for automatically analyzing software applications under anomalous scenarios, software developers or analysts can enhance the exit criteria through which software is released into the market.

If the analysis does in fact reveal insecure behavior, then the software can be effectively “patched” prior to release. The tool will not automatically correct defects—only identify locations that if incorrectly coded could result in security hazards.

The tool’s capabilities are demonstrated with results from analyzing several security-critical software applications. The applications selected are all network servers whose source code is publicly available. The results from the analysis identify locations that are potentially vulnerable to attack. Before presenting analysis results, the fault-injection-based methodology on which it is based is explained.

2 Using fault injection for security analysis

Starting again from the premise that security intrusions are made possible by flaws in software, the next logical question is how to locate these potential “security-fatal” flaws. Because the locations of all flaws in software cannot be known in general, the approach in this work is to simulate flaws in as many locations as possible using automated fault injection techniques.

Consider that when a program, P , is misused, its behavior may differ from its normal behavior. Call the normal functionality of P , f . After being attacked, P will have a modified behavior with functionality, f' , where $f \neq f'$ for at least some input to P . As program P executes with modified functionality f' , its internal program states will almost certainly differ from the states that would have been produced when P was executing with functionality f . The approach employed here simulates program flaws by perturbing program states in f to f' . An Adaptive Vulnerability Analysis (AVA) algorithm is described here to predict *a priori* which states, when perturbed, will have an adverse impact on the secure behavior of P . Given this information, we can begin to home in on which locations would benefit most from code inspection or formal analysis and to discard locations from consideration that are least likely to affect the output behavior of P adversely.

2.1 Prior art

A number of techniques have evolved out of the software engineering discipline for analyzing software. In this section, the work of research groups from the University of Wisconsin and the University of California (Davis) in applying software analysis techniques for security assessment is summarized. Other pioneering work in this area was performed by researchers at the COAST Laboratory at Purdue University [12].

PREPRINT

A University of Wisconsin group using a tool called Fuzz subjected Unix utilities with random streams of input data. Miller et al. found that "... the failure rate of utilities on the commercial versions of UNIX ... tested (from Sun, IBM, SGI, DEC, and NeXT) ranged from 15-43%" [10, 11]. Most of these utilities failed because of errors in coding. The class of errors that caused the most failures were related to misuse of pointers and array subscripts. For example, incrementing the pointer past the end of an array was a common coding error. Using dangerous input functions, such as the `gets` call, turned out to be the second most common cause of errors that crashed system utilities. Besides being a cause of reliability errors, the `gets` call is notorious from the Morris Internet Worm incident [12]. The reason this call and other related input functions are dangerous is that they do not limit or check the length of the input they read. In the case of the Internet worm, supplying the `gets` call with over 512 bytes of data overruns the stack frame, thus enabling arbitrary input data to be executed [9]. This example emphasizes the dangers of using input functions and system calls that do not check or limit input lengths.

The work by Miller et al. managed to crash several system utilities, including `ftp` and `telnet`, by testing the bounds on input functions. In *Practical Unix & Internet Security*, Garfinkel and Spafford point out the ominous potential for security violations in standard software distributed by vendors, relative to the random black-box testing results from the Miller *et al.* study (pg 705,[9]):

What is somewhat frightening about the study is that the tests employed by Miller's group are among the least comprehensive known to testers — random, black-box testing. Different patterns of input could possibly cause more programs to fail. Inputs made under different environmental circumstances could also lead to abnormal behavior. Other testing methods could expose these problems where random testing, by its nature, would not.

Research by Bishop and Dilger at U.C. Davis has studied a class of race condition flaws called time-of-check-to-time-of-use (TOCTTOU) flaws [2]. Their research attempted to identify a coding error in which a program checks for a particular characteristic of an object, then takes some action while assuming that characteristic still holds—when in fact it does not. This type of problem is particularly critical in SUID-root

programs that attempt to verify that a user has access permissions to one file, then modify it. A cracker can exploit this flaw by creating a link from the file that has been granted access, *e.g.* `/usr/spool/mail/john`, to another file that requires higher privilege for access, *e.g.* `/etc/passwd`. If the cracker is clever enough, he or she can create the link *after* access has been granted and *before* the program accesses the file. This sleight of hand can fool the program into modifying a file it would not otherwise have permitted.

Bishop and Dilger's research has focused on a source-code-based technique for identifying patterns of code which could have this programming condition flaw. One of the limitations reported in their paper [2] for this technique is that the static analysis cannot determine if the environmental conditions necessary for this class of TOCTTOU binding flaws exist. Their conclusion is that a dynamic analyzer will be able to test the environment during execution and warn when an exploitable TOCTTOU binding flaw occurs.

Another U.C. Davis group is using property-based assertions and software testing techniques to verify security properties of software [8]. Similar to the work presented in this paper, these different research projects are applying techniques developed in other areas of software assurance (reliability, safety, testing) to the difficult problems of assuring security in computer systems.

The work presented in this paper is distinguished from the work described above in that it combines both white-box analysis with dynamic execution analysis. The University of Wisconsin group performed dynamic analysis using a very coarse measure of reliability—whether a program aborted or not. The work described here uses program source code to perturb data states dynamically and employs a fine-grained approach (via assertions) to measure security. The benefit of our approach is that access to source code provides the ability to isolate instances where flaws can result in security-critical failures.

The TOCTTOU research did in fact use program source code to identify a class of coding errors with security-critical consequences, however, it did not study the behavior of the program under analysis to determine whether the program could in fact violate security as a result of TOCTTOU binding flaws or other program coding errors. Studying the behavior of the code (via assertions of insecure behavior) rather than simply its structure can pinpoint sources of vulnerability that were not hypothesized *a priori* from well-known vulnerable coding techniques. Finally, the approach described in the rest of this paper is distin-

PREPRINT

guished from the property-based testing approach by the application of fault injection functions to force program states that would otherwise be difficult to obtain through standard testing techniques.

2.2 The AVA algorithm

Adaptive Vulnerability Analysis (AVA) is an algorithm for dynamically executing software, providing interesting input to a program, injecting anomalous events during execution, and determining if a security violation has occurred. Repeated execution of the algorithm can be used to estimate the vulnerability, or conversely, the security of the software application under analysis. The algorithm has been implemented in the Fault Injection Security Tool (FIST) described in Section 2.3. The algorithm is developed in [15] and summarized here.

Let P denote the program under analysis, x denote a program input value, Δ denote the set of all possible inputs to P , Q denote the normal usage probability distribution of Δ , \bar{Q} denote the inverse usage probability distribution, \hat{Q} denote a special input set, l denote a program location in P , and $PRED$ denote the violation predicate.

Algorithm 1:

1. For each location l in P that is appropriate, perform Steps 2-7.
2. Set **count** to 0.
3. Randomly select an input x or input sequence from Q , \bar{Q} , or \hat{Q} , and if P halts on x in a fixed period of time, find the corresponding set of data states created by x immediately after the execution of l . Call this set \mathcal{Z} .
4. Alter the sampled value of variable a found in \mathcal{Z} creating $\check{\mathcal{Z}}$, and execute the succeeding code on $\check{\mathcal{Z}}$. The manner by which a is altered will be representative of the threat class from T that is desired.
5. If the output from P satisfies $PRED$, increment **count**.
6. Repeat steps 3-5 n times, where n is the number of input test cases.
7. Divide **count** by n yielding $\hat{\psi}_{a|PQ}$, the vulnerability assessment, for each line l . This means that $1 - \hat{\psi}_{a|PQ}$ is the *security assessment* that was observed, given P , Q , and T .

The first two steps of the algorithm are very basic. AVA is a source-code based methodology, and hence instrumentation is placed between particular statements (which are called “locations” in the code). Either an automated system that implements the algorithm (if it is intelligent enough) or the user must tell the system which locations are relevant for fault-injection. Thus, the first step is to mark where injection is to occur. Next, a counter is initialized to zero, since the algorithm is used to observe how many security intrusions occur because of simulated threats that the prototype attempts for a particular location l .

Unlike most software metrics in use today, the AVA software assessment measure does not look at software structure. It looks at software behavior. The algorithm selects test cases (*i.e.*, program inputs) upon which the program will run in Step 3. The inputs can come from different testing schemes that are more likely to trigger a successful intrusion: rare events (with respect to the operational profile), known input sequences that are unusual or likely to be threatening, totally random inputs, or even the operational profile of the system. The fourth step performs the actual program state corruption or syntactic mutation of the code (*i.e.* this step is the fault injection step). Once the fault that is injected by Step 4 is executed during the analysis phase, the program has been altered in some way. Step 5 then determines if the problem forced during Step 4 causes the program to produce an output event that satisfies our definition for what constitutes as a security violation. If so, the counter is incremented by one. Step 5 is a non-trivial step. That is, it requires definition of a security policy for a program. The definition of a security policy is coded in the form of an assertion that states the program or its environment should not be in a particular state. Specific instances of security violations are given in Section 3. In general, security policies will vary by application.

AVA’s measure of information system security is not an *absolute* metric, such as mean-time-to-failure. Instead, it is a *relative* metric that allows a user to compare different versions of the same system, or to compare different (but similar) systems that have the same purpose.

2.3 Fault Injection Security Tool - FIST

AVA has been implemented in a working tool named the Fault Injection Security Tool (FIST). The tool automates white-box dynamic security analysis of software using program inputs, fault injection, and assertion monitoring of programs written in C and

PREPRINT

C++. A schematic diagram of FIST is shown in Figure 1. The fault injection engine provides a developer or analyst the ability to perturb program states randomly, append or truncate strings, attempt to overflow a buffer, and perform a number of other numerical fault injection functions. The security policy assertion component provides a developer or analyst the ability to determine if a security violation particular to the software application being analyzed has occurred.

2.3.1 Fault injection analysis

A fault injection engine has been implemented to support injection of anomalous states and malicious threats during the execution of the program. Fault injection placement is the process of instrumenting program source code to corrupt program states and introduce anomalous behavior while a program is executing. Fault injection functions are initially placed in every instrumentable location to permit analysis of software flaws anywhere in the program source code. The reasoning is that without knowledge of where actual flaws exist, simulating their effects everywhere can identify which locations are most likely to impact security. Program states are perturbed singly in each test run in order to assess the effect of a single flaw in a given location.

Fault injection is useful for simulating a variety of anomalous program behavior that would otherwise be very difficult, if not impossible, to simulate using standard testing [17]. The main use of fault injection functions for vulnerability analysis is to determine where potential weaknesses exist in a software program that can be leveraged into security violations. Fault injection also reveals the relative importance of variables, statements, or whole functions on the output (and security) of a program. For example, perturbing the result of a display function may have little or no effect on the output of a program. On the other hand, perturbing the result of a function that grabs user input, may well affect the output and perhaps even the security of the application. Finally, fault injection can be used to simulate malicious threats against a software application.

FIST includes numerous fault injection functions for all primitive data types ranging from simple Boolean state flips, to string mangling, to “stack smashing” buffer overflow functions. These functions include the ability to corrupt Booleans, characters, strings, integers, and doubles. The Boolean perturbation function applies a logical negation operation to an unperturbed value. The character perturbation function returns a character randomly selected

from the ASCII table. String perturbation functions provide the ability to truncate strings, concatenate a random string, concatenate a fixed string, generate a new string of random characters, and replace strings with a string randomly selected from a file. In addition to simple perturbation functions, FIST supports composition of fault injection functions from a combination of selected basic fault perturbations. For example, a user can append a fixed string with a random character fault perturbation, thus building a new fault perturbation.

The buffer overflow perturbation function overwrites the return address of the stack frame in which the buffer is allocated with the address of the buffer itself. By tracing the frame pointer back through the stack, the fault injection function is able to determine where to overwrite the return address. The opcodes for machine instructions are written into the buffer being perturbed. Eventually, the activation record containing the modified return address will be popped off the program stack and the program will jump to the machine instructions embedded by the fault injection function. These instructions will be executed as if they were a part of the normal operation of the program. Because different platforms implement different forms of program stacks, the buffer overflow fault injection functions are platform-dependent. Linux x86 and Sparc are the two platforms currently supported.

Unsafe languages such as C make buffer overflow attacks possible because of input functions such as `gets`, `strcat`, and `strcpy` that do not check the length of the buffer into which input is being copied. If the length of the input is greater than the length of the buffer into which it is being copied, then a buffer overflow can result. Safe programming practices that read in constrained input can prevent a vast majority of buffer overflow attacks. The reality of the situation, however, is that many privileged programs in the field today do not employ these safe programming practices. In addition, many of these programs are still coded in commercial software development labs in unsafe languages today. FIST detects the potential for buffer overflow attacks to be successful regardless of how the input is read. Searching for unsafe functions such as `strcat` and `strcpy` is one technique for detecting potential problems; however, it is insufficient by itself. Programmers often write their own dangerous input functions that read in unconstrained input. FIST attempts to overflow buffers regardless of whether the buffer is used in a known dangerous function or is used in a custom-written input function. Furthermore, FIST can overflow buffers for variables that are not

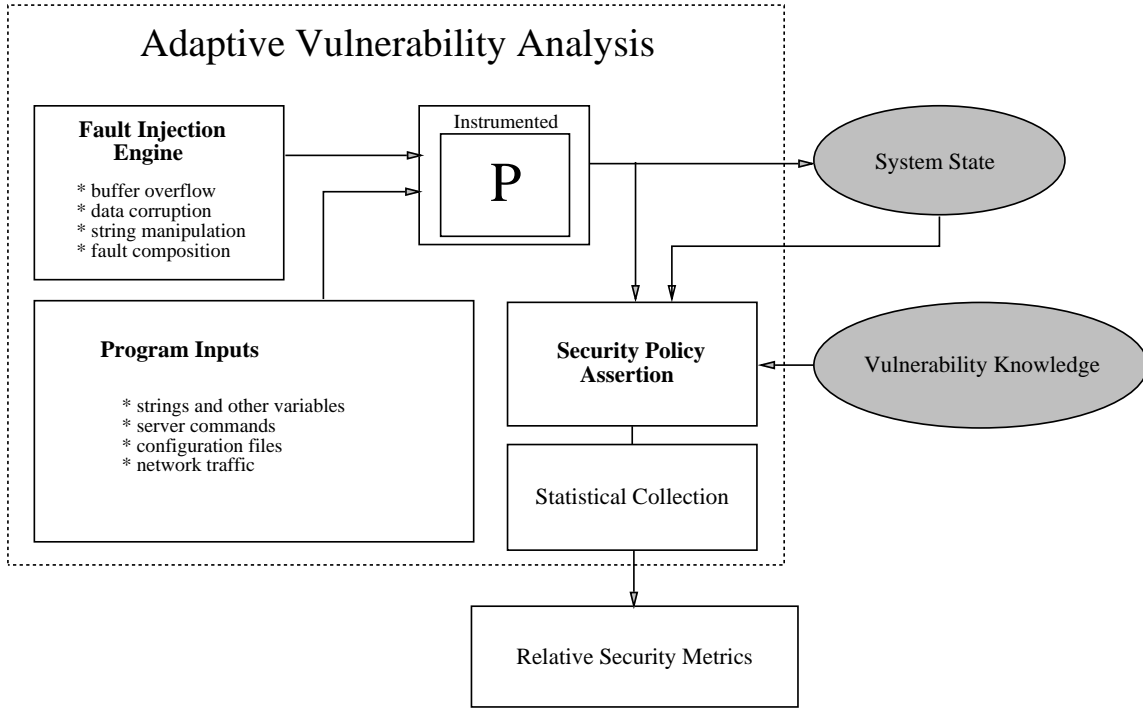


Figure 1: Overview of the Fault Injection Security Tool. A program, P , is instrumented with fault injection functions and assertions about its security policy (based on the vulnerability knowledge of the program). The program is exercised using program inputs. The security policy is dynamically evaluated using program and system states. If a security policy assertion is violated during the dynamic analysis, the specific input and fault injection function that triggered the violation is identified. Algorithm 1 is used to collect statistics about the vulnerability of the program to the perturbed states. One output from the analysis is the relative security metric $\hat{\psi}_{aPQ}$.

pushed on the stack. While this type of perturbation may not result in the execution of arbitrary program code, it may have side effects that compromise program security by corrupting other variables used for access/privilege decisions. If the perturbation results in a security policy breach, the programmer must either ensure that the vulnerable buffers cannot be overflowed from user input or use safe programming practices to ensure that the buffer overflow cannot occur. Once patched, FIST can be re-run to determine if the patch is resilient to attack.

As an alternative to the source-code-based analysis approach, StackGuard, a gcc compiler variant for Linux developed by the Oregon Graduate Institute, attempts to protect buffers from stack smashing attacks by aborting the program if the return address pushed on the stack is overwritten [7]. Stack Guard will not protect programs against all buffer overflow attacks, but can prevent stack smashing attacks from running arbitrary code embedded in user input. For

example, buffer overflow attacks that overwrite local variables that were never intended to be user changeable can result in security violations not prevented by StackGuard [1].

The Fuzz tool [11] can be used to overflow buffers, too, but with inconclusive results. Because the input is randomly generated, the vulnerability of the program to executing user-defined code cannot be assessed. FIST implements specific fault injection functions that determine the program’s vulnerability to specially-crafted buffer overflow attacks.

FIST integrates with the normal build process of the application under analysis. Any source file that is compiled using the FIST pre-processor at build time is instrumented. Libraries can be instrumented using FIST and then linked to applications, but only if the source code for the library is available. Uninstrumented libraries can also be linked to instrumented applications.

PREPRINT

2.3.2 Security policy monitoring

The security-policy-monitoring component of FIST allows users to specify what constitutes a security violation for the software application under analysis. Using assertions to encode this policy, the policy is monitored during the dynamic analysis to determine if it has been violated. The nature of violations will vary from application to application, and the types of violations the user will seek to detect will generally be dependent on both the input to the program and fault injection functions. As a result, the analyst must determine the security policy for the program being analyzed. A number of pre-defined assertion functions have been developed from which a user can specify the security violations for internal program variables, environment variables, and external system states.

Perhaps the broadest assertion function FIST provides allows the user to develop any expression in C to represent a violation assertion. This expression is evaluated during execution to determine if a violation has occurred. If the result of the expression is non-zero, then the violation is assumed to have occurred. This function has been developed for a sophisticated user who does not want to be constrained by the pre-packaged functions provided in the tool. Assertion functions are placed at locations in the source code during the instrumentation step. FIST also provides a mechanism for external assertion monitoring.

The external assertion monitor runs in parallel with the instrumented program and uses a subset of the built-in assertion functions. It is able to monitor files on the system, checking for modifications and/or accesses. For the buffer overflow functions, FIST checks for side effects of the `mycmd` program. The assertion is coded such that a file called `touch.out` should not be modified during the execution of the instrumented program. This assertion will be violated if the buffer overflow succeeds and the `mycmd` program is executed, which in turn will open `touch.out` and modify it. So when checking for buffer overflows, the security policy is simple: `touch.out` should never be modified.

3 Experimental Analysis

Analysis was performed on five different network services. Network daemons are interesting from a security standpoint because they provide services to untrusted users. Most network daemons allow connections from anywhere on the Internet, opening them up to attack from malicious users anywhere. Network daemons sometimes run with super-user, or `root`, privilege levels in order to bind to sockets on reserved ports, or to navigate the entire file system

without being denied access. Successfully exploiting a weakness in a daemon running with high privileges could allow the attacker complete access to the server. Therefore, it is imperative that network daemons be free from security-related flaws that could permit untrusted users access to high privilege accounts on the server.

The programs examined were NCSA `httpd` version 1.5.2.a, the Washington University `wu-ftpd` version 2.4, `kfingerd` version 0.07, the Samba daemon version 1.9.17p3, and `pop3d` version 1.005h. The source code for these programs is publicly available on the Internet. Samba, `httpd`, and `wu-ftpd` are popular programs and can be found running on many sites on the Internet. The analysis of those programs was performed on a Sparc machine running SunOS 4.1.3_U. The other programs, `pop3d` and `kfingerd`, are Linux programs found in public repositories for Linux source code on the Internet. The analysis of those programs was performed on a Linux 2.0.0 kernel. The programs were instrumented with both simple fault injection functions as well as the buffer overflow functions where applicable.

A summary of results from the analysis is shown in Table 1. The table shows the total number of instrumented locations together with the number of simple perturbations and buffer overflow perturbations that resulted in security violations. The last column shows the percentage of the functions in the source code that were executed as a result of the test cases employed. Higher coverage results may result in more potential security hazards flushed out through the analysis. The results should not be interpreted to mean that the locations identified in the analysis are necessarily exploitable, only that they require closer examination from the software's developers to determine if they can be exploited from input and whether fault-tolerant mechanisms should be employed.

In the following, case studies of each of the analyzed network services are presented. The case studies describe illustrative examples of fault injections that resulted in security violations.

3.1 Samba

Samba is a server message block (SMB) daemon for Unix. It allows a Unix file system and printers attached to Unix machines to be accessed by a machine running a Microsoft Windows operating system. Windows clients can treat Unix file systems, defined as "shares" in the daemon configuration files, as remote drives and mount them. Also included with the Samba software is a program called `smbclient` which provides an ftp-like interface allowing Unix client ac-

PREPRINT

Program	Instrumented Locations	Successful Simple Perturbations	Successful Buffer Overflows	Function Coverage
Samba v1.9.17p3	1264	12	15	45.5%
NCSA httpd v1.5.2a	463	27	3	40.14%
wu-ftp v2.4	476	11	3	58.62%
pop3d v1.005h	73	2	1	63.64%
kfingerd v0.07	146	12	5	38.1%

Table 1: Results from FIST analysis of network daemons.

cess to shared drives on Windows machines, or Samba shares on other Unix machines. The analysis concentrated only on file sharing and not on any printer sharing code. We used the `smbclient` program to drive the analysis. Motivating analysis of this daemon was a vulnerability made public in a previous version of Samba [6].

The test data used were fed to the `smbclient` and consisted of commands to navigate the shared file system and retrieve a file. The test case employed for buffer overflow analysis interacted with the server in a normal manner. When analysis was performed with simple perturbations, the test case attempted to connect to the server as a valid user using an incorrect password and then attempting to retrieve a file. The security policy for Samba stated that the target file should not be retrieved.

We instrumented 1264 locations in the code. Simple perturbation caused security violations at 12 locations, while the more complex buffer overflow perturbations resulted in 15 security violations. Vulnerabilities were identified in six modules: `util.c`, `username.c`, `reply.c`, `server.c`, `password.c`, and `loadparam.c`. The following code segment illustrates both a simple and a buffer overflow vulnerability found by FIST.

By perturbing the Boolean value evaluated on line 1344 of `password.c` in the `authorise_login[sic]` function, the daemon allowed the client access to the file system with an invalid password. This finding means that the logic in line 1344 had better be correct or else a security hazard may result. On line 1350, the tool found a buffer overflow violation for the `user` variable.

```

1344 if (!ok && GUEST_OK(snum))
1345     {
1346         fstring guestname;
1347         StrnCpy(guestname, lp_guestaccount
                (snum), sizeof(guestname)-1);
1348         if (Get_Pwnam(guestname, True))
1349     {
1350         strcpy(user, guestname);

```

```

1351     ok = True;
1352     DEBUG(3, ("ACCEPTED: guest account
                and guest ok\n"));
1353 }

```

Examining the buffer overflow more closely, the vulnerable buffer `user` is eventually resolved to type `pstring` — a fixed array of 1024 characters up the call stack from this point. The only way to get into this branch at line 1350, however, is if the call to `Get_Pwnam` evaluates to true. The string in `guestname` has to be long enough to overflow 1024 characters and then some.

3.2 NCSA httpd 1.5.2

This network daemon is meant to serve hyper-text documents which may or may not be password protected. This daemon was selected for analysis because it has been shown to be vulnerable to attacks in the past [5]. The test cases executed attempted to retrieve a password-protected HTML file through the daemon using a correct user name with an incorrect password. A test case was also run that supplied the correct password in order to cover additional code.

We instrumented 463 locations in the source code. Three buffer overflow vulnerabilities were found and 27 simple perturbations resulted in security violations. Six different modules of the `httpd` daemon were vulnerable to the fault injection functions: `http_access.c`, `http_auth.c`, `http_log.c`, `http_request.c`, `http_config.c`, and `util.c`. Next, an example of a simple perturbation that resulted in a security violation is presented.

In the module `http_access.c`, the `evaluate_access` function is vulnerable to fault injection. Within this function, fault perturbations at nine different locations resulted in the security policy being violated. As an example, the following statement when perturbed results in the access-protected page to be illegally retrieved:

```

329 num_dirs = count_dirs(path);

```

PREPRINT

This assignment statement calls the function `count_dirs`, that counts the number of directories in the path of the requested document. This function is necessary to determine if the document, image, or program requested by the Web client is situated in the document root anywhere under an access-controlled directory. For example, if the requested document is three sub-directories under the document root and the document root is access-controlled, then the document should not be fetched for the Web client without first authenticating the Web client. The variable `num_dirs`, which is assigned this value on line 329, is subsequently used in this function to implement the access control. The perturbation function corrupted this variable — which ultimately resulted in the security violation.

Because this variable determines the number of directories to search in the path of the requested document, if this number were changed to zero in the trivial case, then no directories will be searched for access control and the access-controlled page will be delivered without authentication. In the more general case, perturbing this number to any integer less than the number corresponding to the directory where access control is first instantiated will result in a security violation. One interpretation of this result is that if the function `count_dirs` is incorrectly coded such that it returns an incorrect value, or if `num_dirs` is assigned or read incorrectly, then the security of the application may be at risk.

Finally, it is worth noting that an analysis of `httpd 1.4.2` was performed prior to the analysis of `http 1.5.2`. One result from the analysis `httpd 1.4.2` was the discovery of a vulnerability in an input function that upon receiving special characters will increase the length of its input string with “escape” characters without checking if enough memory has been allocated. This vulnerability permits a Web client to crash a Web server by inserting these particular characters. The FIST analysis of `http 1.5.2` determined that this vulnerability had been patched, thus showing the ability to compare the relative security of two versions of one program.

3.3 Washington University `ftpd`

The Washington University `ftpd`, or `wu-ftpd`, is meant to be a replacement for the standard `ftp` daemon that is installed with most network servers. It provides extra configuration options and filters to control access to shared files. Our test case attempted to retrieve a file through the daemon using a series of commands that, under normal circumstances, would not allow the file to be retrieved. Out of the 476 locations instrumented, three buffer overflows and eleven

simple perturbations caused violations of the security policy. Modules found vulnerable were: `ftpd.c`, `extensions.c`, and `realpath.c`.

One buffer overflow violation found was in a reimplementation of the C library function `realpath`. This routine takes a pathname and resolves any symbolic links to return the full pathname. The C library implementation of `realpath` has a history of being prone to buffer overflows which may be why the authors chose to implement their own version. Unfortunately, the `wu-ftpd` implementation also has the potential to be vulnerable to buffer overflows according to the analysis. In short, the version of `realpath` uses string library routines that are prone to buffer overflows. One location found vulnerable is on line 126.

```
125  if (lstat(namebuf, &sbuf) == -1) {
126      strcpy(result, namebuf);
127      return (NULL);
128  }
```

The `if` conditional checks if the string in `namebuf` is a valid pathname. If the string in `namebuf` is an exploit string, this branch will be entered because the call to `lstat` will return `-1`, meaning that `namebuf` does not represent a valid pathname. At this point, a copy is performed without any bounds checking from `namebuf`, containing the exploit string, into `result`, the target buffer for overflow.

This function is invoked as part of the processing done for the server commands `MKD` (make directory), `RMD` (remove directory), `DELE` (delete), among others. If the user arguments to these commands flow unmodified from these commands to the `realpath` function, this violation could be exploited. A specialized client could be crafted to take advantage of this hole in the commands listed above if the exploit string is not modified elsewhere by the daemon.

3.4 `pop3d`

The test case run against the Post Office Protocol 3 daemon for buffer overflow analysis consisted of commands to authenticate a user, open their mailbox, list the contents, retrieve a message, and quit. When testing using simple perturbations, the test case attempted to open a user’s mailbox when supplying an invalid password. Some implementations of `pop3d` have been found vulnerable in the past [3]; the implementation used in this analysis is not known to contain the same flaws. Only one buffer overflow and two simple perturbations were detected out of the 73 locations we instrumented. Vulnerabilities were detected

PREPRINT

in `main.c` and `parse.c`. A buffer overflow vulnerability was detected on line 133 of `main.c`.

```
109 static void
110 initialize()
111 {
112     char buf[MAXHOSTNAMELEN+1];
.
.
.
129     gethostname(buf,MAXHOSTNAMELEN);
130     svr_hostname = malloc(strlen(buf)
                          + 1);
131     if (svr_hostname == NULL)
132         fail(FAIL_OUT_OF_MEMORY);
133     strcpy(svr_hostname,buf);
```

Line 133 performs a `strcpy` without any bounds checking. However, `buf` is populated by the `gethostname()` function. This means that the hostname of the computer this program is running on would have to be extremely long and be the character representation of machine instructions that would run an exploit script or program in order to exploit this potential vulnerability.

3.5 `kfingerd`

`kfingerd` is a replacement for the standard BSD finger daemon. It does not use the local finger program to service the requests; rather, it processes them internally. A user can optionally supply a configuration file in his or her own home directory to permit more information to be returned through the daemon if requested. The test case constructed for the analysis sent finger requests to a specific user whose configuration file, `.finger.rc`, specifies that project information should *not* be supplied to finger requests. The security policy coded for the analysis states that the `.project` file for that user should never be accessed. A total of 146 locations were instrumented of which twelve simple perturbations and five buffer overflow perturbations violated the security policy.

The buffer overflow vulnerabilities discovered in this source code involved the parsing of the configuration file. This file specifies what additional information to return through the finger daemon and which files to pull that information from. FIST found line 95 of `parse.c` vulnerable to the buffer overflow function.

```
87     char buf[256];
88     char p1[256],p2[256],p3[256],p4[256],
      p5[256];
89     a=fopen(".finger.rc","r");
90     if(a==NULL) return; /* If no rc file,
```

```
use defaults */
91     while(fgets(buf,256,a)!=NULL)
92     {
93         if((buf[0]!='#')&&(buf[0]!=' ')&&
            (strlen(buf)>3))
94         {
95             i=sscanf(buf,"%s %s %s %s %s",p1,p2,
                       p3,p4,p5);
```

FIST was able to successfully overflow buffers `p1` through `p5`. From the program input, variables `p1` through `p5` could be overflowed if the data in `buf` could be long enough. Closer examination reveals that `buf` is in fact limited by the `fgets` function—a secure implementation of the `gets` function. This analysis showed a potential vulnerability mitigated by a fault-tolerant input function.

4 Conclusions and future directions

This paper presented an approach for analyzing programs for potential vulnerabilities that have security-critical ramifications. The approach has been implemented in the Fault Injection Security Tool and applied to several security-critical network daemons. The results have demonstrated the ability of FIST to identify critical locations in program source code by simulating simple and complex fault conditions.

One limitation of the approach is that currently there is not an intelligent way to instrument a source file with buffer overflow functions. The source code must be examined by hand for candidate locations for buffer overflow functions. This process is being automated by searching a parse tree of the source code for definitions and uses of buffers that are pushed on the stack. The uses of these buffers will then be instrumented automatically with buffer instrumentation functions.

The second perceived limitation of the approach is that it will not find actual exploits themselves. FIST simulates the effects of flaws in source code to determine if these flaws have security-critical effects. However, merely finding these locations where the flaws were simulated does not imply that an actual flaw at that location exists. Rather, the analysis reveals only the potential for a security-critical flaw. Therefore, fault injection analysis is useful for developers of software to home in on locations that are most vulnerable to flaws that could have severe consequences. The analysis results have identified locations in source code that would normally be overlooked as security-critical during code inspection reviews. The case study developed here for the `httpd` program is one example.

A research task underway on this project is to be

PREPRINT

able to use the vulnerability information found by fault injection analysis and automatically determine if it is exploitable through user input. The approach begins with static data flow analysis to determine all paths from program input to the perturbed location that resulted in the violation. Fault injection functions will be re-instrumented in a second pass along these program slices to determine if the vulnerability can be exploited in program locations prior to the vulnerable location found in the first pass. The ultimate goal is to exploit the vulnerable location via an input function.

References

- [1] S. Bellovin. Re: Stackguard: Automatic protection from stack-smashing attack. Online. Bugtraq archives. See http://www.geek-girl.com/bugtraq/1997_4/0514.html, December 19 1997.
- [2] M. Bishop and M. Dilger. Checking for race conditions in file accesses. In *The USENIX Association, Computing Systems*, pages 131–152, Spring 1996.
- [3] CERT. CA-97.09: Vulnerability in IMAP and POP, April 1997.
- [4] W.R. Cheswick and S.M. Bellovin. *Firewalls and Internet Security*. Addison-Wesley, 1994.
- [5] CIAC. F-11: Unix NCSA httpd vulnerability, February 1995.
- [6] CIAC. H-110: Samba servers vulnerability, September 1997.
- [7] C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Symposium*, pages 63–78, San Antonio, TX, January 1998.
- [8] G. Fink and M. Bishop. Property-based testing: A new approach to testing for assurance. *ACM SIGSOFT Software Engineering Notes*, 22(4), July 1997.
- [9] S. Garfinkel and G. Spafford. *Practical Unix & Internet Security*. O’Reilly & Associates, Inc., 2nd edition, 1996.
- [10] B.P. Miller, L. Fredrikson, and B. So. An empirical study of the reliability of UNIX utilities. *Communications of the ACM*, 33(12):32–44, December 1990.
- [11] B.P. Miller, D. Koski, C.P. Lee, V. Maganty, R. Murthy, A. Natarajan, and J. Steidl. Fuzz revisited: A re-examination of the reliability of UNIX utilities and services. Technical report, University of Wisconsin, Computer Sciences Dept, November 1995.
- [12] E.H. Spafford. The Internet worm program: An analysis. *Computer Communications Review*, 19(1):17–57, January 1989.
- [13] SSECMM. Systems security engineering capability maturity model. Software Engineering Institute., 1995.
- [14] J. Voas, F. Charron, G. McGraw, K. Miller, and M. Friedman. Predicting how badly “good” software can behave. *IEEE Software*, 14(4):73–83, July 1997.
- [15] J. Voas, A. Ghosh, G. McGraw, F. Charron, and K. Miller. Defining an adaptive software security metric from a dynamic software failure tolerance measure. In *Proceedings of the 11th Annual Conference on Computer Assurance*, pages 250–263, June 1996.
- [16] J. Voas and K. Miller. Predicting software’s minimum-time-to-hazard and mean-time-to-hazard for rare input events. In *Proc. of the Int’l Symp. on Software Reliability Eng.*, pages 229–238, Toulouse, France, October 1995.
- [17] J.M. Voas and G. McGraw. *Software Fault Injection: Inoculating Programs Against Errors*. John Wiley and Sons, New York, 1998.