Defense against buffer overflow attack by software design diversity

Kunal Metkar

University of Nevada, Las Vegas

Follow this and additional works at: https://digitalscholarship.unlv.edu/rtds

Repository Citation
Metkar, Kunal, "Defense against buffer overflow attack by software design diversity" (2007). UNLV Retrospective Theses & Dissertations. 2249.
https://digitalscholarship.unlv.edu/rtds/2249

This Thesis is brought to you for free and open access by Digital Scholarship@UNLV. It has been accepted for inclusion in UNLV Retrospective Theses & Dissertations by an authorized administrator of Digital Scholarship@UNLV. For more information, please contact digitalscholarship@unlv.edu.
DEFENSE AGAINST BUFFER OVERFLOW ATTACK BY SOFTWARE DESIGN DIVERSITY

by

Kunal Metkar

Bachelor of Computer Engineering
Pune University, India
2004

A thesis submitted in partial fulfillment of the requirements for the

Master of Science Degree in Computer Science
School of Computer Science
Howard R. Hughes College of Engineering

Graduate College
University of Nevada, Las Vegas
December 2007
Thesis Approval
The Graduate College
University of Nevada, Las Vegas

NOVEMBER 13TH, 2007

The Thesis prepared by

KUNAL METKAR

Entitled

DEFENSE AGAINST BUFFER OVERFLOW ATTACK BY SOFTWARE DESIGN DIVERSITY

is approved in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

Examination Committee Chair

Dean of the Graduate College

Examination Committee Member

Examination Committee Member

Graduate College Faculty Representative
ABSTRACT

Defense against Buffer Overflow Attack by Software Design Diversity

by

Kunal Metkar

Dr. Yoohwan Kim, Examination Committee Chair
Assistant Professor of Computer Science
University of Nevada, Las Vegas

A buffer overflow occurs during program execution when a fixed-size buffer has had too much data copied into it. This causes the data to overwrite into adjacent memory locations, and, depending on what is stored there, the behavior of the program itself might be affected.

Attackers can select the value to place in the location in order to redirect execution to the location of their choice. If it contains machine code, the attacker causes the program to execute any arbitrary set of instructions—essentially taking control of the process. Successfully modifying the function return address allows the attacker to execute instructions with the same privileges as that of the attacked program.

In this thesis, we propose to design software with multiple variants of the modules/functions. It can provide strong defense against the buffer overflow attack. A way can be provided to select a particular variant (implementation) of the module randomly when software is executed. This proves to be useful when an attacker designs the attack for a particular variant/implementation which may not be chosen in the random selection process during execution. It would be much difficult for the attacker to design
an attack because of the different memory (stack-frame) layout the software could have
every time it is executed.
# TABLE OF CONTENTS

ABSTRACT ................................................................................................................................. iii

LIST OF FIGURES ................................................................................................................... vii

ACKNOWLEDGEMENTS ........................................................................................................ viii

CHAPTER 1 INTRODUCTION ........................................................................................................ 1
  1.1 Buffer and Buffer Overflow ............................................................................................. 1
  1.2 Buffer Overflow Attacks .............................................................................................. 2

CHAPTER 2 LITERATURE REVIEW OF PREVENTION STRATEGIES AGAINST
BUFFER OVERFLOW ATTACKS ............................................................................................. 8
  2.1 Vulnerability Prevention Techniques ............................................................................. 8
    2.1.1 Static Analysis ....................................................................................................... 8
      2.1.1.1 Source Code Auditing .................................................................................. 8
      2.1.1.2 Lexical Technique: pscan .......................................................................... 9
      2.1.1.3 Proper Handling of Type Qualifiers ......................................................... 10
      2.1.1.4 Use of Safe Programming Languages ..................................................... 10
    2.1.2 Compiler Modifications ....................................................................................... 11
      2.1.2.1 StackGuard ............................................................................................... 11
      2.1.2.2 ProPolice Compiler ................................................................................... 11
      2.1.2.3 StackShield ............................................................................................... 12
      2.1.2.4 RAD ........................................................................................................ 12
    2.1.3 Operating System Modifications ............................................................................ 13
      2.1.3.1 Libsafe ...................................................................................................... 13
      2.1.3.2 OpenBSD .................................................................................................. 13
      2.1.3.3 'Proof-carrying code' Technique ............................................................ 14
      2.1.3.4 'Program shepherding' Technique ......................................................... 14
    2.1.4 Hardware Modification ........................................................................................... 15
      2.1.4.1 SmashGuard ............................................................................................ 15
      2.1.4.2 Split Stack and Secure Return Address Stack (SAS) ............................. 16
  2.2 Limitations of Prevention Strategies ............................................................................. 18
    2.2.1 Current Limitations of Advanced Static Analysis for C and C++ ....................... 18
      2.2.1.1 C's Liberal Nature Makes the Language Poorly Suited for Static
              Analysis ........................................................................................................ 18
      2.2.1.2 The Added Complexities of C++ Make It Very Difficult to Analyze ....... 18
      2.2.1.3 Static Analysis in a Multi-threaded Environment Is Difficult ................. 18
      2.2.1.4 Better Static Analysis Is Less Efficient .................................................. 18
    2.2.2 Limitations with Libsafe Library ............................................................................ 19
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1</td>
<td>Memory Layout of a Stack Frame after a Function Has Been Called</td>
<td>2</td>
</tr>
<tr>
<td>Figure 2.a</td>
<td>Address Space before the Attack</td>
<td>4</td>
</tr>
<tr>
<td>Figure 2.b</td>
<td>After Injecting the Attack Code</td>
<td>5</td>
</tr>
<tr>
<td>Figure 2.c</td>
<td>Executing Attack Code</td>
<td>6</td>
</tr>
<tr>
<td>Figure 3</td>
<td>Number of Reported CERT Security Advisories and the Number Attributable to Buffer Overflow</td>
<td>7</td>
</tr>
<tr>
<td>Figure 4</td>
<td>N-Variant System Framework</td>
<td>22</td>
</tr>
<tr>
<td>Figure 5</td>
<td>Control Flow through Different Variants of the Module</td>
<td>29</td>
</tr>
<tr>
<td>Figure 6</td>
<td>Stack Frames for Variants of a Module</td>
<td>30</td>
</tr>
<tr>
<td>Figure 7</td>
<td>Stack Address Space (Memory Layout) When One Variant Is Replaced by Another</td>
<td>31</td>
</tr>
<tr>
<td>Figure 8</td>
<td>Structure for Strategy Design Pattern</td>
<td>32</td>
</tr>
<tr>
<td>Figure 9</td>
<td>Control Flow through Different Variants of the Module with Strategy Selector</td>
<td>34</td>
</tr>
<tr>
<td>Figure 10</td>
<td>‘Server’ In Listen Mode</td>
<td>41</td>
</tr>
<tr>
<td>Figure 11</td>
<td>‘Client’ Sending Large Message to Server</td>
<td>41</td>
</tr>
<tr>
<td>Figure 12</td>
<td>Server Hacked by Executing Netcat Utility</td>
<td>42</td>
</tr>
<tr>
<td>Figure 13</td>
<td>Client Takes Control of Server (Attacked) System</td>
<td>42</td>
</tr>
<tr>
<td>Figure 14</td>
<td>When Attacked Program Is Closed, Attacker Loses the Control</td>
<td>43</td>
</tr>
<tr>
<td>Figure 15</td>
<td>Server Crashes When Buffer Overflow Attack Fails</td>
<td>43</td>
</tr>
<tr>
<td>Figure 16</td>
<td>Failed Buffer Overflow Attack When Client Is Unable to Take Control of the System</td>
<td>44</td>
</tr>
</tbody>
</table>
ACKNOWLEDGEMENTS

I would like to dedicate this thesis to my parents and brothers without whose moral support I would not have accomplished the task. I am deeply indebted to Dr. Yoohwan Kim for trusting my abilities and considering me for mentorship. He serves as my role model with his Honesty and Dedication. I am also thankful to Dr. Thomas Nartker, Dr. Kazem Taghva and Dr. Venkatesan Muthukumar for being part of my Advisory Committee.

I would take this opportunity to thank all my friends especially Kirankumar Jayakumar and Swamynathan Sambamurthy for their encouragement and valuable suggestions. A special thanks to Ashwini Raina who always has a word of advice for me in hour of need.
CHAPTER 1

INTRODUCTION

1.1 Buffer and Buffer Overflow

A buffer is simply a contiguous block of computer memory that holds multiple instances of the same data type. C programmers normally associate ‘arrays’ with the word buffer. Most commonly, character arrays. Arrays, like all variables in C, can be declared either static, i.e. allocated at load time on the data segment, or dynamic i.e. allocated at run time on the stack. To overflow is to flow or fill over the bounds. Here, we are referring overflow of dynamic buffers, otherwise known as stack-based buffer overflows.

A buffer overflow occurs during program execution when a fixed-size buffer has had too much data copied into it. This causes the data to overwrite into adjacent memory locations, and, depending on what is stored there, the behavior of the program itself might be affected [1].

During program execution, when a function is called, a “stack frame” is allocated for a function storing function arguments, return address, previous frame pointer, and local variables. Figure 1 shows the ‘stack frame’ i.e. the memory layout when the function is called. We can see three logical areas of memory corresponding to program code - instructions, data and stack [1].
1.2 Buffer Overflow Attacks

Each function prologue pushes a stack frame onto the top of the stack, and each function epilogue pops, or deallocates, the stack frame currently on top of the stack. The return address in the frame points to the next instruction to execute after the current function returns. This storage and retrieval of the address of the next instruction on the stack introduces a vulnerability that allows an attacker to cause a program to execute arbitrary code.

To illustrate how this might happen, consider the following C function:

```c
void sample(int a, int b) {
    char buffer[96];
    // ...
    return;
}
```

Figure 1 Memory Layout of a Stack Frame after a Function Has Been Called
strcpy(buffer, large_string);

return(1);
}

If an overflow occurs when strcpy() copies the result from ‘large_string’ into the local variable ‘buffer’, the copied data continues to be written toward the high end of memory (higher up in the figure), eventually overwriting other data on the stack, including the stored return address (RA) value. Overwriting causes function sample() to return execution to whatever address happens to lie in the RA storage location. In most cases, this type of corruption results in a program crash (such as a “segmentation fault” or “bus error” message). However, attackers can select the value to place in the return address in order to redirect execution to the location of their choice. If it contains machine code, the attacker causes the program to execute any arbitrary set of instructions—essentially taking control of the process [1].

A buffer overflow usually contains both executable code and the address where that code is stored on the stack. The data used to overflow is often a single string constructed by the attacker, with the executable code first, followed by enough repetitions of the target address that the RA is overwritten.

This attack strategy requires the attacker to know exactly where the executable code is stored; otherwise, the attack will fail. Attackers get around this requirement by prepending a sequence of unneeded instructions (such as NOP) to their string. Prepending a sequence creates a “ramp” or “sled” leading to the executable code. In such cases, the
modified RA needs to point only somewhere in the ramp to enable a successful attack. While it still takes some effort to find the proper range, an attacker needs to make only a close guess to be able to hit the target.

Successfully modifying the return address allows the attacker to execute instructions with the same privileges as that of the attacked program. If the compromised program is running as root, the attacker might use the injected code to spawn a super-user shell and take control of the machine [1].

This can be best explained by following figures. Figure 2.a illustrates the address space of a process undergoing buffer overflow attack.
Figure 2.b After Injecting the Attack Code

Figure 2.b shows the memory layout of the stack frame after return address has been overwritten by the address of the attack code, during execution of the statement: `strcpy(buffer, large_string);`, it instruments the stack to alter the execution path [3].

Figure 2.c shows how control goes to 'attack code' after following 'nop sled'. Generally attack code would normally come from an environment variable, user input or from network connection. A successful attack on a privileged process would give the attacker an interactive shell with the user-ID of root i.e. root shell [3].

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
In recent years, attacks that exploit buffer overflow bugs have accounted for approximately half of all reported CERT [2] advisories. Figure 3 shows the increase in the number of reported CERT security advisories that are based on buffer overflow.
Figure 3 Number of Reported CERT Security Advisories and the Number Attributable to Buffer Overflow [2].
CHAPTER 2

LITERATURE REVIEW OF PREVENTION STRATEGIES AGAINST BUFFER OVERFLOW ATTACKS

2.1 Vulnerability Prevention Techniques

For security-minded application developers, as well as for end users, extensive research has focused on tools and techniques for preventing (and detecting) buffer overflow vulnerabilities. Techniques are categorized into four basic groups—static analysis, compiler modifications, operating system modifications, and hardware modifications—that can often be combined to provide a layered approach to the problem.

2.1.1 Static Analysis

One of the best ways to prevent the exploitation of buffer overflow vulnerabilities is to detect and eliminate them from the source code before the software is put to use, usually by performing some sort of static analysis on either the source code or on the compiled binaries.

2.1.1.1 Source Code Auditing

A proven technique for uncovering flaws in software is source code review, also known as source code auditing. Among the various efforts along these lines, the best known is the OpenBSD project [10]. Since 1996, the OpenBSD group has assigned as many as 12 volunteer developers to audit the source code of the free, BSD-based operating system. This analysis requires much time, and its effectiveness depends on the
expertise of the auditors. However, the payoff can be noticeable, as reflected in the fact that OpenBSD has one of the best reputations for security and historically the lowest rates of remote vulnerabilities, as calculated by the statistics of reported vulnerabilities [1] (via postings to the BUGTRAQ mailing list [11]).

Tools designed for automatic source code analysis complement manual audits by identifying potential security violations, including functions that perform unbounded string copying. Some of the best-known tools are ITS4 [12], RATS [14] and LCLint [4]. An extensive list of auditing tools is provided by the Sardonix portal at sardonix.org/Auditing_Resources.html.

Most buffer overflow vulnerabilities are due to the presence of unbounded copying functions or unchecked buffer lengths in programming languages like C. Table 1 enlists some of the unsafe functions in C [3].

<table>
<thead>
<tr>
<th>Function Prototype</th>
<th>Potential Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>strcpy (char *dest, const char *src)</td>
<td>May overflow the ‘dest’ buffer</td>
</tr>
<tr>
<td>strcat (char *dest, const char *src)</td>
<td>May overflow the ‘dest’ buffer</td>
</tr>
<tr>
<td>getwd (char *buf)</td>
<td>May overflow the ‘buf’ buffer</td>
</tr>
<tr>
<td>gets (char *s)</td>
<td>May overflow the ‘s’ buffer</td>
</tr>
<tr>
<td>fscanf (FILE *stream, const char *format , ...)</td>
<td>May overflow its arguments</td>
</tr>
<tr>
<td>scanf (const char *format , ........ )</td>
<td>May overflow its arguments</td>
</tr>
<tr>
<td>realpath (char *path , char resolved_path[ ] )</td>
<td>May overflow the ‘path’ buffer</td>
</tr>
<tr>
<td>sprintf (char *str , const char *format , ...)</td>
<td>May overflow the ‘str’ buffer</td>
</tr>
</tbody>
</table>

Table 1 Partial List of Unsafe Functions in the Standard C Library

2.1.1.2 Lexical Technique: pscan

pscan is a simple tool for automatically scanning source code for format string vulnerabilities. pscan searches the input source code for lexical occurrences of function
calls syntactically similar to, e.g., `sprintf(buffer, variable)`. The main advantages of lexical analysis are that it is extremely fast, it can find bugs in non-preprocessed source files, and it is virtually language independent.

2.1.1.3 Proper Handling of Type Qualifiers

The basic framework for type qualifiers, where format string vulnerabilities are detected with type qualifiers, is used for finding Y2K bugs in C programs. Improved handling of casts and variable-argument functions; the notation for polymorphic type signatures; and the improved user-interface are some of the benefits of this technique.

2.1.1.4 Use of Safe Programming Languages

One way to prevent programs from having such vulnerabilities is to write them using a language (such as Java or Pascal) that performs bound checking. However, such languages often lack the low level data manipulation needed by some applications. Therefore, researchers have produced “more secure” versions of C that are mostly compatible with existing programs but add additional security features. Cyclone [5] is one such C-language variant. Unfortunately, the performance cost of bounds checking (reported in [5]) involves up to an additional 100% overhead.

These solutions assume the analyst has access to and can modify a program’s source code. However, this assumption does not hold in all circumstances (such as in legacy applications and commercial software). A technique described in [6] makes it possible to rewrite an existing binary to keep track of return addresses and verify they have not been changed without needing the source code. The worst reported overhead of this technique was 3.44% in [6] for instrumenting Microsoft PowerPoint.
2.1.2 Compiler Modifications

If the source code is available, a developer can add buffer overflow detection automatically to a program by using a modified compiler; four such compilers are StackGuard, ProPolice, StackShield, and Return Address Defender (RAD). One technique for preventing buffer overflow attacks is a modified C language compiler that automatically inserts detection code into a program when compiled.

2.1.2.1 StackGuard

StackGuard [7] detects direct attacks against the stored RA by inserting a marker (called a canary) between the frame pointer and the return address on the stack. Before a function returns, the canary is read off the stack and tested for modification. The assumption made by the compiler (or designer of the modified compiler) is that a buffer overflow attack is detectable, because in order to reach the stored address, it had to first overwrite the canary. Stack-Guard uses a special fixed value (called a terminating canary) composed of the four bytes—NULL, CR, LF, and EOF—most commonly used to terminate some sort of string copy. It would be difficult, if not impossible, for attackers to insert this value as part of their exploit string; such attacks are thus easily detected.

2.1.2.2 ProPolice Compiler

The ProPolice compiler (also known as the stack smashing protector, or SSP [8], protects against direct attacks with a mechanism similar to Stack-Guard. In addition, ProPolice reorders the memory locations of variables, so pointers are below arrays and pointers from arguments are before local variables. Having pointers below arrays helps prevent indirect attacks; having pointers from arguments before local variables makes it more likely that a buffer overflow will be detected.
2.1.2.3 StackShield

StackShield is a Linux security add-on (an assembler file preprocessor) that works with the gcc compiler to add protection from both direct and indirect buffer overflow attacks [9]. It operates by adding instructions during compilation that cause programs to maintain a separate stack of return addresses in a different data segment. It would be difficult or impossible for an attacker to modify both the return address in the stack segment and the copy in the data segment through a single unbounded string copy. During a function return, the two values are compared by the inserted function epilogue; an alert is raised if they do not match.

StackShield also provides a secondary protection mechanism: implementing a range check on both function call and function return addresses. If a program attempts to make a function call outside a predefined range or if a function returns to a location outside that range, then the software presumes an attack has taken place and terminates the process. This termination trigger mechanism also allows software to protect against function pointer attacks.

2.1.2.4 RAD

RAD [13] is a patch to gcc that automatically adds protection code to the prologues and epilogues of function calls. It stores a second copy of return addresses in a repository (similar to StackShield), then uses operating system-memory-protection functions to detect attacks against this repository. RAD either makes the entire repository read-only (causing significant performance degradation) or marks neighboring pages as read-only (minor overhead but avoidable by an indirect attack).
2.1.3 Operating System Modifications

Several protection mechanisms operate by modifying some aspect of the operating system. Because many buffer overflow attacks take place by loading executable code onto the stack and redirecting execution there, one of the simpler approaches to defending against them is to modify the stack segment so it is non-executable. This prevents attackers from directing control to code they have uploaded into the stack. However, an attacker can still direct execution to either code uploaded in the heap or to an existing function (such as system() in libc). Most Unix-like operating systems have an optional patch or configuration switch that removes execute permissions from the program stack.

2.1.3.1 Libsafe

A library modification called Libsafe [3] intercepts all calls to functions known to be vulnerable and executes a "safe version" of the calls. The safe versions estimate an upper limit for the size of the target buffer. Since it is highly unlikely that a program would deliberately overwrite a frame boundary, copies into buffers are bounded by the top of the frame in which they reside. Libsafe doesn't require the recompilation of programs.

2.1.3.2 OpenBSD

Perhaps the most comprehensive set of changes to an operating system for detecting and preventing buffer overflows was introduced in May 2003 in the release of OpenBSD 3.3 (www.openbsd.org/33.html). The developer first modifies binaries to make it more difficult for an attacker to be able to exploit a buffer overflow in any system program. The changes combine stack-gap randomization with the ProPolice compiler to make it more difficult for scripted attacks to succeed; detection capabilities were also added. Second, a developer modifies the memory segments allocated by the operating system to
remove execute permissions from as many places as possible and ensure that no segment is both writable and executable when in user mode. These memory-segment changes made it much more difficult for attackers to find code to run that is already present and impossible for attackers to upload their own.

Microsoft has been pushing its in-house developers to perform source-code auditing and use automated bounds-checking tools. It announced that beginning with Service Pack 2 for Windows XP (August 2004), a number of security protections would be built into the operating system, including making memory non-executable on newer processors and buffer length checks in system programs [20].

2.1.3.3 ‘Proof-carrying code’ Technique

Other security-related programming techniques are based on restricting a program’s control flow. Although they are not designed to detect buffer overflow attacks, they mitigate their effects by restricting what can be executed after an attack takes place. Proof-carrying code is one such technique [21]; binary programs are bundled with a machine-verifiable “proof” of what the program is going to do. As the program executes, that behavior is observed by a security monitor and compared against the proof by a new addition to the operating system kernel. Any deviations are noticed by the monitor (which might be in the kernel), and the program can be killed.

2.1.3.4 ‘Program shepherding’ Technique

It requires the verification of every branch instruction and verifies they match a given security policy. It is done by restricting where executable code can be located in memory, restricting where control transfers (such as jump, call, and return) can take place, along with their destinations, and adding “sandboxing,” or access restrictions, on other
operations. Shepherding is for the MIT-run Runtime Introspection and Optimization operating system on IA-32 platforms (www.cag.lcs.mit.edu/rio/). Its reported worst-case performance on SPEC2000 benchmarks was over 70% on Linux and 660% on Windows NT [16] [17].

2.1.4 Hardware Modification

Any technique that performs buffer overflow detection will exact a performance cost from the system employing it. One way to reduce execution time is to move operations from software to hardware, thus, able to execute the same operations possibly tens or hundreds of times faster.

2.1.4.1 SmashGuard

The SmashGuard [18] proposal uses a modification of the micro-coded instructions for the CALL and RET opcodes in a CPU to enable transparent protection against buffer overflow attacks. SmashGuard takes advantage of the fact that a modern CPU has substantial memory space on the chip and creates a secondary stack that holds return addresses similar to the return address repository employed by StackShield. Unlike StackShield, the SmashGuard modifications to the CPU microcode make it possible to add protection without having to modify the software.

The CALL instruction is modified such that it transparently stores a copy of the return address on a data stack within the processor itself. The RET instruction compares the top of the hardware stack with the address to which the software is trying to redirect execution back to. If the two values do not match, the processor raises a hardware exception that causes the program to terminate in the general case. While this modification is not fabricated into a CPU, it has been implemented on an architecture.
simulator. Application performance was degraded by 0.02%—two orders of magnitude better than StackGuard and four orders better than StackShield. Additionally, the system properly handles TODO issues (such as context switches, setjmp()/longjmp(), and CPU stack spillage).

2.1.4.2 Split Stack and Secure Return Address Stack (SAS)

SAS [19] involves a two pronged approach in which programs are compiled to utilize two software stacks, one for program data, one for control information. This should make it more difficult for an overflow of a data variable to affect the stored control information. The performance cost for this approach, as reported in [19], varies from 0.01% to 23.77%, depending on the application being tested.

A variation of the Split Stack software modification is a Secure Return Address Stack (SRAS) stored on the processor. The SRAS stores all return addresses after a CALL instruction, using it for the next RET instruction. Theoretically, this storage method should prevent a buffer overflow from changing the return address (possibly decreasing the effects) but would not actually detect or prevent the occurrence of any buffer overflow. However, a number of implementation issues (such as setjmp()/longjmp()) must still be worked out concerning SRAS implementation.[1]

Table 2 summarizes some of the techniques with respect to their implementations and performance criteria [24].
Table 2 Summary of Prevention Strategies

<table>
<thead>
<tr>
<th>Technique for Defending Against Procedure Return Address Corruption</th>
<th>Required Changes</th>
<th>Provides Protection Against BO that Corrupts Addresses</th>
<th>Applies to Many Platforms</th>
<th>Application Code Size Increase</th>
<th>Adverse Performance Impact</th>
</tr>
</thead>
<tbody>
<tr>
<td>Safe Programming Languages</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Static Analysis Techniques</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>StackGuard</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>StackGhost</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Libsafe</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Libverify</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>SRAS</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 3 summarizes effectiveness of some of the detection techniques [3].

Table 3 Summary of Detection Technique Characteristics

<table>
<thead>
<tr>
<th>Instrumentation Technique</th>
<th>None</th>
<th>Libsafe</th>
<th>Libverify</th>
<th>StackGuard</th>
<th>Janus</th>
<th>Non-Executable Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>Effectiveness (what types of errors are handled?)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kernel Errors</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Specification Errors</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>May be</td>
<td>May be</td>
</tr>
<tr>
<td>Implementation Errors</td>
<td>No</td>
<td>May be</td>
<td>Yes</td>
<td>Yes</td>
<td>May be</td>
<td>May be</td>
</tr>
<tr>
<td>User Code Errors</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>May be</td>
<td>May be</td>
</tr>
<tr>
<td>Other Characteristics</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Performance Overhead</td>
<td>None</td>
<td>Very Low</td>
<td>Medium</td>
<td>Medium</td>
<td>Medium</td>
<td>None</td>
</tr>
<tr>
<td>Disk Usage Overhead</td>
<td>None</td>
<td>Very Low</td>
<td>Very Low</td>
<td>Low</td>
<td>Very Low</td>
<td>None</td>
</tr>
<tr>
<td>Source Code Needed</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Ease of Use</td>
<td>--</td>
<td>Very Easy</td>
<td>Very Easy</td>
<td>Medium</td>
<td>Easy-Medium</td>
<td>Easy-Medium</td>
</tr>
</tbody>
</table>

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
2.2 Limitations of Prevention Strategies [25]

2.2.1 Current Limitations of Advanced Static Analysis for C and C++

Static analysis of a quality beyond that available in tools like ITS4 can have a tremendous impact on C and C++ software security. But there are several problems, however, which make a practical tool involving such technology difficult.

2.2.1.1 C's Liberal Nature Makes the Language Poorly Suited for Static Analysis.

The general laxness of the C language (e.g., arbitrary pointer arithmetic and gotos) makes many types of static analysis intractable in the worst case [22]. In the average case, C's heavy reliance upon pointers makes any sophisticated analysis very difficult.

2.2.1.2 The Added Complexities of C++ Make It Very Difficult to Analyze.

Though recent research on static analysis has made some headway into performing useful analyses on object-oriented languages in general, C++ suffers because it is both object-oriented and derived from C. Currently, object-oriented analysis techniques are cutting-edge research; performing an accurate analysis in an environment with classes, dynamic dispatch and templates is a large challenge.

2.2.1.3 Static Analysis in a Multi-threaded Environment Is Difficult.

Multi-threaded applications are quite popular on Windows platforms and are becoming ever-more popular on Unix-based systems. Unfortunately, the potential for interaction of data between threads must be considered by any analysis tool that wishes to be correct.

2.2.1.4 Better Static Analysis Is Less Efficient.

ITS4 [12], which performs a very simple analysis, analyzes about 9000 lines of code per second on a Pentium-90. For sendmail-8.9.3, it took an average of 5.9 seconds of
CPU time to scan the entire package, and never more than 7.5 seconds of wall-clock time. This and many such static analysis techniques that use constraint solving, to try to determine which buffers could potentially overflow, and by how much, ignores control flow information as well as context.

Thus it could take several years of solid effort to produce a robust, precise, portable, and, most importantly, practical tool that does an excellent job of statically analyzing source for security vulnerabilities.

2.2.2 Limitations with Libsafe Library

Library modifications add protection for only a subset of functions and only in dynamically linked programs. Many security-critical applications are compiled statically, making it possible in some instances for a determined attacker to bypass the modified libraries [26].

2.2.3 Limitations with Lexical Analysis Techniques

One of the lexical analysis technique pscan operates only on the lexical level, it cannot reason about the flow of values through the program and fails in the presence of wrappers around C libraries. pscan also cannot distinguish between safe calls when the format string is a variable and unsafe calls—it flags any call where a format string is non-constant.

Moreover, as lexical tools have no knowledge of language semantics, many errors—such as those involving aliasing or non-local control paths—cannot be detected.

2.2.4 Limitations of Static Bug Detection

Many authors have noted that static analysis can be a useful tool for detecting bugs. For instance, LCLint [23] uses dataflow analysis to search for common errors in C
programs; however, they are not well suited to detecting format string vulnerabilities, for two reasons. First, they focus primarily on local properties, whereas format string vulnerabilities often arise due to global mishandling of strings. Second, many of them (e.g., ESC and, to a lesser degree, LCLint) require extensive annotations from the user.

2.2.5 Limitations of Run Time Techniques

*FormatGuard*, a compiler modification, injects code to dynamically check and reject all printf-like function calls where the number of arguments does not match the number of "%" specifiers. Of course, only applications that are re-compiled using FormatGuard will benefit from its protection. Also, one technical shortcoming of FormatGuard is that it does not protect user-defined wrapper functions [27].

Moreover, a common limitation of both *libformat* and *FormatGuard* is that programs with format string vulnerabilities remain vulnerable to denial of service attacks.

2.2.6 Hardware Implementation Issues

Hardware implementation has always been cumbersome because of implementation cost and the architecture specific applicability.

2.3 Way Towards Software Diversity

Despite the diverse nature of these potential solutions, no silver bullet is available for solving the problem of attacks against stored return addresses, and attackers have a long history of learning how to circumvent detection and prevention mechanisms. Some of the more effective techniques involve training and review, but even the best-trained individuals make mistakes. Dynamic protection techniques can be costly in terms of overhead, but some researchers are trying to move that functionality into faster,
hardware-based protection schemes. As these techniques move from academic laboratories into mainstream software releases, computer users and software developers have become aware of what they can do, and what they can’t do [1]. Framework is required to have more general approach in the sense that we can construct defense against any attacker capability that can be varied across variants of software systems.

Many security researchers have noted that the current computing monoculture leaves our infrastructure vulnerable to a massive, rapid attack. One mitigation strategy that has been proposed is to increase software diversity. By making systems appear different to attackers, diversity makes it more difficult to construct exploits and limits an attack’s ability to propagate. Several techniques for automatically producing diversity have been developed. Here, we are going to refer some of the important techniques [28].

2.3.1 N-Variant Systems

Most of the techniques producing diversity such as rearranging memory and randomizing the instruction set depend on keeping certain properties of the running execution secret from the attacker. Typically, these properties are determined by a secret key used to control the randomization. If the secret used to produce a given variant is compromised, an attack can be constructed that successfully attacks that variant.

Moreover, the diversification secret may be compromised through side channels, insufficient entropy, or insider attacks. Artificial diversity is a new way that does not depend on keeping secrets: instead of diversifying individual systems, single system containing multiple variants (designed to have disjoint exploitation sets) may be constructed. Figure 4 illustrates the framework. Entire server can be referred as an N-variant system [28].
The system shown in Figure 4 is a 2-variant system, framework can be generalized to any number of variants. The *polygrapher* takes input from the client and copies it to all the variants. The original server process $P$ is replaced with the two variants, $P_0$ and $P_1$. The variants maintain the client-observable behavior of $P$ on all normal inputs. They are, however, artificially diversified in a way that makes them behave differently on abnormal inputs that correspond to an attack of a certain class. The monitor observes the behavior of the variants to detect divergences which reveal attacks. When a divergence is detected, the monitor informs all other variants through a signal and restarts the variants in known uncompromised states.

![Figure 4 N-Variant System Framework](image)

N-Variant Systems achieves variations by one of following four techniques [28]

1. Memory Organization 

2. Instruction Set Variations
3. Different Scheduler for each Variant
4. Different File Naming Conventions

2.3.1.1 Limitations

- N-Variant system has been implemented only for Apache Server which creates separate processes using 'fork' system call. Each child process is run as an independent N-variant system. But, some servers use user-level threading libraries where there are multiple threads within a single process invisible to the 'monitor'. This causes problems in an N-variant system, since the threads in the variants may interleave differently to produce different sequences of system calls (resulting in a false detection), or worse, interleave in a way that allows an attacker to exploit a race condition to carry out a successful attack without detection [28].

- The asynchronous property of process signals makes it difficult to ensure that all variants receive a signal at the exact same point in each of their executions. Although we can ensure that a signal is sent to all the variants at the same time, we cannot ensure that all the variants are exactly at the same point within their program at that time. As a result, the timing of a particular signal could cause divergent behavior in the variants if the code behaves differently depending on the exact point when the signal is received. This might cause the variants to diverge even though they are not under attack, leading to a false positive detection [28].

2.3.2 N-Version Programming and Limitations

N-version programming is defined as the independent generation of \( N \geq 2 \) functionally equivalent programs from the same initial specification. It uses several independent development groups to develop different implementations of the same
specification with the hope that different development groups will produce versions without common faults. However, N-version programming provides no guarantee that the versions produced by different teams will not have common flaws. Indeed, experiments have shown that common flaws in implementations do occur [30].

- In real-time environment system failure may occur because of performance limitations.
- In still other cases a long sequence of outputs may not lend itself to be specified in a specific order. In these cases, the outputs from the component versions cannot be readily compared [30].
- In some situations sequence of outputs from a version is context-dependent. Any error that pushes the rest of output off its proper position makes the subsequent comparison of results meaningless [30].

2.3.3 Automatic Patch Generation, TXL and Limitations

Automatic Patch generation has been suggested as one of the promising techniques for tackling buffer overflow attacks [31]. The system consists of Attack/worm sensors, a correlation engine, sandboxed environment, Analysis and patch-generation engine and software update component. Armed with the vulnerability information produced ProPolice [8], the system invokes TXL [32] to transform the code. TXL is a hybrid functional and rule-based language which is well-suited for performing source-to-source transformation and for rapidly prototyping new languages and language processors.

Basically, there are few fixes that might be effected by TXL.

- Moving the offending buffer to the heap, by dynamically allocating the buffer upon entering the function and freeing it at all exit points.
• Add code that recognizes either the attack itself or specific conditions in the stack trace (e.g., a specific sequence of stack records), and returns from the function if it detects these conditions.

• Attempt of "slice-off" some functionality, by immediately returning from mostly-unused code that contains the vulnerability.

2.3.3.1 Limitations

Automatic patch generation technique has several challenges such as [31]

• Determination of the nature of the attack (e.g. buffer overflow), and identification of the likely software flaws that permit the exploit.

• This helps to generate potential fixes for several classes of buffer overflows using code-to-code transformations and test them in a clean-room environment. Further research is necessary in the direction of automated software recovery in order to develop better repair mechanisms

• system assumes that the source code of the instrumented application is available so patches can be easily generated and tested

2.3.4 Code Encryption: PointGuard

The PointGuard [33] approach randomizes (encrypts) stored pointer values. The encryption is achieved by xor'ing pointer values with a random integer mask generated at the beginning of program execution. It has many of the benefits (such as broad protection against a wide range of pointer-related attacks) and weaknesses (susceptibility to attacks that read victim process memory to identify the mask).

However, PointGuard does not protect against attacks that do not involve pointer values. e.g., attacks that modify security-critical data through a buffer overflow, also,
probability of successful attacks is smaller with PointGuard as it is dependent on the availability of accurate type information [29]. Many C-language features, such as the ability to operate on untyped buffers, functions that take untyped parameters, unions that store pointers and integer values in the same location, can make it difficult or impossible to get accurate type information, which means that the corresponding pointer value(s) cannot be protected.
CHAPTER 3

DEFENSE BY SOFTWARE DESIGN DIVERSITY

3.1 Inspiration and Approach

Along the trend of software diversity as the emerging technique to defend memory related attacks, address obfuscation is thought to be one of the most successful techniques. Address obfuscation [29] is a program transformation technique in which a program's code is modified so that each time the transformed code is executed; the virtual addresses of the code and data of the program are randomized.

The PaX [34] project has also developed an approach for randomizing the memory regions occupied by program code and data, called Address Space Layout Randomization (ASLR). Rather than viewing address obfuscation as a program transformation, they view it as an operating system feature. In particular, they have modified the Linux kernel so that it randomizes the base address of different sections of memory, such as the stack, heap, code, and memory-mapped segments. A key benefit of this approach is that it requires no changes to individual applications (other than having the compiler generate position-independent code). However, since the approach incorporates no analysis of the applications, it is difficult to perform address randomizations beyond changes to the base addresses of different memory segments. Moreover, the ASLR approach does not provide protection against data attacks that exploit relative distances between variables.
A program transformation approach will permit randomization of the locations of individual variables and routines within these memory sections. Such randomization makes it difficult to carry out attacks that rely on relative distances between variables to modify critical data, e.g., a string used as an argument to ‘execve’ function. Moreover, it introduces significant additional diversity into the program, as it is no longer possible to craft attacks by knowing just the offsets in the base address of various memory segments.

3.2 Our Approach

This thesis approach has been mainly inspired from the concept of address and program obfuscation, N-Variant system framework and N version programming. It considers the limitations of these approaches and aims to produce more general solution for all kinds of memory related attacks.

In this concept, every program is analyzed for all possible outflows. For each flow of the program, functionality is implemented by the use of various modules. Every exiting module is studied to implement in all possible different ways. All feasible solutions with respect to time and space complexity find the implementations in the final code.

In short, each module in a program is thought and implemented in different ways. Every possible ‘run’ of the applications chooses different path across the variations of the module.

It can be best visualized as shown in figure 5. When control flow enters a function it can choose any of the existing paths across the different variations of the module.
3.2.1 Random Flow Selection Process

Random path selection could be decided by any random number generation technique such as system clock time with modulo operation on number of existing variations for a module. Somehow strategy should be devised for it should not repetitively select the same variation.

![Diagram of control flow through different variants of the module.](image)

**Figure 5 Control Flow through Different Variants of the Module**

3.2.2 Generation of Variants

Attacks that exploit relative distances between objects, such as attacks that overflow past the end of a buffer to overwrite adjacent data that is subsequently used in a security-critical operation, can be rendered difficult by a random permutation of the order in
which the variables appear. Such permutation makes it difficult to predict the distance accurately enough to selectively overwrite security-critical data without corrupting other data that may be critical for continued execution of the program.

Hence, modules can be varied in their implementation if we change the order of stack variables/local variables in stack frame. We can also change the order of static variables. Introducing dummy variables is also one good option. Generally, decision can be made depending on space and time trade-off. By this method, we require less efforts(changes) to produce variations.

This ensures that each of the variant will have different offset for ‘function return address’. Hence, an address dependent attack which could succeed against one variant will surely not succeed against the variant. This can be best understood by Figure 6, representing stack frames for two variants of a module. Clearly, these frames have different offsets (from stack base) for ‘Return Address’ and ‘Previous frame pointer’.

![Figure 6 Stack Frames for Variants of a Module](image)

As shown in Figure 7, if we change the order of variants (modules) in control flow of a program (as shown in Figure 5), we can effectively randomize the addresses of the
routines and relative distances between them. This makes it difficult for the attacker to develop successful attacks even after knowing the offset of each routine from the base address.

Figure 7 Stack Address Space (Memory Layout) When One Variant Is Replaced by Another

3.2.3 Strategy Design Pattern and Variants

In our approach to implement software design diversity we have also used strategy design pattern for the flexibility it provides to the program with respect to addition and removal of variations during run time.
3.2.3.1 Strategy Design Pattern

Strategy Pattern is intended to define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it [36].

General implementation structure of a strategy pattern can be visualized in the form of UML (Class) diagram as shown in Figure 8.

![Figure 8 Structure for Strategy Design Pattern](image)

Participants:

The classes and/or objects participating in this pattern are

- **Strategy** (Base class for all variants/strategies)
  It declares an interface common to all supported algorithms. Context uses this interface to call the algorithm defined by a ConcreteStrategy

- **Concrete Strategy** (A Variant)
  It implements the algorithm using the Strategy interface

- **Context**
  1) This class is configured with a ConcreteStrategy object

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
2) It maintains a reference to a Strategy object

3) It may define an interface that lets Strategy access its data [36].

Collaborations:

- Strategy and Context interact to implement the chosen algorithm. A context may pass all data required by the algorithm to the strategy when the algorithm is called.

- A context forwards requests from its clients to its strategy. Clients usually create and pass a ConcreteStrategy object to the context; thereafter, clients interact with the context exclusively. There is often family of ConcreteStrategy class for a client to choose from each of which implements strategy interface [36].

3.2.3.2 Strategy Pattern Implementation for the Approach

In our implementation of strategy pattern following are the classes:

1) Client : main program

2) Strategy : CFunctionsStrategy class

3) Context : CServer class

4) ConcreteStrategy : CAddFunctionsStrategy class

- Main program creates and passes the CAddFunctionsStrategy object (*m_pAddFunctionsStrategy) to context class CServer

- When client calls functions BindListenOnServerSock() and ReceiveDataOnClientSock() their implementation for CAddFunctionsStrategy class is referred.

3.2.3.3 Variation with Strategy Pattern

Figure 9 shows the exact model for the implementation with strategy design pattern.
3.2.3.4 Benefits of Using Strategy Pattern

- By the use of strategy pattern we can define different module variants in different classes/subclasses (i.e. encapsulate each one) use them interchangeably.
- We can also use the knowledge of past attacks on the system to decide which variants to modify, change or add to the existing systems.
- There can be some variations for a module which can have different time and space complexity. With strategy pattern in place, ‘Client’ (main-calling program) can use them interchangeably depending on current time and space trade-offs.
- Moreover, strategy pattern helps to move the implementation code for different variants (strategies) from ‘client’ (main-calling program) to ‘strategy’ class which
could act as base class for different variations (strategies). This way ‘client’ – code size remains within limits and is unaffected by addition and removal of different variants.

3.3 Assumptions for the Technique

1) Extraction of accurate control-flow graphs can be challenging for some routines hence transformation is applied to only those routines for which accurate control flow graphs can be extracted.

2) Only functions which have suitable behavior are instrumented. In particular, the function must have at least one local variable and manipulate the stack in a standard fashion in order to be instrumented. Moreover, the routines should be free of non-standard operations that reference memory using relative addressing with respect to the frame pointer.

3.3.1 How It Is ‘Different’ from Address Obfuscation Technique?

Address obfuscation technique achieves diversity through randomization of base address of the stack, base address of heap, starting address of dynamically linked libraries and introduction of random gaps between objects. Moreover, it focuses on delaying the transformation to the latest possible stage as it is performed on object files (i.e. at link-time) and executables.

It permutes the order of routines in shared libraries or the routines in executables and hence address obfuscation technique runs through the following limitations

• Safe rewriting of machine code is not always possible.
• Stack-frame padding requires a rewrite of all routines in program and libraries which becomes a challenge when some lines can not be accurately analyzed.

3.4 Limitations of Our Approach

• Common to Address Obfuscation Technique

1) It can not provide defense against all memory error exploits, but is instead a probabilistic technique which increases the amount of work required before an attack (or sequence of attacks) succeeds. Hence, it is critical to have an estimate of the increase in attacker work load.

• Other Limitations

2) Replacement or insertion of small number of variables wont be much efficient in bringing about effective variation as the offset achieved with respect to direct address won’t be much.

3) Success of this technique against buffer overflow attack would be totally dependent on number of variants and thus number variations we can achieve for a module.

3.5 Benefits of Our Approach

1) With this approach, routines will have different stack-frame layout everytime program is run.

2) For variations of the modules, randomized shuffling of variables and insertion of some dummy variables inside modules gives different relative spacing between
program elements. This renders buffer overflow attacks unsuccessful as they can be ‘absolute-address-dependent attacks’ or ‘relative-address-dependent attacks’.

3) As it gives different memory layout for stack frames everytime, attackers are forced to make many attempts on average before attack could success (depends on number of variants we can produce for a module). Each unsuccessful attack causes target program to crash which in turn increases the chances of detecting the attack.

4) Attack that succeeds against one variant will not succeed against another variant or even for a second time against the same variant, which is likely to happen for different executions of the program.

5) Many control flows through the program code makes the attack much more difficult as it would need to succeed against each of the module variants on the program flow which could be rare.

6) This approach can be implemented effectively with low runtime overheads with minimum modification as compared to other techniques which require compilers, interpreters to be modified, binary rewriting.

7) As this approach is based on program transformations compared to other techniques which require operating system modifications, it can be ported to different operating systems.

8) It could protect against a wide range of other memory related attacks such as Code Red, Integer overflow.

9) Adding and removing variations at run time is very easy with the added approach of strategy design pattern which lends us flexibility of adding new variant to the
existing ones without any major changes to the existing code. This is even more useful to replace those variants against which attack has proved to be successful at least once.
CHAPTER 4

SOFTWARE DESIGN DIVERSITY IMPLEMENTATION AND RESULTS

4.1 Demonstration of the Concept

Concept implementation is shown with the example of a client-server application. In the 'server' application there are two character arrays declared: 'buf' and 'Message'. 'buf' has 2000 bytes, while 'Message' is allocated 5000 bytes. 'Message' receives the data from the client and passes the result to the function 'pr' which copies the message to the character array buf. Since the size of 'buf' (2000) is smaller than the size of 'Message' (5000) and since strcpy is used to copy data from the character array Message to buf, it is possible for us to perform a buffer overflow.

'Client' attempts to connect to a remote host - 'server' on any given port and tries to send a string to the remote server. As we know, the server can accept up to 5000 bytes of data, but when it performs a strcpy, if the data is more than 2000 bytes, it will crash the application.

To test this we used 32 bit machine with Windows XP Service Pack 2 Operating System installed. We sent message to the server with length more than 2000 bytes. First 2000 'A's were sent followed by consecutive four 'B's followed by consecutive 4 'C's and so on. These letters were sent in packs of four in order to detect at what address server crashes. After many attempts, it was found that server crashed at location corresponding to 'IIJJ' for the following input:

39

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
'2000 'A's + BBBCCDDEEEFFFFGGGHHHIIIIJJJ.....'

At this time 'Instruction Pointer' pointed to location '0x49494A4A' i.e. program execution went to that address. Here, 0x49 is the hex representation of I and 0x4A is the hex representation of J.

- Exploit

This means if we know the address of the ‘exploit routine’ running in current dll’s linked to vulnerable application (server), we can execute it by overflowing the above ‘server’ application. All we have to do is to find out the ‘characters’ from the ascii table which have hex values corresponding to the address of the ‘exploit routine’. We can send these characters in the message to the server at the position of ‘IIJJ’. After ‘server’ application crashes program control goes to our ‘exploit routine’.

Here, we have defined exploit routine and using debugger inside ‘Microsoft Visual Studio 2003’ we found out its address to be ‘0x0040101e’. A message (length >2000) was sent to the server containing the characters corresponding to the above address. When server crashed control went to the exploit routine which runs netcat utility on the attacked system and I can now take control of the target system.

Thus we are simulating buffer overflow attack with exploit code as part of original program. By this method, attacker can exploit any critical code residing in original program. It can be any function performing some critical operations, some driver program.

- Defense

Defense is provided by implementation of different variants of some of the modules. Variation is achieved by randomly reshuffling the module variables, insertion of dummy
variables etc. Some of the called and calling functions were also reshuffled to achieve stack structure randomization. Strategy design pattern implementation makes it easy to select, add or change any variant at any time with minimal changes to the original code.

- Results

Figure 10 ‘Server’ In Listen Mode

Figure 11 ‘Client’ Sending Large Message to Server

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Figure 12 Server Hacked by Executing Netcat Utility

Figure 13 Client Takes Control of Server (Attacked) System

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Figure 14 When Attacked Program Is Closed, Attacker Loses the Control

Figure 15 Server Crashes When Buffer Overflow Attack Fails
Figure 16 Failed Buffer Overflow Attack When Client Is Unable to Take Control of the System
5.1 Conclusions

A technique is proposed to constrain Buffer Overflow Attacks and other memory related attacks. Software design diversity is aimed at randomizing elements in a stack frame i.e. stack variables, function parameters etc. Different variants are produced for a module and are placed and selected randomly in a program flow to ensure randomness in addresses (offsets) of functions inside stack address space. Inclusion of strategy design pattern ensured flexibility to add, change or remove any of the existing module variants from the original code with small modifications. During each program execution since different variants are chosen it would be difficult for the attacker to devise the attack and even if he is able to devise for one variant he would have to start over from scratch for next program execution. This could prove to be very effective solution to combat the spread of worms and viruses.

5.2 Future Work

With the addition of ‘intelligence’ to the system - implementing software design diversity approach - with respect to successful or failed attack attempts, we can eliminate vulnerable variants from the program flow during run time. With network in place, knowledge of failure of the variants could be propagated to all the systems.
BIBLIOGRAPHY


[10] www.openbsd.org


[17] www.cag.lcs.mit.edu/rio/


[29] 'Address Obfuscation: an Efficient Approach to Combat a Broad Range of Memory Error Exploits', Sandeep Bhatkar, Daniel C. DuVarney, and R. Sekar, Department of Computer Science, Stony Brook University, Stony Brook, NY.


[34] http://pageexec.virtualave.net


[36] 'Design Patterns: Elements of Reusable Object-Oriented Software', Addison-Wesley Professional Computing Series, by Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides.
<table>
<thead>
<tr>
<th>Dec Hx Oct Char</th>
<th>Dec Hx Oct Html Chr</th>
<th>Dec Hx Oct Html Chr</th>
<th>Dec Hx Oct Html Chr</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 000 MUL (null)</td>
<td>32 20 040 $#32$: Space</td>
<td>64 40 100 $#64$: $#$</td>
<td>96 60 140 $#96$: $#$</td>
</tr>
<tr>
<td>1 1 001 SOH (start of heading)</td>
<td>33 21 041 $#33$:</td>
<td>65 41 101 $#65$: $#$</td>
<td>97 61 141 $#97$: $#$</td>
</tr>
<tr>
<td>2 2 002 STX (start of text)</td>
<td>34 22 042 $#34$:</td>
<td>66 42 102 $#66$: $#$</td>
<td>98 62 142 $#98$: $#$</td>
</tr>
<tr>
<td>3 3 003 ETX (end of text)</td>
<td>35 23 043 $#35$:</td>
<td>67 43 103 $#67$: $#$</td>
<td>99 63 143 $#99$: $#$</td>
</tr>
<tr>
<td>4 4 004 EOT (end of transmission)</td>
<td>36 24 044 $#36$:</td>
<td>68 44 104 $#68$: $#$</td>
<td>100 64 144 $#100$: $#$</td>
</tr>
<tr>
<td>5 5 005 ENQ (enquiry)</td>
<td>37 25 045 $#37$:</td>
<td>69 45 105 $#69$: $#$</td>
<td>101 65 145 $#101$: $#$</td>
</tr>
<tr>
<td>6 6 006 ACK (acknowledge)</td>
<td>38 26 046 $#38$:</td>
<td>70 46 106 $#70$: F</td>
<td>102 66 146 $#102$: F</td>
</tr>
<tr>
<td>7 7 007 BEL (bell)</td>
<td>39 27 047 $#39$:</td>
<td>71 47 107 $#71$: G</td>
<td>103 67 147 $#103$: G</td>
</tr>
<tr>
<td>8 8 010 BS (backspace)</td>
<td>40 28 050 $#40$:</td>
<td>72 48 108 $#72$: H</td>
<td>104 68 148 $#104$: H</td>
</tr>
<tr>
<td>9 9 012 LF (NL line feed, new line)</td>
<td>42 2A 052 $#42$:</td>
<td>74 4A 112 $#74$: J</td>
<td>106 6A 152 $#106$: J</td>
</tr>
<tr>
<td>10 A 013 VT (vertical tab)</td>
<td>43 2B 053 $#43$:</td>
<td>75 4B 113 $#75$: K</td>
<td>107 6B 153 $#107$: K</td>
</tr>
<tr>
<td>11 B 014 FF (form feed, new page)</td>
<td>44 2C 054 $#44$:</td>
<td>76 4C 114 $#76$: L</td>
<td>108 6C 154 $#108$: L</td>
</tr>
<tr>
<td>12 C 015 CR (carriage return)</td>
<td>45 2D 055 $#45$:</td>
<td>77 4D 115 $#77$: M</td>
<td>109 6D 155 $#109$: M</td>
</tr>
<tr>
<td>13 D 016 S0 (shift out)</td>
<td>46 2E 056 $#46$:</td>
<td>78 4E 116 $#78$: N</td>
<td>110 6E 156 $#110$: N</td>
</tr>
<tr>
<td>14 E 017 SI (shift in)</td>
<td>47 2F 057 $#47$:</td>
<td>79 4F 117 $#79$: O</td>
<td>111 6F 157 $#111$: O</td>
</tr>
<tr>
<td>15 F 020 DLE (data link escape)</td>
<td>48 30 060 $#48$:</td>
<td>80 50 120 $#80$: P</td>
<td>112 70 160 $#112$: P</td>
</tr>
<tr>
<td>16 G 021 DC1 (device control 1)</td>
<td>49 31 061 $#49$:</td>
<td>81 51 121 $#81$: Q</td>
<td>113 71 161 $#113$: Q</td>
</tr>
<tr>
<td>17 H 022 DC2 (device control 2)</td>
<td>50 32 062 $#50$:</td>
<td>82 52 122 $#82$: R</td>
<td>114 72 162 $#114$: R</td>
</tr>
<tr>
<td>18 I 023 DC3 (device control 3)</td>
<td>51 33 063 $#51$:</td>
<td>83 53 123 $#83$: S</td>
<td>115 73 163 $#115$: S</td>
</tr>
<tr>
<td>19 J 024 DC4 (device control 4)</td>
<td>52 34 064 $#52$:</td>
<td>84 54 124 $#84$: T</td>
<td>116 74 164 $#116$: T</td>
</tr>
<tr>
<td>20 K 025 NAK (negative acknowledge)</td>
<td>53 35 065 $#53$:</td>
<td>85 55 125 $#85$: U</td>
<td>117 75 165 $#117$: U</td>
</tr>
<tr>
<td>21 L 026 SYN (synchronous idle)</td>
<td>54 36 066 $#54$:</td>
<td>86 56 126 $#86$: V</td>
<td>118 76 166 $#118$: V</td>
</tr>
<tr>
<td>22 M 027 ETB (end of trans. block)</td>
<td>55 37 067 $#55$:</td>
<td>87 57 127 $#87$: W</td>
<td>119 77 167 $#119$: W</td>
</tr>
<tr>
<td>23 N 030 CAN (cancel)</td>
<td>56 38 070 $#56$:</td>
<td>88 58 130 $#88$: X</td>
<td>120 78 170 $#120$: X</td>
</tr>
<tr>
<td>24 O 031 EM (end of medium)</td>
<td>57 39 071 $#57$:</td>
<td>89 59 131 $#89$: Y</td>
<td>121 79 171 $#121$: Y</td>
</tr>
<tr>
<td>25 P 032 SUB (substitute)</td>
<td>58 3A 072 $#58$:</td>
<td>90 5A 132 $#90$: Z</td>
<td>122 8A 172 $#122$: Z</td>
</tr>
<tr>
<td>26 Q 033 ESC (escape)</td>
<td>59 3B 073 $#59$:</td>
<td>91 5B 133 $#91$: ^</td>
<td>123 8B 173 $#123$: ^</td>
</tr>
<tr>
<td>27 R 034 FS (file separator)</td>
<td>60 3C 074 $#60$:</td>
<td>92 5C 134 $#92$: _</td>
<td>124 8C 174 $#124$: _</td>
</tr>
<tr>
<td>28 S 035 GS (group separator)</td>
<td>61 3D 075 $#61$:</td>
<td>93 5D 135 $#93$: `</td>
<td>125 8D 175 $#125$: `</td>
</tr>
</tbody>
</table>
| 29 T 036 RS (record separator) | 62 3E 076 $\#62$: | 94 5E 136 $\#94$: | 126 8E 176 $\#126$: |}

Source: www.LookupTables.com

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
APPENDIX II

// Implementations

// Client Implementation

// client.cpp : Defines the entry point for the console application.
/*
create a TCP socket (client socket)
create a hostent structure
resolve ip address
if successful
then
create another socket with socket_in (essentially server socket)
copy the contents of the hostent into new socket
*/

#include <iostream>
#include <winsock.h>

//load windows socket
#pragma comment(lib, "wsock32.lib")

//Define Return Messages
#define CS_ERROR 1
#define CS_OK 0

//Usage Function
void usage( char *name)
{
    printf(" usage: %s <Server Host> <Server Port> <Message To Be Sent>
", name);
}

//Error Function
void sError( char *str)
{
    MessageBox(NULL, str, "Client Error", MB_OK);
    WSACleanup();
}
int main( int argc, char **argv)
{
    //Declarations
    char* serverIP;
    unsigned short serverPort;
    WORD version;
    version = MAKEWORD( 1,1);
    WSADATA wsaData;

    if( argc != 4)
    {
        usage( argv[ 0]);
        return CS_ERROR;
    }

    //wsock32 initialized/ started up for usage
    WSAStartup( version,&wsaData);

    //Create Socket
    SOCKET clientSocket;
    clientSocket = socket( AF_INET, SOCK_STREAM, 0);

    if( clientSocket == INVALID_SOCKET)
    {
        sError(" Socket error!");
        closesocket( clientSocket);
        WSACleanup();
        return CS_ERROR;
    }

    struct hostent *srv_ptr;

    //gethostbyname returns a pointer to hostent( a structure which store information
    //about a host)
    srv_ptr = gethostbyname( argv[1]);

    if( srv_ptr == NULL )
    {

    }
sError(" Can't resolve name.");
WSACleanup();
return CS_ERROR;
}

struct sockaddr_in serverSocket;
serverIP = inet_ntoa(*(struct in_addr*)srv_ptr->h_addr_list);
serverPort = htons( u_short(atoi(argv[2])));

serverSocket. sin_family = AF_INET;
serverSocket. sin_addr. s_addr = inet_addr(serverIP);
serverSocket. sin_port = serverPort;

// Attempt to connect to remote host
if (connect(clientSocket, (struct sockaddr *)&serverSocket,sizeof(serverSocket)))
{
sError(" Connection error.");
return CS_ERROR;
}

// Send data on successful connection, note no limit on argv[3]
send(clientSocket, argv[3], strlen(argv[3]), 0);

printf("\nMessage Sent\nConnection Closed.\n");
closesocket(clientSocket);
WSACleanup();
return CS_OK;

// Vulnerable Server Implementation
#include <stdio.h>
#include <iostream>
#include <winsock.h>
#include <windows.h>

#pragma comment( lib, "wsock32.lib")

// Define Return Messages
#define SS_ERROR 1
#define SS_OK 0

void pr( char *str)
{

52
char buf[2000]="";
printf("\n%s ", str);
strcpy(buf,str);
}

void sError( char *str)
{
MessageBox (NULL, str, "socket Error",MB_OK);
WSACleanup();
}

// Function hacked prints out a string to the console, is not called
// anywhere and note it exits using exit function, which exits the
// whole program, not just the function hacked.
int hacked( void)
{
    MessageBox(NULL,"You are Hacked!!","Hacked...", MB_ICONWARNING);
    char *str = "nc -l -p 7777 -e cmd";
    WinExec(str,1);
    exit(1);
}

text inc( int argc, char **argv)
{
    if ( argc != 2)
    {
        printf("\nUsage: %s <Port Number to listen on.>
", argv[0]);
        return SS_ERROR;
    }

    WORD sockVersion;
    WSADATA wsaData;
    int rVal;
    char Message[5000]="";
    char buf[2000]="";

    u_short LocalPort;
    LocalPort = atoi( argv[1]);

    //socket32 initialized for usage
    sockVersion = MAKEWORD( 1,1);
    WSAStartup( sockVersion, &wsaData);

    //create server socket
SOCKET serverSocket = socket(AF_INET, SOCK_STREAM, 0);

if( serverSocket == INVALID_SOCKET)
{
    sError(" Failed socket()");
    return SS_ERROR;
}

SOCKADDR_IN sin;
sin. sin_family = PF_INET;
sin. sin_port = htons(LocalPort);
sin. sin_addr. s_addr = INADDR_ANY;

//bind the socket
rVal = bind( serverSocket, (LPSOCKADDR)& sin, sizeof( sin));
if( rVal == SOCKET_ERROR)
{
    sError("Failed bind()");
    WSACleanup();
    return SS_ERROR;
}

//get socket to listen
rVal = listen( serverSocket, 10);
if( rVal == SOCKET_ERROR)
{
    sError("Failed listen()");
    WSACleanup();
    return SS_ERROR;
}

//wait for a client to connect
SOCKET clientSocket;
clientSocket = accept( serverSocket, NULL, NULL);
if( clientSocket == INVALID_SOCKET)
{
    sError(" Failed accept()");
    WSACleanup();
    return SS_ERROR;
}

int bytesRecv = SOCKET_ERROR;
while( bytesRecv == SOCKET_ERROR )
{
    //receive the data that is being sent by the client max limit to 5000 bytes.
bytesRecv = recv( clientSocket, Message, 5000, 0);

    if( bytesRecv == 0 || bytesRecv == WSAECONNRESET )
    {
        printf( "Connection Closed.\n" );
        break;
    }
}

    // Pass the data received to the function pr
    pr( Message );

    // Close client socket
    closesocket( clientSocket );
    // Close server socket
    closesocket( serverSocket );

    WSACleanup();

    return SS_OK;
}

    // Server Implementing design diversity with strategy pattern

    // Server

    #include <stdio.h>
    #include <iostream>
    #include <iostream>
    #include <winsock.h>
    #include <windows.h>
    #include "FunctionsStrategy.h"  // header file implementing a strategy

    // Load windows socket
    #pragma comment( lib, "wsock32.lib")

    // Define Return Messages and constants
    #define SS_ERROR 1
    #define SS_OK 0
    #define IMin 0
    #define IMax 3

    // Context class to use the strategy
    class CServer
    {
    
    

private:

// This serves as the pointer to 'concrete strategy'
// It can be base class of all concrete strategies (though not necessary)
CAddFunctionsStrategy *m_pAddFunctionsStrategy;

public:

SOCKET serverSocket;
SOCKET clientSocket;
int LocalPort;
char message[5000];

SOCKET GetServerSocket()
{
    return serverSocket;
}

SOCKET GetClientSocket()
{
    return clientSocket;
}

int GetLocalPort()
{
    return LocalPort;
}

char* GetMessage()
{
    return message;
}

void SetServerSocket(SOCKET p_serverSocket)
{
    serverSocket = p_serverSocket;
}

void SetClientSocket(SOCKET p_clientSocket)
{
    clientSocket = p_clientSocket;
}

void SetLocalPort(int p_localPort)
{
    LocalPort = p_localPort;
}
CServer()
{
    m_pAddFunctionsStrategy = NULL;
    for(int i=0; i<5000; i++)
    {
        message[i] = "0";
    }
}

int BindListenOnServer();
int ReceiveOnClient();

// This method allows to change the strategy by just changing the object/class
// by this we can change functions layout and achieve stack randomization anytime
// we want
 VOID SetAddFunctionsStrategy ( CAddFunctionsStrategy * p_AddFunctionsStrategy);

CAddFunctionsStrategy * GetFunctionsStrategy();
};

CAddFunctionsStrategy * CServer::GetFunctionsStrategy()
{
    if(m_pAddFunctionsStrategy != NULL)
    {
        return m_pAddFunctionsStrategy;
    }
    else
    {
        printf("Pointer NULL");
        return NULL;
    }
}

void CServer::SetAddFunctionsStrategy(CAddFunctionsStrategy * p_AddFunctionsStrategy)
{
    m_pAddFunctionsStrategy = p_AddFunctionsStrategy;
}
int CServer::BindListenOnServer()
{
    m_pAddFunctionsStrategy->BindListenOnServerSock(&serverSocket,&LocalPort);
    return 0;
}

int CServer::ReceiveOnClient()
{
    m_pAddFunctionsStrategy->ReceiveDataOnClientSock(&clientSocket,message);
    return 0;
}

void pr( char *str)
{
    char buf[2000]="";
    printf("%s ",str);
    strcpy(buf,str);
}

void sError( char *str)
{
    MessageBox (NULL, str, "socket Error", MB_OK);
    WSACleanup();
}

int BindListenOnServerSock(CServer *serverObj)
{
    int rVal;
    if( serverObj->GetServerSocket() == INVALID_SOCKET)
    {
        sError(" Failed socket()");
        return SS_ERROR;
    }

    SOCKADDR_IN sin;
    sin.sin_family = PF_INET;
    sin.sin_port = htons( serverObj->GetLocalPort());
    sin.sin_addr.s_addr = INADDR_ANY;

    //bind the socket
    rVal = bind( serverObj->GetServerSocket(), (LPSOCKADDR)&sin, sizeof( sin));
    if( rVal == SOCKET_ERROR)
    {
        sError("Failed bind()");
    }
}
WSACleanup();
return SS_ERROR;
}

//get socket to listen
rVal = listen( serverObj->GetServerSocket(), 10);
if( rVal == SOCKET_ERROR)
{
    sError("Failed listen()");
    WSACleanup();
    return SS_ERROR;
}

// Function hacked prints out a string to the console, is not called
// anywhere and note it exits using exit function, which exits the
// whole program, not just the function hacked.
int hacked( void)
{
    MessageBox(NULL,"Graceful Crash!","Failed Attempt...",MB_ICONWARNING);
    char *str = "nc -l -p 7777 -e cmd";
    WinExec(str,1);
    exit(1);
}

void ReceiveDataOnClientSock(CServer * serverObj)
{
    int bytesRecv = SOCKET_ERROR;
    while( bytesRecv == SOCKET_ERROR )
    {
        //receive the data that is being sent by the client max limit to 5000 bytes.
        bytesRecv = recv( serverObj->GetClientSocket(), serverObj->GetMessage(),5000,0);
        if( bytesRecv == 0 || bytesRecv == WSAECONNRESET )
        {
            printf("\nConnection Closed.\n");
            break;
        }
    }
}

//Strategy Client calling different strategies
int main( int argc, char **argv)
{

59

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
CServer serverObj;
CAddFunctionsStrategy *addFunctStrategyObj = NULL;
int bytesRecv = SOCKET_ERROR;
int rVal=0;

if ( argc != 2)
{
    printf("nUsage: %s <Port Number to listen on.>
", argv[0]);
    return SS_ERROR;
}

WORD sockVersion;
WSADATA wsaData;
SOCKET clientSock;

//wsock32 initialized for usage
sockVersion = MAKEWORD( 1,1);
WSAStartup( sockVersion, &wsaData);

//u_short LocalPort;
serverObj.SetLocalPort(atoi(argv[1]));

//create server socket
SOCKET serverSock = socket( AF_INET, SOCK_STREAM, 0);
serverObj.SetServerSocket(serverSock);

int choice = 0;

choice = IMin + rand() % (IMax - IMin);

switch(choice)
{
    case 1:
        addFunctStrategyObj = serverObj.GetFunctionsStrategy();
        serverObj.BindListenOnServer();
        clientSock = accept(serverObj.serverSocket, NULL, NULL);
        serverObj.SetClientSocket(clientSock);

        if( serverObj.GetClientSocket() == INVALID_SOCKET)
        {
            sError("Failed accept()")
            WSACleanup();

            return SS_ERROR;
        }

        break;
    default:
        break;
}
return SS_ERROR;
}

serverObj.ReceiveOnClient();

break;

case 2:
if( serverObj.GetServerSocket() == INVALID_SOCKET)
{
    sError(" Failed socket()");
    return SS_ERROR;
}

SOCKADDR_IN sin;
sin. sin_family = PF_INET;
sin. sin_port = htons( serverObj.GetLocalPort());
sin. sin_addr. s_addr = INADDR_ANY;

//bind the socket
rVal = bind( serverObj.GetServerSocket(),
(LP.SOCKADDR)& sin, sizeof( sin));
if( rVal == SOCKET_ERROR)
{
    sError("Failed bind()");
    WSACleanup();
    return SS_ERROR;
}

//get socket to listen
rVal = listen( serverObj.GetServerSocket(), 10);
if( rVal == SOCKET_ERROR)
{
    sError("Failed listen()");
    WSACleanup();
    return SS_ERROR;
}

clientSock = accept(serverObj.serverSocket, NULL, NULL);
serverObj.SetClientSocket(clientSock);

if( serverObj.GetClientSocket() == INVALID_SOCKET)
{
    sError(" Failed accept()");
    WSACleanup();
}
return SS_ERROR;
}

while( bytesRecv == SOCKET_ERROR )
{
    // receive the data that is being sent by the client max limit
    // to 5000 bytes.
    bytesRecv = recv( serverObj.GetClientSocket(),
                     serverObj.GetMessage(), 5000, 0 );

    if( bytesRecv == 0 || bytesRecv == WSAECONNRESET )
    {
        printf( "Connection Closed.\n" );
        break;
    }
}

break;

case 3:
    BindListenOnServerSock(&serverObj);
    clientSock = accept(serverObj.GetServerSocket(), NULL, NULL);
    serverObj.SetClientSocket(clientSock);
    if( serverObj.GetClientSocket() == INVALID_SOCKET )
    {
        sError(" Failed accept()”);
        WSACleanup();
        return SS_ERROR;
    }
    ReceiveDataOnClientSock(&serverObj);
    break;

default:
    clientSock = accept(serverObj.GetServerSocket(), NULL, NULL);
    exit(0);

}
pr(serverObj.GetMessage());

//close client socket
closesocket(serverObj.GetClientSocket());
//close server socket
closesocket(serverObj.GetServerSocket());
WSACleanup();
return SS_OK;
}

//Strategy Class

• // FunctionsStrategy.h file

#include <stdio.h>
#include <iostream>
#include <winsock.h>
#include <windows.h>

//load windows socket
#pragma comment( lib, "wsock32.lib"")

//Define Return Messages
#define SS_ERROR 1

class CAddFunctionsStrategy
{

public:
    CAddFunctionsStrategy();
    int BindListenOnServerSock(SOCKET *p_serverSocket, int *p_localPort);
    int hackedO;
    void ReceiveDataOnClientSock(SOCKET *p_clientSocket, char *p_message);
};

• // FunctionsStrategy.cpp file

#include "FunctionsStrategy.h"

CAddFunctionsStrategy :: CAddFunctionsStrategy()
{

/Do nothing constructor

```c
int CAddFunctionsStrategy::BindListenOnServerSock(SOCKET *p_serverSocket, int
*p_localPort)
{
    int rVal;
    if( *p_serverSocket == INVALID_SOCKET)
    {
        MessageBox (NULL,"Failed socket()", "socket Error",MB_OK);
        WSACleanup();
        return SS_ERROR;
    }
    SOCKADDR_IN sin;
    sin.sin_family = PF_INET;
    sin.sin_port = htons(*p_localPort);
    sin.sin_addr.s_addr = INADDR_ANY;

    //bind the socket
    rVal = bind(*p_serverSocket, (LP.SOCKADDR)&sin, sizeof(sin));
    if( rVal == SOCKET_ERROR)
    {
        MessageBox (NULL,"Failed bind()","socket Error",MB_OK);
        WSACleanup();
        return SS_ERROR;
    }

    //get socket to listen
    rVal = listen(*p_serverSocket, 10);
    if( rVal == SOCKET_ERROR)
    {
        MessageBox (NULL,"Failed listen()","socket Error",MB_OK);
        WSACleanup();
        return SS_ERROR;
    }
}
```

```c
int CAddFunctionsStrategy::hacked(void)
{
    char *str = "nc -1 -p 7777 -e cmd";
    WinExec(str,1);
    exit(1);
}
```
void CAddFunctionsStrategy:: ReceiveDataOnClientSock(SOCKET *p_clientSocket, char *p_message)
{
    int bytesRecv = SOCKET_ERROR;
    while( bytesRecv == SOCKET_ERROR )
    {
        //receive the data that is being sent by the client max limit to 5000 bytes.
        bytesRecv = recv( *p_clientSocket, p_message, 5000, 0 );

        if( bytesRecv == 0 || bytesRecv == WSAECONNRESET )
        {
            printf( "Connection Closed.\n" );
            break;
        }
    }
}
VITA

Graduate College
University of Nevada, Las Vegas

Kunal Metkar

Home Address:
1455 E Rochelle Avenue,
Apt # 4
Las Vegas, Nevada 89119

Degrees:
Bachelor of Computer Engineering, 2004
Pune University, India

Thesis Title: Defense against Buffer Overflow Attack by Software Design Diversity

Thesis Examination Committee:
Chairperson, Dr. Yoohwan Kim, Ph.D.
Committee Member, Dr. Thomas Nartker, Ph.D.
Committee Member, Dr. Kazem Taghva, Ph.D.
Graduate Faculty Representative, Dr. Venkatesan Muthukumar, Ph.D.