# Analyzing Programs for Vulnerability to Buffer Overrun Attacks*

Anup K. Ghosh and Tom O'Connor
Reliable Software Technologies Corporation
21515 Ridgetop Circle, #250, Sterling, VA 20166
phone: (703) 404-9293, fax: (703) 404-9295
email: {aghosh,toconnor}@rstcorp.com
http://www.rstcorp.com

## Abstract

*This paper presents an approach for analyzing security-critical software for vulnerability to buffer overrun attacks. In practice, buffer overruns are a commonly exploited attack against security-critical software systems. Buffer overrun attacks are made possible by flaws in designing and implementing software. This paper describes a software analysis tool that dynamically analyzes software source code to determine the potential to successfully overrun program buffers in order to execute arbitrary system commands. The methodology employs software fault injection to insert malicious strings into potentially vulnerable buffers during execution. If the buffer overrun attack is successful, arbitrary code can be executed at the whim of the attacker on the host system. Programs that are found to be vulnerable can be fortified to prevent buffer overrun attacks from being successful in the field. Three new algorithms for buffer overrun analysis are presented.*

## 1 Introduction

In practice, most external security violations are made possible by flaws in software. The empirical evidence of this assertion is stored in the archives of `Bugtraq`[1] — the on-line mailing list which discusses flawed software packages that have been exploited to violate security. Nearly everyday, new flaws discovered in software that can be exploited to violate security are posted to the list.

Other evidence is demonstrated by CERT advisories and vendor initiated bulletins released by CERT (`www.cert.org`). Since 1995, 25 out of 81 CERT advisories mention buffer overruns as the specific cause of the particular program vulnerability described by each advisory. Prior to 1995, the reporting in CERT advisories left out the causes of program vulnerabilities, making analysis an exercise in speculation.

The `Bugtraq` archive is an overt symptom of a larger problem in software development today. Currently, software is developed and released in relatively short cycles to meet market demands. Little security-based testing is performed prior to release, and little assurance is provided for end users as to the quality (*i.e.*, security) of the software that is purchased — be it an operating system or a desktop application such as a Web browser.

Once software is in the field, elite anonymous hackers (*i.e.*, experts) begin their informal analysis of the software, searching for vulnerabilities that can be exploited. If successful, the exploit is widely distributed among underground networks for other interested parties to exploit. Note, that it no longer takes an expert to exploit software, once the exploit is "in the wild". Once enough systems are exploited in the field to garner notice, on-line discussion groups such as `Bugtraq` and incident response groups such as CERT are notified of the vulnerability. In addition to alerting the group of security-aware end users and system maintainers, the incident response teams are responsible for notifying the software vendor of the problem if they have not already heard. Generally, the vendor is notified first so that a patch to the software can be released with the public announcement.

For companies that develop and release the software, the expense in adequately responding to security vulnerabilities is very high, not to mention the corresponding drop in consumer confidence which

*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.

[1] `Bugtraq` archives can be viewed on-line at http://www.netspace.org/lsv-archive/bugtraq.html

cannot be measured. Both Netscape and Microsoft experienced well-publicized security-related flaws in their Internet browsers in 1997. Developers of operating systems such as Sun Microsystems and Hewlett-Packard also spend considerable human resources tracking bugs that have the potential to be security flaws in their commercial operating systems. Because many commercial operating systems are now used in enterprise-critical or defense-related applications, the security of these systems is paramount for the commercial OS vendor's consumers. The costs also transfer to the end users. The time and expense involved for system administrators to patch, upgrade, and maintain the security of computer systems is also very high and growing with both new software and more sophisticated attacks.

Once patched, the cycle starts anew either with the patched software, the next release of the software, or new software. The cycle of developing, releasing, hacking, and patching software has resulted in more costly software development, losses due to security intrusions, and the high cost to maintain computer systems. The costs of security vulnerabilities can be reduced substantially by addressing the problem at its root: during software development.

This paper presents a technique and tool that analyzes software in source code form to detect one of the most significant vulnerabilities in software today: the buffer overrun. The tool performs dynamic white-box analysis for vulnerability to buffer overrun attacks. The tool is best suited for developers or analysts of software to determine whether program buffers are vulnerable to overrun attacks. If vulnerable buffers are found, the program code can be fortified using safe coding practices such as using constrained buffers or using safe input functions such as `strncpy` in C with appropriate arguments. Another alternative is to use stack smashing protection code such as those available through StackGuard and MemGuard [7]. Buffer overruns that do not smash the stack, but instead overwrite data in the same stack frame, cannot be protected against by StackGuard or MemGuard [5].

The goal of this research effort is to provide a tool in the hands of developers so that programs can be assessed for vulnerability to malicious attacks *prior* to releasing the software — effectively breaking the cycle of releasing software, then later patching it after the vulnerable software has been installed in end users' sites. While no technique can provide 100% assurance of security, this tool provides another weapon in the arsenal that software developers can use to develop high quality software in security-critical applications.

## 2 How the buffer overrun attack works

Buffer overrun attacks are made possible by program code that does not properly check the size of input data. When input is read into a buffer and the length of the input is not limited to the length of the buffer allocated in memory, it is possible to run past the buffer into critical portions of the stack frame. Overrunning the buffer results in writing to memory that is not reserved exclusively for the buffer. The consequences of overrunning a buffer can range from no discernible effect to an abortion of the program execution to execution of machine instructions contained in the input. If the unconstrained input can write over specific data in the program stack frame, then it may be possible to execute arbitrary program code included in the input.

The stack frame is the part of a process's address space that is used to keep track of local function data when a function is called. When calling a function, a new stack frame is created for the function that is called. The calling function "pushes" the address of the next instruction to be executed after returning from the called function on this stack. This address is known as the return instruction pointer. After the program finishes executing the called function, a pointer to the next instruction to be executed is "popped off" the stack. The value of the opcode pointed to be the instruction pointer is loaded and that instruction is executed.

By overwriting a buffer allocated on the stack, it is possible to change the instruction pointer to point to another address. In the case of many program crashes caused by buffer overruns, the instruction pointer is overwritten with random or garbage data that does not correspond to a legitimate instruction address. Upon returning from the called function, the processor attempts to execute an invalid instruction and an exception is generated. In this case, the program will normally abort execution, usually (but not always) without serious consequence on security.

On the other hand, if the input stream that overruns the buffer is carefully crafted, it is possible that the instruction pointer can be overwritten in a principled manner. That is, a specific address can be written into the instruction pointer so that when it is evaluated, the next instruction to be executed is located at an address in the stack frame. With the address pointing back into the stack, it is possible to execute any instructions embedded in the input stream that have been written into the stack.

The process to implement the buffer overrun is

```
void overflow(char *pointer){
   char small[100];

   strcpy(small,pointer);
}

void main(){
   char large[2000];
   int i;

   for (i=0;i<2000;i++)
     large[i] = 'X';
     overflow(large);
}
```

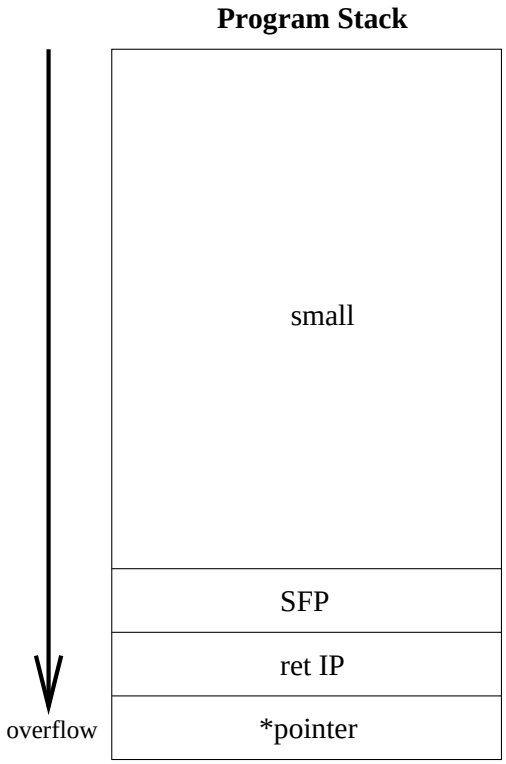


Figure 1: Smashing the Stack. The buffer overrun is made possible by copying a larger array over top of a smaller array. The `strcpy` function does not check the length of the target array that is being filled. As a result, the instruction pointer is overwritten which results in a program crash or possibly even execution of arbitrary code.

known in the hacker community as "smashing the stack". It is detailed in technical depth in "Smashing the Stack for Fun and Profit" [12]. The process for smashing the stack or overrunning buffers is illustrated in Figure 1. In the program's `main` function, an array variable `large` is defined to have length 2000. This array is filled with 2000 "X" characters. Next, the function `overflow` is called with a pointer to `large` passed as an argument. In `overflow`, a new array, `small`, is defined with length 100. The stack in the right side of Figure 1 shows how the memory is allocated for the `overflow` function. The variable `small` is allocated 100 characters. After `small`, memory is reserved for the stack frame pointer (SFP), the return instruction pointer (IP), and the pointer that was pushed on the stack when `overflow` is called. The `overflow` function simply copies the contents of the large variable array to the small variable array.

Unfortunately, the C function `strcpy` does not check the length of the variable it is copying before copying it to `small`. As a result, the 2000 characters are written over the 100 character long array. This means that after the first 100 Xs are copied, the rest of the 1900 characters will overwrite the SFP, the return IP, and even the pointer.

After the `overflow` function finishes executing, the processor will pop off the return IP address and execute the instruction located at that address. In this example, the address pointed to by the integer value of X is probably not an instruction, and as a result, this program will probably crash. However the `large` array could have been intelligently loaded with input that places a meaningful address at the return IP location. After returning from the `overflow` function, the next instruction that will execute will be the located at the address stored in the return IP location. If the attacker wrote this location with an address somewhere in the buffer that was overrun, then the attacker will be able to execute code of his or her own choice.

This technique is as effective as being able to access and modify the program source code, recompile it, and execute it without ever having access to the local source code. Smashing the stack is one of the primary attacks launched against SUID root programs, *i.e.*, programs that run as the super-user. The problem illustrated by Figure 1 was that a programming

error allowed a large buffer to overwrite a small buffer. In the figure it may seem fairly obvious that this would happen, but in many programs, the programmer is assuming that the user input is much smaller than what a malicious user may in fact be entering. The exploit was made possible in this case because the programmer used the `strcpy` function instead of something else that would have performed bounds checking.

## 3 Analyzing programs for buffer overrun vulnerability

Now that the mechanism by which buffer overruns are employed has been illustrated, the mechanism by which the vulnerability to buffer overruns is detected will be described. Given knowledge of how buffer overruns are accomplished in programs and heuristics about the types of programmer errors that lead to this kind of vulnerability, a sufficiently skilled programmer can manually inspect source code for potential vulnerabilities to buffer overruns. The tool described here is designed to assist the analyst in determining whether buffers are in fact vulnerable once identified.

Other software engineering analysis techniques have been applied to the problems of computer security. Pioneering work in analyzing buffer overruns has been performed by researchers at the COAST Laboratory at Purdue University [13]. In addition, research out of the University of Wisconsin has analyzed Unix utilities for reliability and robustness, with corresponding implications on security [10, 11]. A static software analysis technique employed by UC Davis researchers can analyze software for vulnerability to a class of race condition flaws called time-of-check-to-time-of-use (TOCTTOU) flaws [6]. Another UC Davis group is using property-based assertions and software testing techniques to verify security properties of software [8]. These different research projects are employing techniques developed in other areas of software assurance (reliability, safety, testing) to the difficult problems in assuring security in computer systems. This paper presents the use of fault injection analysis for analyzing vulnerability to buffer overrun attacks. Fault injection analysis has been used in other areas of software assurance for analyzing the behavior of programs under anomalous circumstances such as unexpected input from users or other software components [14].

The Fault Injection Security Tool (FIST) uses fault injection analysis to observe the effect on system security of faults injected during the execution of a program under analysis [9]. FIST employs several different fault injection functions that simulate the effects of programmer errors and malicious threats against programs. One such function allows the instrumentation of a buffer overrun function on buffers identified by the analyst. If the buffer overrun function is successful, it will modify a file that exists on the file system. If the buffer overrun is unsuccessful, then the file system will remain unmodified.

Any number of buffers may be instrumented with the buffer overrun function, although only one fault injection is executed per experiment run in order to assess the effect of the buffer overrun at a given location.

FIST is able to automatically determine where in the stack a given buffer lives by reading the value of the frame pointer and then following it up the stack. Once it finds the proper stack frame, it locates the return address of that stack frame. It then overwrites just that value with the location of the buffer that is being "overrun."

The opcodes for machine instructions are written into the buffer that is perturbed through fault injection. Eventually, 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 have different opcode values and use their stacks differently, the buffer overrun fault injection functions are platform-dependent. Intel x86 and Sparc are the two platforms currently supported by FIST.

The machine instructions in a buffer overrun exploit normally attempt to create an interactive shell process with which to compromise the security of the system. With the correct amount of assembly level programming skills for the target machine, the exploit instructions can be crafted to execute *any command* at the privilege level of the process being attacked. The FIST exploit instructions attempt to run a program called `mycmd`, which includes side effects that are detectable by an external monitor.

## 4 Case study: analyzing `wu-ftpd` 2.4

The Washington University `ftpd`, or `wu-ftpd`, is meant to be a replacement for the standard ftp daemon that comes with most operating systems. It provides extra configuration options and filters to control access to shared files. The `wu-ftpd` version 2.4 is a fairly mature network daemon available for most platforms. 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, unless specifically configured to reject certain sites. This opens

them up to attack from anywhere. Network daemons usually run with a higher privilege level than normal users to run on privileged ports; so successfully exploiting a weakness in a running daemon could allow the attacker complete access to the server.

Due to its high visibility, `wu-ftpd` has come under a good deal of scrutiny in the past which has exposed a number of security violations [1, 3, 2, 4]. Subsequent releases have patched known flaws. Prior problems have dealt with gaining root shell access. Some dealt with how to exploit a poorly configured server, allowing unauthorized access through anonymous ftp usage. Most security breaches now have configuration options as a patch. The case study here describes an analysis of potential buffer overrun vulnerabilities.

The analysis instrumented 6 out of 18 modules with buffer overrun instrumentation functions in forty-four (44) different locations. These locations were identified by manual inspection. Out of the 44 locations that were instrumented three (3) overrun violations were found by FIST. That is, FIST was able to execute arbitrary code in three of the instrumented locations. The nature of these potential vulnerabilities are described next.

Violations were detected in a re-implementation 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` can be leveraged in buffer overrun attacks which may be why the authors chose to implement their own version. Unfortunately, the `wu-ftpd` implementation is also prone to buffer overruns according to the analysis. In short, the version of `realpath` uses string library routines that are prone to buffer overruns. The two buffer overruns occur at line 122 and line 159:

```
44   char *
45   realpath(char *pathname, char *result)
46   {
.
.
.
122       strcat(namebuf, where);
```

and

```
159       strcpy(result, workpath);
160       return (result);
```

Both of these locations in the code are reached when the input to `realpath` is properly formatted, *i.e.* a valid pathname string. For an exploit to succeed, the input needs to be a string containing the character representation of machine opcodes. The input to this function specified by the input parameter `pathname` is parsed for characters like forward slash, and combinations of characters such as two periods followed by a forward slash. If the machine instructions resolve the ASCII sequences previously listed, the instructions string could be modified or truncated, ruining the exploit.

The violation on line 159 does not appear to be exploitable because it is reached only when `realpath` is able to evaluate a valid pathname. An exploit string would most likely be caught by an escape condition early in the loop because it would not evaluate to a valid pathname. However, the escape condition, the one that checks to see if the incoming string is actually a valid pathname, is vulnerable to a buffer overrun as well.

```
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 legal or valid pathname. At this point, a copy is performed without any bounds checking from `namebuf` to `result`, where `namebuf` contains the exploit string.

Checking to see if this vulnerable point can be exploited by a user can currently only be done by hand. The `realpath` function is invoked as part of the processing done for the server commands `MKD` (make directory), `RMD` (remove directory), `DELE` (delete), and a few others. If the user arguments to these commands can flow untouched and unmodified from these commands to the `realpath` function, this violation is exploitable. Someone could craft a very simple and specialized client program to take advantage of this hole in the commands listed above.

Investigation of the latter vulnerability found that the buffer overrun cannot be exploited because of string truncation and manipulation done in the lexer. Among other reasons, the lexer has an internal buffer shorter than what is needed to overrun the buffer in `realpath`. Any input line longer than the lexer's buffer is split and treated as two different commands.

In summary, this case study showed that out of 44 candidate locations instrumented with buffer overrun functions, 3 locations could in fact be exploited by FIST. However, upon further examination of these locations, it was determined that the likelihood of *user*

input successfully reaching vulnerable buffers unmangled is very low. The tool in this case served to provide additional assurance that for the sections of `wu-ftpd` analyzed with FIST, the likelihood of a buffer overrun attack succeeding is fairly small.

# 5 Improving buffer overrun analysis

Two significant improvements to the buffer overrun analysis are planned and described here. The first improvement will be automating to some extent the selection of buffers for instrumentation. Currently, in order to analyze programs for vulnerability to buffer overrun attacks, the analyst must manually inspect the source code for viable candidates to instrument. Once found, the analyst instruments these locations with the buffer overrun function provided by FIST. If the buffer overrun is successful, FIST will be able to execute arbitrary system commands at the privilege level of the program being analyzed. Performing manual inspection requires a degree of knowledge of how buffer overrun exploits work and experience in using buffer overrun attacks. An algorithm is developed in the next subsection that provides heuristics for parsing program source code for viable locations to instrument. The rules may be adapted as more experimental results dictate new rules.

The second improvement will be to modify the buffer overrun function to more accurately mimic a buffer overrun exploit as sent by a malicious user. The goal is to provide fewer false positives in the analysis, but still provide identification of vulnerable buffers. The algorithm for dynamically creating buffer overrun exploits is presented in Section 5.2.

Before describing each of the two improvements, it is useful to present the overall algorithm for analyzing programs for vulnerability to buffer overrun attacks. The algorithm presented next employs the algorithms presented in the following two subsections.

For notational purposes, the program being analyzed is called $P$, an input to $P$ is called $x$, each location $l$ corresponds to a statement in the program that can be instrumented with a buffer overrun function, the set $\mathcal{I}$ is the set of all candidate locations for instrumenting with buffer overrun functions, and $PRED$ is an assertion that the buffer overrun attack has succeeded. Currently $PRED$ is implemented as a system monitor that determines if a file has been modified on the file system which will occur if the buffer overrun succeeds.

**Algorithm 1**: Analysis of Vulnerability to Buffer Overruns

1. Parse the source code for $P$ to identify the set of candidate locations $\mathcal{I}$ using Algorithm 2.
2. Instrument locations $\mathcal{I}$ with buffer overrun functions.
3. Set **count** to 0.
4. Select an input $x$ that executes at least one $l$ in $\mathcal{I}$.
5. Run $x$ on $P$. If $P$ halts on $x$ in a fixed period of time, find the corresponding set of data states created by $x$ immediately before the execution of $l$. Call this set $\mathcal{Z}$.
6. Alter $\mathcal{Z}$ according to Algorithm 3 and call the resulting set of data states $\breve{\mathcal{Z}}$.
7. Execute the succeeding code in $P$ on $\breve{\mathcal{Z}}$.
8. If $P$ satisfies $PRED$, increment **count** and add the $l$ to $\breve{\mathcal{I}}$ where $\breve{\mathcal{Z}}$ resulted $PRED$ being satisfied.
9. Repeat steps 4 - 8 until all $l$ in $\mathcal{I}$ have been exercised.
10. **count** is the number of times the the buffer overrun succeeded and $\breve{\mathcal{I}}$ is the set of locations that are vulnerable to buffer overrun attacks.

Algorithm 1 is the methodology for analyzing programs for vulnerability to buffer overrun attacks. This algorithm will determine how many buffers are vulnerable to buffer overrun attacks and the locations of these buffers. Next, the algorithms for automatically instrumenting programs with buffer overrun functions and for dynamically creating the buffer overrun string are described.

## 5.1 Automatic instrumentation of buffers

Currently, the only recourse an analyst has to determine buffer overrun vulnerability is to perform manual inspection of source code to identify potential candidates. Using FIST, these locations can be instrumented with the buffer overrun functions to determine potential vulnerability. In order to automate the inspection process, an algorithm that employs inspection heuristics to source code is presented next.

**Algorithm 2**: Identifying Candidate Locations

Parse the program source code and identify all buffers that are allocated on the stack, *i.e.* have function scope. Apply the following heuristics:

1. Add location $l$ in $P$ to the set $\mathcal{I}$ if $l$ references one of the buffers identified at parse time using one of the following C functions that does not perform bounds checking: `strcpy, strcat, sprintf, gets, fscanf, scanf, sscanf, realpath`.
2. Add location $l$ in $P$ to the set $\mathcal{I}$ if $l$ references one of the buffers identified at parse time using one of the following C functions that does perform bounds checking, but could be improperly used in the context of the program: `memcpy, memmove, bcopy, strncpy, strncat`.
3. Add location $l$ in $P$ to the set $\mathcal{I}$ if $l$ it operates on a buffer identified at parse time using a user-defined function that reads or copies input.
4. Remove any $l$ in $\mathcal{I}$ that has a string literal as the source argument. This usage of vulnerable functions cannot be exploited.

The algorithm simply employs heuristics for adding or removing locations in $P$ to the set of candidate locations to be instrumented for buffer overrun analysis. These rules may be augmented or refined with more experimentation.

### 5.2 Creating the exploit

The buffer overrun analysis method described in Section 3 used knowledge of the frame pointer value to determine where the buffer resides in memory. The buffer itself is written with simply the opcodes for the exploit code rather than the entire string that would normally be overwritten in a standard buffer overrun attack. Finally, in order for the code in the buffer to be executed, FIST overwrites the return address with the value of the location where the buffer resides.

While this process is fairly sophisticated and completely automated for specific processor architectures, it does not faithfully mimic the kinds of buffer overrun attacks that occur in practice. Generally, in order to exploit a buffer vulnerability, the attacker must be able to send a very large string that can effectively overrun the buffer and overwrite the stack. The return instruction pointer is replaced with an address that points to a location of the first instruction in the exploit code in the stack.

Because FIST has complete access to the program state, it can use fault injection functions to change any value in the program state during execution that may not be otherwise possible from user input. As a result, FIST can successfully exploit buffers that in practice would not be otherwise possible. The case study in Section 4 is an example where FIST successfully exploited a potential vulnerability that in practice appears to be difficult to exploit from user input.

In order to reduce the number of false positives in analyzing buffer overrun vulnerability, a new technique for injecting buffer overruns that employs large strings similar to buffer overrun attacks is developed. Next, an algorithm for dynamically creating a buffer overrun for an instrumented location is presented.

**Algorithm 3**: Creating the Buffer Overrun

For each $l$ in $\mathcal{I}$ executed in Algorithm 1:

1. Record address of target buffer to be overrun: $BA$.
2. Using the stack frame pointer (SFP) as a reference point, calculate the distance from $BA$ to the return instruction pointer ($IP$) for the stack frame in which the buffer lives. The length of exploit string ($|ES|$) is equal to $BA$ - $IP$.
3. Create an exploit string that contains the opcodes for system exploit commands padded by copies of the return address of the target buffer ($BA$). Because the opcodes are fixed and their length is static for each processor architecture, the amount of padding necessary is equal to $|ES|$ - length of opcodes.
4. Write exploit string in source buffer before $l$ executes. (Step 6 in Algorithm 1)
5. Execute $l$ to overrun target buffer in location $l$. (Step 7 in Algorithm 1)

The algorithm first captures the address of the target buffer that is to be overrun. Then it follows the

SFP up the stack to the stack frame in which the buffer resides. The distance from the target buffer address ($BA$) to the return instruction pointer ($IP$) for that stack frame is computed. This distance is equal to the length of the exploit string that will be loaded in the source buffer by the buffer overrun function. First, the opcodes are written into the string, then the remaining bytes of the string are padded with copies of the target buffer address ($BA$). When the source buffer is copied onto the target buffer, the portion of the string that overruns the stack will be the target buffer address. If the fault injection is successful, the return $IP$ will be overwritten with the $BA$ so that the next instruction popped off the stack when the function returns will be the first instruction in the exploit string.

If this exploit string executes, the system commands will modify a file in the local file system. A monitor will detect this event and increment the counter as described in Algorithm 1.

## 6 Conclusions

This paper presents an approach for analyzing programs for vulnerability to one of the most pervasive security problems today: buffer overrun attacks. A white-box fault injection analysis tool, FIST, has been implemented that implements a buffer overrun function for user-selected buffers in program source code. The tool replaces the value in the return Instruction Pointer in the appropriate stack frame with the address of the target buffer. The opcodes for system exploit commands are written to the target buffer. In cases where the buffer overrun is able to successfully write a return address in the return instruction pointer on the stack, the fault injection analysis will be successful in executing arbitrary commands on the system.

Results from applying this analysis method to the `wu-ftpd` version 2.4 are presented. The results demonstrated that out of 44 buffers instrumented (by inspection), 3 of the buffers showed the potential to be vulnerable to buffer overrun attacks. Further analysis showed that reaching these buffers with an unmangled exploit string from input was difficult and unlikely.

To further improve the analysis technique, three algorithms are presented that describe an approach for automatically instrumenting candidate buffers, executing a buffer overrun function that more closely mimics buffer overrun attacks in practice, and recording the success or failure for each candidate location that was instrumented.

Future research involves implementation of the improved algorithms presented here, further experimental analysis for security-critical programs, and the development of new algorithms for tracing vulnerabilities back to program input.

## References


[1] CERT
Advisory. wuarchive ftpd vulnerability, April 9 1993. CERT Advisory CA-93:06. Available online: ftp://ftp.cert.org/pub/cert_advisories/.

[2] CERT Advisory. ftpd vulnerabilities, April 14 1994. CERT Advisory CA-94:08. Available online: ftp://ftp.cert.org/pub/cert_advisories/.

[3] CERT
Advisory. wuarchive ftpd trojan horse, April 6 1994. CERT Advisory CA-94:07. Available online: ftp://ftp.cert.org/pub/cert_advisories/.

[4] CERT Advisory. wu-ftpd misconfiguration vulnerability, November 30 1995. CERT Advisory CA-95:16. Available online: ftp://ftp.cert.org/pub/cert_advisories/.

[5] S. Bellovin. Re: Stackguard: Automatic protection from stack-smashing attack. Online. Bugtraq archives. See http://www.geekgirl.com/bugtraq/1997_4/0514.html, December 19 1997.

[6] M. Bishop and M. Dilger. Checking for race conditions in file accesses. In *The USENIX Association, Computing Systems*, pages 131–152, Spring 1996.

[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] A.K. Ghosh, T. O'Connor, and G. McGraw. An automated approach for identifying potential vulnerabilities in software. In *Proceedings of the 1998 IEEE Symposium on Security and Privacy*, Oakland, CA, May 3-6 1998.

[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 revisted: A re-examination of the reliability of UNIX utilities and services. Technical report, University of Wisconsin, Computer Sciences Dept, November 1995.

[12] Aleph One. Smashing the stack for fun and profit. Online. Phrack Online. Volume 7, Issue 49, File 14 of 16. Available: www.fc.net/phrack/, November 9 1996.

[13] E.H. Spafford. The Internet worm program: An analysis. *Computer Communications Review*, 19(1):17–57, January 1989.

[14] J.M. Voas and G. McGraw. *Software Fault Injection: Inoculating Programs Against Errors*. John Wiley and Sons, New York, 1998.