



**Calhoun: The NPS Institutional Archive**  
**DSpace Repository**

---

Theses and Dissertations

1. Thesis and Dissertation Collection, all items

---

2019-09

# DEFENSIVE BINARY CODE INSTRUMENTATION TO PROTECT AGAINST BUFFER OVERFLOW ATTACKS

Rogers, Alexis L.; Sowers, Ryan

Monterey, CA; Naval Postgraduate School

---

<http://hdl.handle.net/10945/63497>

*Downloaded from NPS Archive: Calhoun*



Calhoun is a project of the Dudley Knox Library at NPS, furthering the precepts and goals of open government and government transparency. All information contained herein has been approved for release by the NPS Public Affairs Officer.

**Dudley Knox Library / Naval Postgraduate School**  
**411 Dyer Road / 1 University Circle**  
**Monterey, California USA 93943**

<http://www.nps.edu/library>



# NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

## THESIS

**DEFENSIVE BINARY CODE INSTRUMENTATION TO  
PROTECT AGAINST BUFFER OVERFLOW ATTACKS**

by

Alexis L. Rogers and Ryan Sowers

September 2019

Thesis Advisor:  
Second Reader:

Doron Drusinsky  
James B. Michael

**Approved for public release. Distribution is unlimited.**

THIS PAGE INTENTIONALLY LEFT BLANK

<b>REPORT DOCUMENTATION PAGE</b>			<i>Form Approved OMB No. 0704-0188</i>	
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington, DC 20503.				
<b>1. AGENCY USE ONLY</b> (Leave blank)		<b>2. REPORT DATE</b> September 2019		<b>3. REPORT TYPE AND DATES COVERED</b> Master's thesis
<b>4. TITLE AND SUBTITLE</b> DEFENSIVE BINARY CODE INSTRUMENTATION TO PROTECT AGAINST BUFFER OVERFLOW ATTACKS			<b>5. FUNDING NUMBERS</b>	
<b>6. AUTHOR(S)</b> Alexis L. Rogers and Ryan Sowers				
<b>7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)</b> Naval Postgraduate School Monterey, CA 93943-5000			<b>8. PERFORMING ORGANIZATION REPORT NUMBER</b>	
<b>9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)</b> N/A			<b>10. SPONSORING / MONITORING AGENCY REPORT NUMBER</b>	
<b>11. SUPPLEMENTARY NOTES</b> The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.				
<b>12a. DISTRIBUTION / AVAILABILITY STATEMENT</b> Approved for public release. Distribution is unlimited.			<b>12b. DISTRIBUTION CODE</b> A	
<b>13. ABSTRACT (maximum 200 words)</b>  Buffer overflow and heap overflow injection attacks have been studied for some time. Recent techniques to prevent execution of payloads inserted into memory have been successful by using stack canaries, non-executable stacks, and address space layout randomization (ASLR). Attackers now use a technique called return oriented programming to maliciously execute code without ever inserting such a payload into memory. They do so by identifying binary snippets in the original program that constitute a malicious procedure. There have been patches in place to help decrease the susceptibility to this type of attack, but what is needed is a permanent fix. We propose such a solution, applied directly to the compiled binary, consisting of a masking function to obfuscate the return address, an unmasking function (i.e., reversing the previous), and instrumenting code to perform these functions seamlessly. We implemented a proof of concept of our solution that prevents an unexpected address jump via binary return address obfuscation.				
<b>14. SUBJECT TERMS</b> buffer overflow, return oriented programming, ROP, gadget, return instruction pointer, stack canary, obfuscation, binary			<b>15. NUMBER OF PAGES</b> 75	
			<b>16. PRICE CODE</b>	
<b>17. SECURITY CLASSIFICATION OF REPORT</b> Unclassified	<b>18. SECURITY CLASSIFICATION OF THIS PAGE</b> Unclassified	<b>19. SECURITY CLASSIFICATION OF ABSTRACT</b> Unclassified	<b>20. LIMITATION OF ABSTRACT</b> UU	

THIS PAGE INTENTIONALLY LEFT BLANK

**Approved for public release. Distribution is unlimited.**

**DEFENSIVE BINARY CODE INSTRUMENTATION TO PROTECT AGAINST  
BUFFER OVERFLOW ATTACKS**

Alexis L. Rogers  
Civilian, Scholarship for Service  
BS, University of Washington, 2014

Ryan Sowers  
Civilian, Scholarship for Service  
BS, University of California, Merced, 2012

Submitted in partial fulfillment of the  
requirements for the degree of

**MASTER OF SCIENCE IN COMPUTER SCIENCE**

from the

**NAVAL POSTGRADUATE SCHOOL  
September 2019**

Approved by: Doron Drusinsky  
Advisor

James B. Michael  
Second Reader

Peter J. Denning  
Chair, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

## ABSTRACT

Buffer overflow and heap overflow injection attacks have been studied for some time. Recent techniques to prevent execution of payloads inserted into memory have been successful by using stack canaries, non-executable stacks, and address space layout randomization (ASLR). Attackers now use a technique called return oriented programming to maliciously execute code without ever inserting such a payload into memory. They do so by identifying binary snippets in the original program that constitute a malicious procedure. There have been patches in place to help decrease the susceptibility to this type of attack, but what is needed is a permanent fix. We propose such a solution, applied directly to the compiled binary, consisting of a masking function to obfuscate the return address, an unmasking function (i.e., reversing the previous), and instrumenting code to perform these functions seamlessly. We implemented a proof of concept of our solution that prevents an unexpected address jump via binary return address obfuscation.



THIS PAGE INTENTIONALLY LEFT BLANK

## TABLE OF CONTENTS

<b>I.</b>	<b>BACKGROUND .....</b>	<b>1</b>
<b>A.</b>	<b>PROGRAM VULNERABILITIES .....</b>	<b>1</b>
<b>B.</b>	<b>WHAT IS BUFFER OVERFLOW .....</b>	<b>2</b>
<b>C.</b>	<b>BUFFER OVERFLOW MITIGATIONS.....</b>	<b>6</b>
<b>D.</b>	<b>RELATED WORK.....</b>	<b>8</b>
<b>II.</b>	<b>MOTIVATION .....</b>	<b>11</b>
<b>A.</b>	<b>BUFFER OVERFLOW PERSISTENCE .....</b>	<b>11</b>
<b>B.</b>	<b>RESEARCH QUESTIONS.....</b>	<b>13</b>
<b>III.</b>	<b>RESEARCH METHODOLOGY .....</b>	<b>15</b>
<b>A.</b>	<b>ADDITIONAL BUFFER OVERFLOW PROTECTION.....</b>	<b>15</b>
<b>B.</b>	<b>EXECUTABLE STRUCTURE .....</b>	<b>16</b>
<b>C.</b>	<b>TOOLS USED .....</b>	<b>18</b>
<b>D.</b>	<b>INSTRUCTION MODIFICATIONS.....</b>	<b>19</b>
<b>E.</b>	<b>MASKING AND UNMASKING FUNCTIONS .....</b>	<b>22</b>
<b>F.</b>	<b>REDIRECTING PROGRAM FLOW.....</b>	<b>24</b>
<b>G.</b>	<b>ADDRESS MODIFICATIONS .....</b>	<b>26</b>
<b>H.</b>	<b>BINARY EXTENSION .....</b>	<b>29</b>
<b>I.</b>	<b>CODE HIDING.....</b>	<b>30</b>
<b>IV.</b>	<b>RESULTS .....</b>	<b>33</b>
<b>V.</b>	<b>CONCLUSION .....</b>	<b>35</b>
<b>A.</b>	<b>SUMMARY .....</b>	<b>35</b>
<b>B.</b>	<b>DEPLOYMENT .....</b>	<b>35</b>
<b>C.</b>	<b>ADAPTING BEYOND ELF .....</b>	<b>36</b>
<b>D.</b>	<b>FUTURE WORK.....</b>	<b>37</b>

<b>APPENDIX A. MMAP EDITOR .....</b>	<b>39</b>
<b>APPENDIX B. INSTRUCTION INSERTION.....</b>	<b>41</b>
<b>APPENDIX C. UHAUL.....</b>	<b>43</b>
<b>APPENDIX D. EXTEND AND PATCH.....</b>	<b>49</b>
<b>APPENDIX E. CODE HIDER.....</b>	<b>51</b>
<b>LIST OF REFERENCES .....</b>	<b>53</b>
<b>INITIAL DISTRIBUTION LIST .....</b>	<b>57</b>

## LIST OF FIGURES

Figure 1.	Fgets example .....	2
Figure 2.	Example stack layout .....	3
Figure 3.	How to overflow the stack buffer .....	4
Figure 4.	Format string example .....	5
Figure 5.	String copy example .....	11
Figure 6.	Insert_instructions function .....	20
Figure 7.	Replace_instructions function.....	21
Figure 8.	Delete_extra_bytes function .....	22
Figure 9.	Masking function, me1 .....	23
Figure 10.	Unmasking function, me2 .....	24
Figure 11.	Caller redirection .....	25
Figure 12.	Callee redirection .....	25
Figure 13.	Redirected program flow .....	26
Figure 14.	Addressing changes .....	27
Figure 15.	Indirect addressing.....	28
Figure 16.	ELF binary structure .....	29
Figure 17.	Breakpoint set .....	33

THIS PAGE INTENTIONALLY LEFT BLANK

## LIST OF ACRONYMS AND ABBREVIATIONS

ASCII	American Standard Code for Information Interchange
ASLR	address space layout randomization
BTB	branch target buffer
CPU	central processing unit
CVE	Common Vulnerability Enumerations
CVSS	Common Vulnerability Scoring System
CWE	Common Weakness Enumeration
DIFT	Dynamic Information Flow Tracking
ROP	return oriented programming
ELF	executable and linkable format
GDB	GNU Project Debugger
GNOME	GNU Network Object Model Environment
HMI	human-machine interface
ICS	industrial control system
IDA	Interactive Disassembler
I/O	input/output
Mach-O	Macintosh Operating System File Format
NOP	no operation
NX	no execution
OS	operating system
PE	portable executable
SCADA	supervisory control and data acquisition
W^X	write xor execute

THIS PAGE INTENTIONALLY LEFT BLANK

## **DISCLAIMER**

This material is based on work supported by the National Science Foundation under Grant No. 1565443. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.



THIS PAGE INTENTIONALLY LEFT BLANK

## ACKNOWLEDGEMENTS

We would like to thank our advisor, Dr. Doron Drusinsky, for providing us with this research topic as well as the extended nano binary that was integral in performing the techniques carried out in this research.

We would also like to thank our second reader and professor, Dr. James Bret Michael, for his guidance and encouragement in this process.

There were a couple of professors throughout our education here at NPS that provided key insights and foundational knowledge on the topics that are addressed in this research. We would like to thank Chris Eagle for providing us with a detailed understanding of low-level programming and the ways vulnerabilities are exploited on this level. We would also like to thank Kevin Wyatt, visiting lecturer, for giving us perspective on how our research relates to Windows executables and how it can be more broadly applied.

This research (and the background knowledge required to conduct it) was made possible by a grant provided by the National Science Foundation and supported by the Office of Personnel Management. We are grateful to them for providing this educational opportunity. We would like to specifically thank Dr. Cynthia Irvine and Dr. Duane Davis for allowing us the opportunity to take part in the Scholarship for Service program at NPS.

THIS PAGE INTENTIONALLY LEFT BLANK

# **I. BACKGROUND**

## **A. PROGRAM VULNERABILITIES**

Originally designed to require fewer lines of kernel code, the C language is still a top contender on the IEEE Spectrum of most popular programming languages [1]. C comes in at number four, yielding its ranks only to that of Java, Python, and C++, which all, at some level, are higher-level languages built upon C. This particular ranking is one of Internet popularity, which shows how it continues to be the go-to for future programs and applications. In addition, legacy programs designed for the Microsoft Windows Operating System (holding 90% of the market share [2]), MacOS, Linux, and their respective mobile device operating systems are programmed in C. This not only demonstrates C's ubiquity, but also shows the impact a C-specific vulnerability can have if exploited. The potential impact in terms of frequency and severity could be enormous considering that numerous safety and mission-critical systems, including Internet of Things applications, are implemented in C.

With the C language's minimal abstraction from machine-specific code, it is highly efficient. With C's own compiler handling the machine code generation, portability (independence from computer type) is achieved, thus cutting down on unnecessary steps and time. In addition, there is "at least one compiler for almost every architecture" [2], unlike assembly languages, which are processor-specific [3].

The C Standard library is a repository of built-in macros and functions that allow a programmer to utilize pre-built functions within their code [4]. For example, without the addition of this library, instead of using a one-liner function like `printf("Hello world")`, you would have to write directly in assembly, with the attendant expansion factor in number of lines of code. Common occurrences like opening files, gathering input, or calculating string lengths are simplified and standardized by this library.

Continual updates to the C standard library have allowed the C language to become more secure with each new implementation. Most specifically, in regard to this

research, a pivotal update was seen in the 2007 C11 Edition [5]. The C11 Edition increased the library by including functions like `fgets()` and `strcat_s()`, among the others which end with ‘\_s’ [6]. These new functions impose restrictions on input, giving programmers a much-needed protection against data overrun vulnerabilities [4]. Figure 1 is an example of the newly added bounds checking functionality seen within `fgets()`. This function adds security to the discontinued `gets()` function it replaced (also seen in Figure 1).

```
char *fgets(char *str, int n, FILE *stream)

char *gets(char *str)
```

Figure 1. Fgets example

The `fgets()` function has been corrected from `gets()` to take additional arguments that support specifying the amount (`int n`) of characters to gather from the user. Any additional characters entered will not be recognized. Though an improvement to the previously vulnerable library functions, these additions do not provide complete safety. Between the numerous buffers that can exist between the underlying function and the input device, incorrect specifications in the `int n` length, and the existence of unprotected legacy code, vulnerabilities still exist.

## B. WHAT IS BUFFER OVERFLOW

There are C library functions (e.g., `gets()`, `strcpy()`, or `syslogd()`) that operate on input without restriction, or bounds checking. These allow more data to be taken in by the program than intended. Excess input can lead to the overwriting of existing instructions and/or crashing the application all together. The main impetus for the C11 addition to the library was to create a stopping point, or bound, where no more input can be written to the memory space provided by the program. This is a sanitary means to control how much data can be collected and where that data will be stored.

When a computer program requires user input, it creates memory space to store that information. As with any function variable, the memory space, or buffer, is inserted on the stack, as seen in Figure 2. This is placed above other functional data that tell the program what location to jump back or return to after it has finished the task.

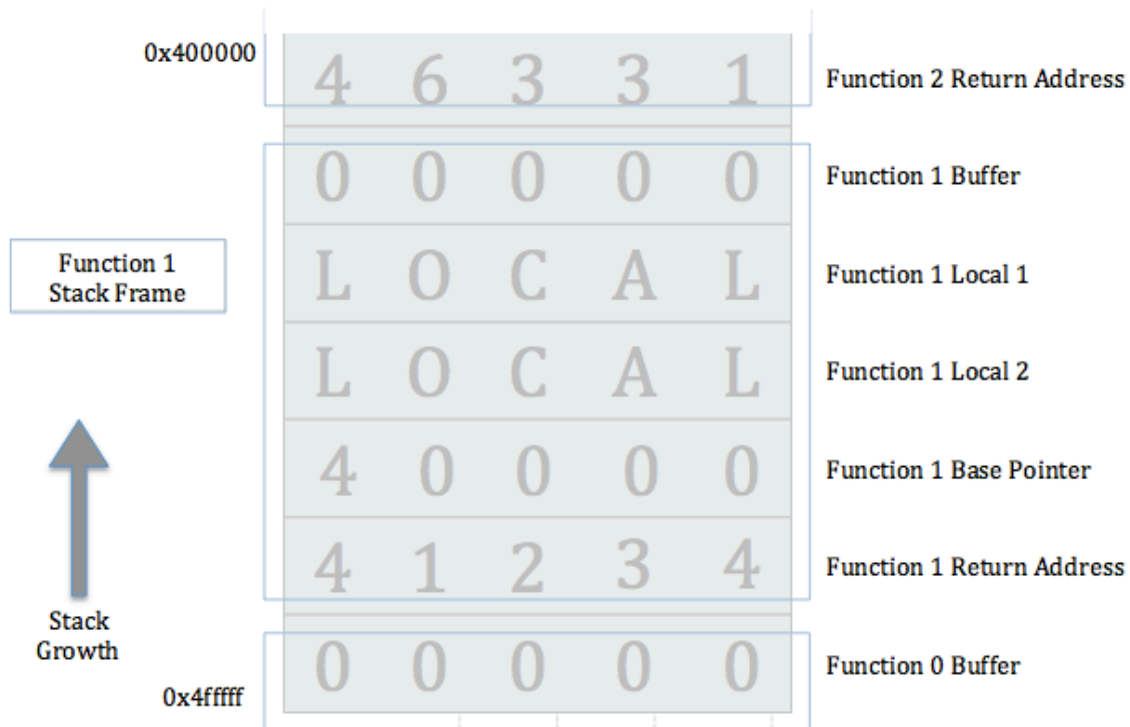


Figure 2. Example stack layout

When a user enters more data than the buffer can store, memory addresses following the buffer will be overwritten with some of the input. When a malicious attacker plans to deliberately overflow an input buffer, he can fill the buffer with instructions or addresses of his choosing. As seen in Figure 3, when a buffer is overfilled, it can overwrite the return address. This results in control of the system running the program being turned over to the location the user specified.



Figure 3. How to overflow the stack buffer

There are multiple possible objectives for carrying out a buffer overflow attack, all of which described here concern overwriting the return address. From the very beginning it was clearly seen that, “By supplying addresses outside the space allocated to the user’s programs, it is often possible to get the monitor to obtain unauthorized data for that user, or at the very least, generate a set of conditions in the monitor that causes a system crash” [7]. In its simplest form, the malicious entry of an invalid address will crash the application. Yet, for more advanced malicious intents, an attacker can overflow a buffer with a return address as well as malicious payload code. By injecting such malicious code as well as overwriting the return address to point to the start of those instructions, the attacker has caused the application to execute in the manner he has chosen, gaining control over it.

A common type of buffer overflow attack, which works when nonexecutable stack protection is enabled, is Return Oriented Programming (ROP) [8]. Utilizing snippets of instruction, or gadgets, already present in the program, the attacker simply rewrites the return address to point to an individually selected gadget, achieving the

desired functionality of the attack as he traverses the sequence of instructions. The stringing of these gadgets together results in the attacker gaining control of the system. In this case, the attacker is not executing his own provided payload code, as is done in typical code injection attacks; the attacker is relying on already existing instructions to perform malicious activities. This is similar to code injection, yet without the overhead.

Another type of buffer overflow attack takes advantage of a vulnerability in how output is formatted by the `printf()` function. Utilization of this library function is shown in Figure 4.

```
#include <stdio.h>

int main(){
    char my_string[] = "world";
    printf("Hello %s \n", my_string);          //Example 1
    printf("Hello %s %s %x %x", my_string);    //Example 2
}
```

```
Output$ Hello world
        Hello LOCAL LOCAL 40000 41234          //Example 1
                                                //Example 2
```

Figure 4. Format string example

The `printf()` function prints what is between the quotes, as characters, to standard out. The `%` symbol is a signal to check what follows the closing quotes for an argument to put in its place. In this example, it will print the argument pushed onto the stack at runtime. This function has the ability to take a variable number of arguments with at least one being mandatory [9]. If no following arguments were entered, the above function would still go to the stack to retrieve the topmost value and print the type specified after the `%` symbol. In this instance, the `"c"` represents printing the data as a character. Numerous types can be specified. When a pointer is the specified type, the processor reads the stack value as an address of where to retrieve data from. This type of



attack takes advantage of two flaws. By simply using numerous % symbols, an attacker can leak an infinite amount of data from the stack. This includes return addresses and other stack data, as well as the input the user writes into the function's buffer. This is the key to the second flaw. Because an attacker can write to the stack, he can put an address into the buffer. He can then have the rest of the input structured to jump there. The attacker can also exploit production code in unconventional ways, such as when the format string is programmatically set. After all, the format string is just a string, so it too could be in a buffer.

### **C. BUFFER OVERFLOW MITIGATIONS**

The concept of a buffer overflow has been around for nearly fifty years since the U.S. Air Force [7] published the "Computer Security Technology Planning Study" in October 1972. This paper outlined several different classes of computer attacks, one of which was titled, "Incomplete Parameter Checking." This attack described a technique in which an attacker could provide an illegal program address as input and then redirect execution to that address. This was possible because there was no parameter validation. This may be the first development of the idea of gaining control of a program for unintended use via an injection.

Just over twenty years later, there was a paper published in *Phrack* magazine called, "Smashing the Stack For Fun and Profit" [10]. In this paper, computer scientist Elias Levy (known by the handle, "Aleph One") defined the term "smash the stack" as being able to "corrupt the execution stack by writing past the end of an array declared auto in a routine" [10]. This is one of the reasons the modern-day buffer overflow exploit came into the public's eye and began to be heavily analyzed and researched.

Due to the fact that this concept has been widely studied for many years, there have been many fixes at the programming language and operating systems levels to address buffer overflows. Some of these fixes include address space layout randomization (ASLR), executable stack control, and stack canaries. Each of these fixes addresses a different aspect of the vulnerabilities that can lead to buffer overflow attacks. These

techniques of buffer overflow prevention will be briefly described and then will be followed with an explanation of how and why buffer overflow can still be a concern.

ASLR is an operating-system protection against buffer overflow attacks. It works by ensuring that program code is loaded into a different section of memory every time it is executed, as well as providing randomized sections of memory for the stack, heap, and libraries. In the case where ASLR is implemented, when an attacker attempts to overwrite the return address of a function on the stack, he must also know the memory location he needs to point to in order to execute his code, which was dynamically determined at runtime. Unfortunately, there are ways of determining these addresses. One way is by leaking memory locations with format string exploits. Other methods leak information via side channels. In 2016, researchers from the State University of New York and University of California, Riverside [11] published a means of subverting ASLR by using a side-channel attack on the branch target buffer (BTB). A second way is by utilizing a no-operation (“nop”) slide/sled, which is a sequence of no-operation instructions that the machine will recognize but that will not execute any functionality. This allows the attacker to land in a range of addresses (where the “nops” exist), decreasing how specific he must be in his return address selection. Following the “nop” sled would be the attacker’s code or a jump to such code. The attacker would still need to know what range of memory the code exists in (possibly via a memory leak) in order to choose a return address, but the precision required is decreased.

A compiler-level defense against buffer overflow attacks is requiring that areas of memory not be writable and executable. One of these features in existence today is W<sup>X</sup>, for “write XOR execute,” which utilizes an NX bit, for “No eXecute.” This allows entire pages of memory to be writeable or executable but not both. Consider the following situation where this would be useful: An attacker writes code to the stack, in a dynamically allocated buffer. With the NX bit on, the attacker would not be allowed to execute such code, defeating his attempt to take control of the program. A rising workaround for non-executable stacks is ROP. Alternatively, the attacker could place his code in an unprotected region of memory, such as the heap, and cause it to execute from there.

Another compiler-level buffer overflow prevention technique is the use of stack canaries or cookies. A canary is a random value placed onto the stack after (by order of operation) a return address and before a dynamically allocated buffer in a function. This value is placed there by the operating system at runtime and is verified before the function returns. This value functions to prevent an attacker from arbitrarily overflowing the buffer, overwriting the canary, and overwriting the return address. For example, an attacker would overflow a buffer in order to overwrite the return address that exists below it on the stack. If a stack canary is utilized, the attacker will inadvertently overwrite this value as well. When the program goes to return using the return address on the stack (that the attacker has now replaced), it will attempt to verify the value that is located where it expects the stack canary to be. The attacker has overwritten this with his own code or junk instructions and, therefore, it will not be successfully verified. The program will then crash or exit. Unfortunately, these stack-canary values can also be leaked or even brute-forced under certain circumstances. For example, when a process forks, the child process inherits the same canary. This situation can be exploited to leak information about the canary and/or to brute-force it. The attacker can try different values in the child process. If he fails, then a new child process is forked and a new value is tried. This process continues until the child stops crashing. If the value can be obtained by the attacker, then it can be replaced in its original position and the program will not detect any corruption. The attacker proceeds with overwriting the return address following the canary.

#### **D. RELATED WORK**

In December 2012, Wartell et al. [12] described a method of securing untrusted code by utilizing binary rewriting. The researchers performed rewriting of untrusted executables in order to guarantee their safety by enforcing security policies and ensuring control flow integrity. This work was done on applications that run on 32-bit versions of Microsoft Windows XP/Vista/7/8. Of note, the rewriting tool that Wartell et al. have designed (REINS) does not prevent against ROP attacks by protecting the stack pointer or return address, but rather monitors system calls, which is a more complex task. In contrast, our binary instrumentation technique is done on trusted binaries in order to

make them more secure against attack. Although it is at present crafted for ELF binaries on 64-bit Linux, it can be adapted to other operating systems and binary formats.

Another paper [13] of interest to this research looks for a more optimal implementation of stack cookies. Stack cookies and canaries are of interest because of their seemingly similar functionality to our tool. The issue with cookies and canaries is their vulnerability to format string attack. Contrary to this, because of the XOR masking function our tool utilizes (described in Chapter III), even when this canary value is leaked, the attacker's return address will not work. In efforts to realize the reach of our tool, we found a paper [14] detailing the vulnerabilities with Android mobile devices. These devices are based upon a Linux kernel and the C library for the embedded devices. To rectify the insufficient security mechanisms in canaries, it was proposed to XOR canaries [14]. This has been successful in applications like Visual Studio and hardened versions of Linux despite the large overhead. The security and continued use of an XOR canary are promising nudges that the basis of our tool is on the right track. This is especially true when factoring in the added advantage our tool gains by not needing to store the reference canary in secure memory space. Utilizing the built-in exception handling of the processor deters the need for our function to save a comparison value.

There has also been work done on the implications of binary obfuscation. Joshi et al. [15] claim that the susceptibility of programs to ROP-based attacks may increase in software that uses code obfuscation techniques. These techniques described in the paper perform significant changes to program instructions, adding and modifying many opcodes. We do not believe that our method of return address obfuscation and code additions will increase the likelihood of ROP-based attacks. Our tool effectively only adds two possible ROP gadget instructions: a jump at the end of the *me1* function and a return at the end of the *me2* function (as described in Chapter III, Section E). The modifications made to redirect program flow (described in Chapter III, Section F) change the destination address of one call (no opcode change) and change a return instruction to a jump (insignificant difference).

Other research similar to ours centers on a technique called PointGuard, a buffer overflow protection applied to pointers [16]. This technique takes vulnerable pointers and

passes them through an encryption algorithm before storing them. When the pointers need to be used, they are passed through a decryption algorithm. If an attacker overwrites the pointer, it will be decrypted and will result in a pointer to an invalid instruction causing the program to crash. Our research can be considered an alternative to this technique in the following ways. First, our tool is implemented at the binary level as opposed to an intermediate compilation stage. This affords us the flexibility to place our instructions at any location in the binary. It can also be applied to already compiled and existing programs. Thus, our modifications are not subject to the compiler's control. Compilers have techniques they use to optimize as well as handle "overflow" which may result in unexpected changes to the implementation of the desired security mechanisms. Second, in the PointGuard technique, register spillage may reveal unencrypted pointer values [16]. By manipulating the binary directly, we are able to ensure that the deobfuscated return address remains outside of the scope of a vulnerable function.

Research into mitigating buffer overflows continues to grow. With this quantity of research, it may be hard to imagine why buffer overflows remain one of the top attacks performed today. We attribute a large portion of their ubiquity not only to bad coding practices and legacy software use, but also to education. *Hacking: The Art of Exploitation* [17] is a book that details all the steps necessary to carry out a buffer overflow attack. This textbook is commonly required at many educational facilities as part of a Computer Science curriculum. Not only does this book highlight how to carry out a buffer overflow attack, but also it details methods involving code injection, like format string exploits and various types of shell code. We are seeing buffer overflow attacks persist due to the number of people who can perform them. With exploitation techniques taught in educational facilities, the volume of people who can discover and maliciously exploit buffer overflows continues to grow, contributing greatly to their persistence.

## II. MOTIVATION

### A. BUFFER OVERFLOW PERSISTENCE

Despite integration of prevention techniques, buffer overflows have not been eradicated. One reason for their persistence is that, even after a programmer has enabled bounds checking to help prevent buffer overflow, he may misunderstand how input is handled thus still allowing vulnerabilities. Consider the code segment in Figure 5. When using a function like `strncpy()`, the programmer must account for the input being null terminated, otherwise the code is left open to vulnerabilities.

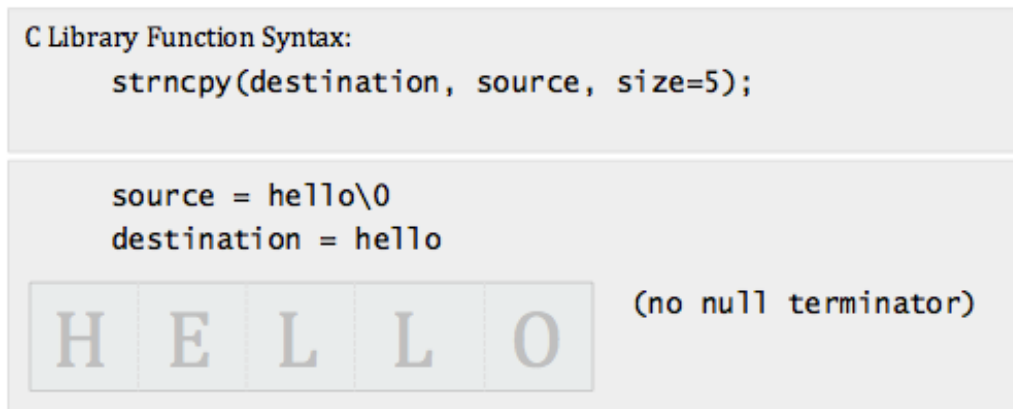


Figure 5. String copy example

More scrupulous coding practices, keener understanding of how input is handled, and other mitigations mentioned above are key means for combatting buffer overflows. Unfortunately, even with these practices employed within current code, many older devices contain outdated code. Part of the persistence of buffer overflows can be attributed to C's longevity; legacy applications utilize vulnerable functions. This is a lead factor in the industrial control system issues [18]. Because of the new trend of networking these systems, their exploitation has been on the rise over the past decade. SCADA, HMI, and controllers all fall victim to even the most trivial programming errors.

In 2015, Kaspersky Lab [18] did a review of industrial control system (ICS) vulnerabilities worldwide. Of the 189 vulnerabilities reported in ICS components that year, “the most widespread types are buffer overflows,” accounting for 9% of all detected vulnerabilities. Of these 17 vulnerabilities, more than half of them (8) were rated as high risk according to the Common Vulnerability Scoring System (CVSS) and half of those (4) had the highest risk level of 10. These software systems either do not take advantage of modern buffer overflow prevention techniques, or they do have protections but they are subverted by hackers via attack vectors, such as those previously described. Of note, at least a half dozen software weaknesses that can lead to buffer overflows are listed in MITRE’s Common Weakness Enumeration (CWE) database [19]. From these weaknesses, stem hundreds of fully realized buffer overflow vulnerabilities as reported in the Common Vulnerability Enumerations (CVE) database [20] in the past year alone. This translates to potentially thousands in the history of buffer overflows. (This CVE database, maintained by MITRE, contains all publicly known and reported software vulnerabilities.) Many of these vulnerabilities have been patched and removed with software updates, but this gives an idea of how commonplace they are and how they might remain in software that is no longer tended to.

There are alternatives to programming in C, such as using the Java and Perl languages. They are not susceptible to buffer overflows because they implement inherent bounds checking through their use of arrays. In these languages, there are checks for abnormal conditions (i.e., exceptions) [21], which monitor data as it is put into memory. In these cases, bounds are set before memory space is allocated. If entry into this space is exceeded, the `ArrayIndexOutOfBoundsException` error is thrown. To allow C this type of security, these exceptions can be built within the compiler. Yet, these are not widely adopted due to the increase in overhead incurred [6]. When handling large amounts of data, these inefficiencies have great effect. Therefore, instead of switching languages, novel C-specific safeguards need to be created. In this paper, we hope to provide just that, a robust, efficient guard against buffer overflows.

## **B. RESEARCH QUESTIONS**

Buffer overflow and heap overflow injection attacks are nothing new. A classical buffer overflow attack operates in two phases: (i) attack a function's return address on the stack, and (ii) insert a malicious payload for the program to "return" to, instead of the original return address. Modern protections have mostly addressed the second phase by disallowing code to execute from the stack. Nevertheless, phase (i) is still used as the preamble to other attacks. This thesis addresses techniques for foiling such attacks. The thesis explores the following questions:

1. Can program code be instrumented in a way that prevents attackers from being able to take advantage of buffer overflows with the goal of taking control of a program?
2. Can this instrumentation be transparent to the attacker (i.e., added to the binary code instead of the source code)?
3. Can masking and unmasking functions be implemented in order to ensure that only the proper return addresses are being followed?



THIS PAGE INTENTIONALLY LEFT BLANK

### III. RESEARCH METHODOLOGY

#### A. ADDITIONAL BUFFER OVERFLOW PROTECTION

This brings us to our work on return address obfuscation. By going after the return address in our research, we are attempting to protect the very thing that buffer-overflow-based ROP attacks attempt to achieve: return address overwrites. This helps protect against attackers taking over program control flow by going after the root cause: the pointer to malicious code. In a fully-fledged version of our proof of concept presented here, all return addresses would be protected via redirections to masking and unmasking functions when they are placed onto and removed from the stack during execution. The full implementation is discussed in Chapter III, Section C. We will address each of the previously mentioned buffer overflow prevention techniques in order to show how our technique adds protection on top of them.

We first address ASLR. If the return address is protected, then there is no concern about ASLR being subverted. Even if the attacker discovers the addressing scheme used via a memory leak and/or is able to utilize a large enough “nop” sled to slide into his code, the final return address overwrite will fail. Before the function returns, the attacker-modified return address will be passed through the unmask function we have added and will subsequently be altered to a value that the attacker did not intend. That will likely cause the program to crash.

Secondly, with our technique there is no concern about the areas of memory being writable and/or executable because the attacker will never get to the code he wants to execute. By ensuring that any attacker-initiated return address modification is unable to be followed, we have preempted the safety check that W<sup>X</sup> provides. This also prevents the attacker from being able to initiate execution of a series of ROP gadgets. He is prevented from making that initial jump from his inserted return address. The same prevention applies to an attempt to jump to and execute heap memory instructions. The initial return address overwrite is prevented from being followed.

Finally, there is no concern about a stack canary getting leaked or brute-forced and re-written, because the modified return address will fail anyway. If the attacker successfully determines the stack canary and is able to replace it, followed by inserting his own return address, that return address will proceed through our unmasking function and will result in an address that he did not intend as well as an address that will likely crash the program.

This is not to say that our concept of return address obfuscation should necessarily be used to replace these features, but rather that it is an added element to protect program flow. In contrast to the compiler- and OS-based buffer overflow protections, our obfuscation technique is applied at the binary level. This means it can be used by anyone on any binary. It acts as a final fail-safe to these three previously mentioned attack prevention techniques. Our method of program protection is also made possible by the fact that our technique to protect the return address occurs outside of the function itself. Program flow is redirected to an extended part of the program code that was non-existent in the originally compiled program. The attacker cannot see that the return address is being masked or unmasked from the function that he is trying to exploit. The attacker cannot look at source code to see our added functionality, since it has been added to the binary via an extended .text section. The attacker could, however, look at the binary program code or attempt to disassemble the program and analyze the assembly code. These are much higher barriers of entry, though. See Chapter IV, Section A for further details on deployment.

## **B. EXECUTABLE STRUCTURE**

Our obfuscation technique applies additional instructions to an already compiled executable. Modifications occurring at the binary level need to be strategic. After a programmer writes an application in their chosen language, it goes through the following compilation process: preprocessing, compilation, assembly, and linking, which enables it to run, or execute, their desired task. The code written in C gets to the executable, or binary, form through this process and is now capable of performing the programmer's

intended action [22]. The compilation process is a series of stripping and mapping techniques that allow the executable to perform virtually identical on different platforms.

The formal specification set out by the compiler during compilation allows the operating system to interpret its underlying machine instructions correctly [22]. In an attempt to universally load and run the executable, a structure of how the program needs to be laid out in address space needs to be defined. This structure of the executable, provided by the linker, is packed into a header that is included with the binary output of the instructions written by the programmer. For our proof of concept, we use a Linux 64-bit operating system that specifically uses the Executable and Linking Format (ELF). An ELF Header is placed at the beginning of the file and details exactly how the code needs to be organized [23]. On other platforms, such as Windows or MacOS, the structure is referred to as Portable Executable (PE) Header or Mach-O Header [24], respectively. Each of these headers is further segmented to handle each section of an executable. For example, they suggest how to layout the functions or where to begin executing instructions. Jeopardizing this structure can be as simple as program strings displaying incorrectly or as detrimental as the program failing to execute. This is why this information is linked during compilation and not visible to the user outside the binary. This does not mean these headers are untouchable. Our obfuscation functionality can only be implemented by strategically manipulating the header, all while preserving execution ability.

Another consideration in binary manipulation is extending it with additional code. When looking to add instructions to a binary, the optimal space to insert is within the .text section among the other instructions. This allows the new code to be interwoven into executable code, permitting sequential instruction flow, if necessary. Inserting code necessitates the shifting of all the following data as well as updating references. To achieve the primary goal of this research, we work with a pre-extended version of the Nano (a well-known Linux text editor) x86-64 binary, generated using the Rose Compiler framework [25]. It contains an additional 0x400 bytes at the end of the original .text section.

One benefit of the ELF executable format is the VOID. This is a section that follows the .text section where null bytes are inserted into the gap that spans the end of the first program segment until the allocated page of executable data is aligned [22]. This VOID space can commonly contain hundreds of bytes or more, though may also be nonexistent. If there were sufficient space, it would make it an ideal candidate for extended code. Utilizing this area cuts down on readdressing the segments that fall below the .text section as those that fall below do not need updating. See Chapter III, Section H for our utilization of this space.

### **C. TOOLS USED**

After a file has been compiled, it contains both instructions and headers in machine-readable format. Options for viewing this information in a more human readable format are a hexadecimal editor, decompiler (produces C code), or disassembler (produces assembly code). All these options will provide the hexcode of the binary with the American Standard Code for Information Interchange (ASCII) representation. In efforts to combat buffer overflow, we need to view the ELF Headers to locate various sections within the Nano binary. In addition, we need to make changes to entries in these headers to allow the binary to once again execute properly. To facilitate these changes, we work with the Bless Hex Editor and the following disassemblers: IDA and Ghidra (includes a decompiler).

The Bless Hex Editor 5.1 is specific to Linux (GNOME) operating systems and is instrumental in binary manipulation. A hexadecimal editor allows you to view and manipulate a binary file on a byte-by-byte basis. The Bless editor was specifically chosen for its ability to add additional bytes to the binary in contrast to limiting changes to that which only yield the same size executable. The ability to extend instructions is mandatory for our research. The result of making changes within Bless is a modified Nano executable that performs similarly to the original with the additional functionality we desire.

Ghidra v9.0.4 is an open source reverse engineering tool that assists in reviewing the binary as decompiled and disassembled, more human readable, C and assembly

language codes. Viewing in this manner allows us to understand the flow of the Nano application as well as choose the best function to demonstrate our protective measures. Due to code obfuscation and misdirection, we found limitations in this tool. Many portions of the binary could not be decompiled nor disassembled into code and were left as individual bytes. Although, coupling this tool with others helps to piece together a complete image.

The Hex-Rays Interactive Disassembler (IDA), freeware version 7.0, provides the binary as the disassembled, human readable 64-bit assembly language code. This application is able to provide the complete assembly of Nano and is able to fill the gaps where the other tools only provide hexadecimal output.

#### **D. INSTRUCTION MODIFICATIONS**

Our research employs two different methods of binary modification. The first method is to alter bytes of the target binary. We refer to this as “*replace\_instructions*.” This involves overwriting existing bytes with desired bytes in order to provide new functionality. For example, the x86-64 instruction (in hexadecimal) “e8 98 f9 ff” (“call *offset*”) could be overwritten as “e8 eb a1 01 00” (“call *new\_offset*”). These two instructions each contain five bytes. No bytes are added or removed from the size of the program and no instructions had their program addresses changed. The second method of binary modification involves adding new bytes where bytes did not exist or, in other words, inserting new bytes/instructions in between already existing bytes/instructions. We refer to this functionality as “*insert\_instructions*.” This involves opening up a space at the desired program address by shifting all of the following bytes in the program memory space down by the inserted number of bytes. For example, we might want to insert the bytes “68 e0 87 40 00” (“push *address*”) right before a call instruction. This adds five new bytes to the program, so all instructions after this added one need to have their addresses shifted down in memory by five bytes. If the added instruction “push *address*” occurs at memory address 0x408e43, then the instruction that used to exist at this address (“call *offset*”) would now have the address 0x408e48 (0x408e43 + 5). It is no small task to make these kinds of changes to the memory map of a compiled program.

There does exist, however, a Python library called *mmap* [26] that can be utilized to do this [27].

The Python library *mmap* provides memory mapped file I/O support in Python 3. With this, we are able to load the entire contents of the target binary file into memory and manipulate it directly, shifting and adding bytes as desired, which, in turn, has the effect of changing the file size. Two functions are written in order to replace and insert bytes into the binary. The first function, *insert\_instructions*, involves inserting a specified number of bytes into the file at a specified offset (Figure 6).

```
def insert_instructions(address, instructions):
    if address >= 0x400000:
        offset = address - 0x400000
    else:
        offset = address
    if type(instructions) == str:
        instructions = bytes.fromhex(instructions)

    insertIntoMmap(offset, instructions)
```

Figure 6. *Insert\_instructions* function

This function does some calculations to determine if the value passed to it is a file address or offset (offset is required), converts them to byte form if necessary (if passed as a string), and then calls *insertIntoMmap*, which manipulates the memory mapped file data to insert the specified bytes. As an example, say we have a file of size 100 decimal (100d) bytes, and we wish to insert 25d new bytes into the file at offset 75d. When *insertIntoMmap* is called, it is passed the offset at which to place the bytes (75d in our example) and the data to be inserted at that location. It will then acquire the current file size and the number of bytes to be added, and it will calculate the new file size. Once it has done this, we seek to the desired offset into the file object and write the data. We then manipulate the memory-mapped file by moving byte 75d down to position 100d, seek to position 75d, and write our 25d bytes. We now have a file that has the original bytes 0d to

74d, new bytes 75d to 99d, and original bytes that used to be at offsets 75d to 99d now at offsets 100d to 124d.

The second function, *replace\_instructions*, involves deleting a specified number of bytes from the file and writing new bytes in their place (Figure 7).

```
def replace_instructions(address, instructions):  
    if address >= 0x400000:  
        offset = address - 0x400000  
    else:  
        offset = address  
    if type(instructions) == str:  
        instructions = bytes.fromhex(instructions)  
  
    deleteFromMmap(offset, offset+len(instructions))  
    insertIntoMmap(offset, instructions)
```

Figure 7. Replace\_instructions function

This function first calls another function, *deleteFromMmap*, which manipulates the memory-mapped file. As another example, say we have a file size of 100d bytes, and we wish to replace bytes 50d to 74d with some other data. The first step will be to delete those bytes from the memory map of the file. *deleteFromMmap* calculates the number of bytes to remove based on the starting and ending file offsets that are passed to it. In our example, the number of bytes to be removed will be  $75d - 50d = 25d$ . It calculates the new file size after the bytes are removed. In our example, this will be  $100d - 25d = 75d$ . It then moves bytes 75d to 100d up to the location of byte 50d, making byte 75d now byte 50d and effectively deleting bytes 50d to 74d from the original file. Finally, it truncates the file object by calling the *truncate()* function on the file descriptor for the file we are working with. Once the desired bytes are deleted, the *replace\_instructions* function then calls *insertIntoMmap*. As previously described, this function manipulates the memory mapped file to insert the desired bytes. To continue with our example, we want to insert 25d new bytes at offset 50d, which will make the offsets of these added bytes 50d to 74d, as in the original file. *insertIntoMmap* is passed the offset to place the bytes (50d in our



example) and the data to be inserted at that offset. It then uses the length of the data and the current size of the file to calculate the new file size. Once it has done this, we seek to the desired offset of the file object and write the data. We then manipulate the memory mapped file by moving byte 50d down to position 74d, seek to position 50d, and write our 25d bytes. We now have a file that has the original bytes 0d to 49d, new bytes 50d to 74d, and original bytes 75d to 99d.

If a file is modified in the way that was just described, so as to end up with a net gain of bytes (in our concept this is nine bytes), we decide to preserve the original file size. This helps maintain some of the integrity of the original program by reducing the number of header values that need to be updated (i.e., file size values). In order to achieve this, we have a final call to a function, *delete\_extra\_bytes* (Figure 8).

```
def delete_extra_bytes(address, num_bytes):
    if address >= 0x400000:
        offset = address - 0x400000
    else:
        offset=address

    deleteFromMmap(offset, offset+num_bytes)
```

Figure 8. Delete\_extra\_bytes function

This function performs the same data validation checks (converting an address to an offset as necessary) and then calls *deleteFromMmap* which performs as previously described. In this case, the offset that is passed as the location to delete bytes from is part of the extended .text section that is not utilized.

## E. MASKING AND UNMASKING FUNCTIONS

Now that we have described the functions that were written to perform the desired changes, we can describe the changes made to the binary so as to achieve our goal of protecting a function's return address on the stack. The technique consists of two steps: 1) implementing masking and unmasking functions and 2) redirecting program flow. The

first step inserts two functions into the extended .text section that implement mask and unmask operations on the target return address. Inserting them into this extended .text section allows them to be referenced as legitimate code within the binary executable, all while placing them below the original binary instructions within the .text segment. The masking function is referred to as *me1* (Figure 9).

```
me1:
    pop r8          ;Save real return address.
    pop r9          ;Save real call address.
    xor r8, 0x2a    ;Mask return address.
    push r8         ;Replace return address.
    jump r9         ;Jump to real call address.
```

Figure 9. Masking function, *me1*

This function performs a simple *xor* operation on the passed return address with a predefined value (i.e., *xor return\_address 42*). When *me1* is called, the address of the previous function (*caller*) and the original address to be called (*callee*) are on the stack. The first two instructions of *me1* are to pop that return address into a register (*r8*) and then pop the original address to be called into another register (*r9*) where they can be further utilized. The *xor* operation is performed on register *r8* with the constant 42 (though this value seems fixed, it can be chosen by the end-user, as detailed in Chapter IV, Section A) and then that value is pushed back onto the stack where it belongs as the return address for the *caller*. Finally, the instruction “*jump r9*” is called, which allows program flow to continue as originally expected to the original *callee* (before “*call me1*” was injected into the program).

The unmasking function is referred to as *me2* (Figure 10).

```

me2:
    pop r8          ;Retrieve masked return address.
    xor r8, 0x2a    ;Unmask return address.
    push r8         ;Replace real return address.
    ret            ;Return to real return address.

```

Figure 10. Unmasking function, me2

This function reverses the masking operation performed by *me1*. Just before the return address is used in the program flow (*ret* instruction), the program performs a jump to *me2* where that return address is popped off of the stack into register *r8*. The same *xor* operation is performed to get back the correct return address. The address is then pushed back onto the stack and a *ret* instruction is called to return program flow to its originally intended address, all while preserving the stack.

## F. REDIRECTING PROGRAM FLOW

The goal of this project is to perform a proof of concept of this obfuscation functionality on a single operation of the program Nano. In our preparation work, we identify a function that executes when a user issues the “open file” command by pressing *control+r* from within Nano. In order to integrate these two new functions (*me1* and *me2*) into the program flow, we locate the instruction that calls this “open” function (the *caller*) and the function that performs the “open” operation (the *callee*). Subsequently, we make modifications to the binary code to ensure program flow is redirected to our mask and unmask functions so as to achieve the desired obfuscation. The specific process consists of two steps, illustrated in Figures 11 and 12, respectively. The first step, illustrated in Figure 11, is to modify the *caller* in order to redirect flow to *me1* before the *callee* is executed. To this end, the *callee*’s address is pushed onto the stack (to enable the original control flow to the *callee* once masking is complete), then we replace the called address with the address of *me1*. This involves adding five new bytes for the “push” operation and modifying the five bytes of the “call” operation.

```

;Original instruction:
call sub_4087e0          ;Original call instruction.

;Updated instructions:
push 4087e0             ;Save reall call address.
call 42303c             ;Call me1.

```

Figure 11. Caller redirection

The next step, illustrated in Figure 12, is to integrate instructions to redirect flow from the *callee* back to *me2* before it returns to the *caller*. As soon as the *callee* finishes executing, it would normally execute a *ret* instruction to return to its *caller*. In order to redirect flow to our *me2*, we have to replace this one byte “return” instruction with a five byte “jump” instruction that jumps to *me2*. The last command that *me2* performs is a *ret*, which returns flow to the *caller* in the same way that it would have originally returned (preserving the stack).

```

;Original instruction:
ret                    ;Original return to caller.

;Updated instructions:
jump 423049            ;Change return to jump, insert address to me2.

```

Figure 12. Callee redirection

A side effect of the above-mentioned modifications is that any instructions addressed via offsets become incorrect. Consequently, references to such addresses in the program that are shifted need to be updated to account for the added bytes. See Figure 13 for updated program flow.

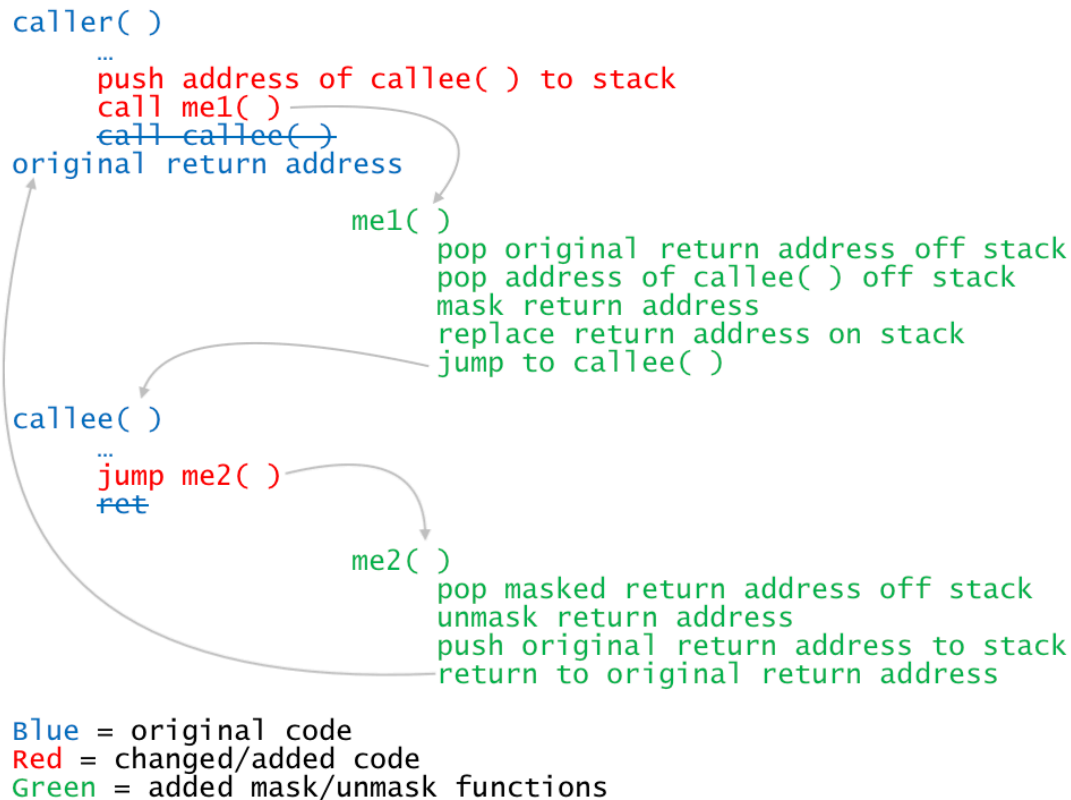


Figure 13. Redirected program flow

## G. ADDRESS MODIFICATIONS

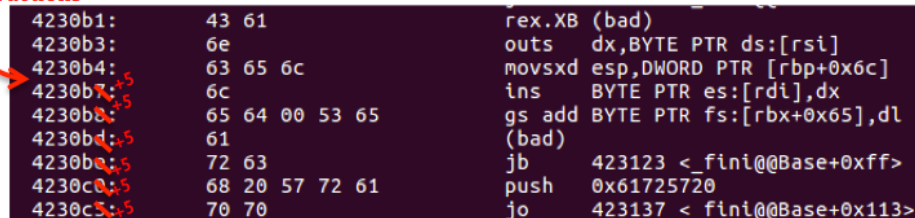
As described so far, we instrumented the binary code in such a way that the size of the .text section has increased with the desired effect of protecting the return address. Because of the ELF header structure, when adding additional lines of code to the existing program we need to account for the addresses that fall below the newly inserted instructions. Each instruction has a corresponding address. Jumping or calling specific instructions is done by referencing this address. When lines of code are inserted, you have to correct for the new address or any reference that utilizes an offset to this location. Consequently, we have written the “UHaul” program, that takes into account where additional code will be inserted then updates the original addresses accordingly.

Redirecting addresses within the existing executable depends on a number of details. In our particular case, obfuscation requires changes within the *callee* and *caller* functions. UHaul accomplishes the consequential redirection by first retrieving an object

dump of the original program's .text section (where executable instructions lie). The objdump command results in a listing of the original Nano application's instructions with their corresponding addresses. Restricting changes to only the .text section was feasible because we worked with a binary that had an extended .text section. A result of this extension was patching of references to the .rodata, which had been shifted as well. Had we not worked on an extended binary, we would have had to change the addressing of the sections that follow. This severely decreased the number of changes required in other sections within the executable.

After the UHaul program receives output from the objdump, each address (line by line in file) is then checked for its location relative to the changes necessary to the extended Nano. UHaul goes through each line checking each address for where it falls relative to the changes (see Figure 14). Based on the checked address's location, an addition is made to the address value resulting in its new location in the binary. For example, if an instruction falls between change one and change two, five will be added to the original address to account for the five bytes of code in change one.

Insert 5 bytes of instructions



4230b1:	43 61	rex.XB (bad)
4230b3:	6e	outs dx,BYTE PTR ds:[rsi]
4230b4:	63 65 6c	movsxd esp,DWORD PTR [rbp+0x6c]
4230b7:	6c	ins BYTE PTR es:[rdi],dx
4230b8:	65 64 00 53 65	gs add BYTE PTR fs:[rbx+0x65],dl
4230bd:	61	(bad)
4230be:	72 63	jb 423123 <_fint@@Base+0xff>
4230c0:	68 20 57 72 61	push 0x61725720
4230c5:	70 70	jg 423137 <_fint@@Base+0x113>

Figure 14. Addressing changes

Once the address shift changes are made, jump and call instructions need to be considered. When an instruction needs to change program flow from moving sequentially to the next line of code to moving to another location within the program, a jump or a call instruction is used. Movement in a program can be achieved either by supplying the address where the instruction lies (i.e., direct addressing) or through an offset value (i.e., indirect addressing). In indirect addressing, the offset is a displacement relative to the address of the next instruction. When we encounter changes to those instructions that

utilize indirect addressing, identifying the difference between a jump or call is done by looking at the opcode. See Figure 15 on how to redirect based on a given offset.

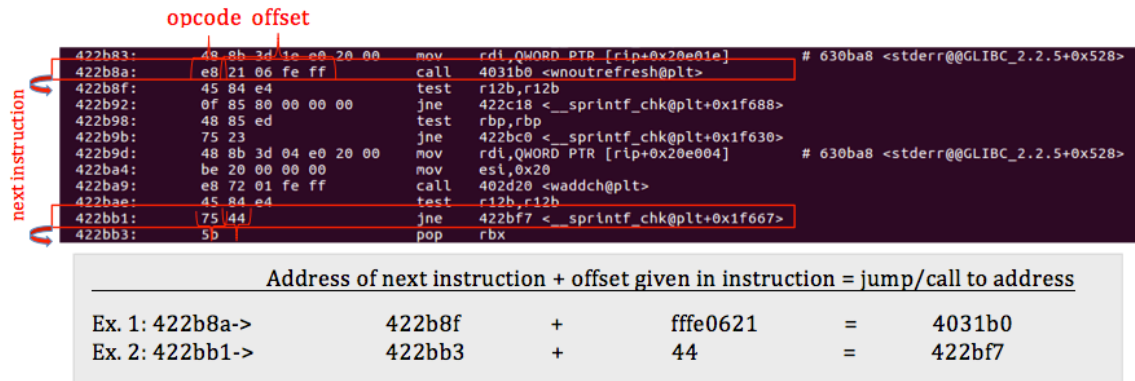


Figure 15. Indirect addressing

When our program encounters indirect addressing, we need to correct the offset. The instructions are parsed from the objdump by selecting opcodes specific to indirect addressing. Once found, the offset is checked if its value is a signed integer. Values that are negative, or jump to instructions previous to the change(s), are corrected by two's compliment. The offset value is then added to the address of the next instruction to check if it requires a change (i.e., was affected by the inserted instructions). If a change is necessary, the addition/subtraction is done on the offset value and added to the change array.

For those with instructions using direct addressing, addition directly to the address is applied. Those opcodes that are specific to direct addressing are searched for in the objdump. When these opcodes are encountered, code is written to extract the instruction from the objdump file then compare the address to those of the changes. Depending on where that address is located, arithmetic operations may be performed to account for the shifting. A list of these changes is kept in an array that corresponds to the addresses where the instructions fall and are passed to the *mmap* function which maps them into the extended binary.

## H. BINARY EXTENSION

A secondary goal of this research is to implement the extending of the .text section and the updating of the necessary headers, independent of the extension that is made with the Rose compiler. The first step in performing a .text section extension is to conduct a dependency analysis, locating all the headers that would be affected by adding bytes into the middle of the binary. (For purposes of this discussion, assume  $N \in I$  and  $N \geq 1$ .) See Figure 16 for the ELF binary structure.

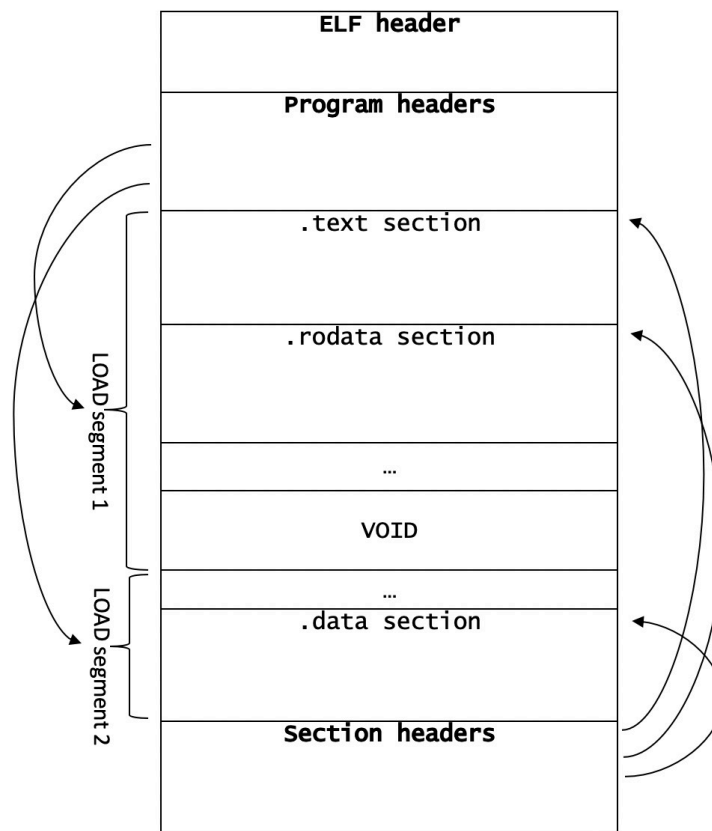


Figure 16. ELF binary structure

The ELF header is located first as it gives additional information about how to locate the rest of the headers. The ELF header always starts at offset zero in the binary and is often a constant size of 64 bytes, as is the case for our work with Nano. The ELF



header contains information such as the program header offset, the section header offset, the program headers' size, the number of program headers, the section headers' size, the number of section headers, and the section header string table index.

The section headers are modified first. The first section header that needs changes is that of the `.text` section, in which we update the size value by adding  $N$  bytes. Every section that occurs after the `.text` section and up to the start of the next program segment (e.g., `.fini` and `.rodata`), needs to have its starting address and offset shifted by the  $N$  bytes added to the `.text` section. The sections that exist in a new segment (e.g., starting after address `0x620000` in Nano) do not need to be modified, because there is empty, unused space between segments that we refer to as the VOID. This consists of unused null bytes at the end of the first program segment (that is aligned with address `0x400000`). In order to preserve the addressing of sections in this second segment (e.g., `.dynamic`, `.got`, and `.bss`), we remove bytes from this VOID area.

After dealing with the section headers, we need to modify/update the program headers. Updated program headers include the first LOAD section, which contains our updated `.text` section, as well as any program headers that exist after the `.text` section and before the VOID. The load section contains a value for the size of its entry and the size of the memory space that it utilizes. As for the program headers that exist in the binary after the `.text` section and before the VOID (e.g., `PT_GNU_EH_FRAME` in Nano), their starting file offsets as well as their virtual and physical starting addresses are updated. Each of these modifications to values in the binary is performed using our *mmap* functionality that was described in Chapter III, Section A. The extended binary is then run through the UHaul program in order to get a patched executable.

## **I. CODE HIDING**

A third goal of this research, based on the discovery of the previously described VOID section of Nano (of size 2,812 bytes), became to hide our added functionality in this area. Placing code in this area avoids having to extend the `.text` section of the program. Instead of adding the obfuscation and deobfuscation functions to the `.text` section, they are added in this VOID space. Shifting still occurs as described in Chapter

III, Section F, but the added bytes can be recovered from the VOID section. The advantage of using this VOID space is that it does not belong to any program sections, and is therefore not disassembled. It is possible to have executable code in this area because it is part of the same memory segment as the .text section and holds the same permissions. We were able to test this by writing instructions that performed a print of “Hello World” which succeeded in executing from this memory space.

THIS PAGE INTENTIONALLY LEFT BLANK

## IV. RESULTS

We tested `nano.new.NEW.GOLDEN.patched` (the modified, instrumented version of Nano) from the terminal using the GNU project debugger (gdb). Using gdb, we set a breakpoint at the instruction where the program returns using our obfuscated return address. The breakpoint is set at the jump to `me2`. See Figure 17.

```
Thread 1 "nano.new.NEW.GO" hit Breakpoint 3, 0x0000000004088fb in ?? ()
1: x/4i $rip
=> 0x4088fb:    jmp     0x423049 <- redirection to me2()
    0x408900:    nop     DWORD PTR [rax+0x0]
    0x408904:    lea     rdx,[rsp+0x90]
    0x40890c:    mov     rsi,r12
2: /x $r8 = 0x681c00
3: /x $r9 = 0x0
(gdb) x/xg $rsp
0x7fffaeb47b58: 0x00000000000408e7b <- original return address on the stack
(gdb) set {int}0x7fffaeb47b58 =0x00408e6b
(gdb) x/xg $rsp
0x7fffaeb47b58: 0x00000000000408e6b <- attacker-controlled return address on the stack
(gdb) si
0x00000000000423049 in ?? ()
1: x/4i $rip me2():
=> 0x423049:    pop     r8
    0x42304b:    xor     r8,0x2a <- attempt at unmasking
    0x42304f:    push    r8
    0x423051:    ret     <- return to attacker's address fails here
2: /x $r8 = 0x681c00
3: /x $r9 = 0x0
(gdb)
```

Figure 17. Breakpoint set

Mimicking an attacker changing the return address, we changed it manually in gdb. This has the effect of simulating a buffer overflow that attempts to redirect program flow. When `me2` deobfuscates the return address with the `xor` operation, the result is, as planned, an invalid address; hence, the program crashes, inducing a failed attack.

THIS PAGE INTENTIONALLY LEFT BLANK

## V. CONCLUSION

### A. SUMMARY

Our Nano editor program handles our specific test case. We showed that program code could be instrumented to prevent attackers from taking advantage of buffer overflows by obfuscating function return addresses. This instrumentation was done directly on the binary, as opposed to compiler- or OS-based protections. Any attempt to corrupt a return address causes the program to crash. By doing this, we have hardened defenses against buffer overflow attacks.

In order to implement these added security features, a binary was extended and modified to preserve original program flow and functionality. This necessitated changing the headers and updating instruction addresses and offsets throughout. The success of the implementation of our tool on a Linux ELF binary gives us confidence that this tool can be ported to other executable formats and operating systems. In addition, our tool provides a foundation for more tools to be developed that are tailored to a user's specific needs.

### B. DEPLOYMENT

Note that the work in this thesis was a limited proof of concept. To fully utilize this technique, one would either a) provide this obfuscation to every return address / function in the program or b) provide the obfuscation to specific functions where attacks may be possible. Of course, if vulnerable functions are known to exist in advance, there are likely other more appropriate coding practice fixes that can be implemented (such as using a C *fgets* function to get bounded input from a user rather than *gets*, which is boundless).

In an ideal attack scenario, Alice, the attacker, is oblivious to the use of a masking function in the binary and would not discover it in her attempt at an attack. This is likely the case when Alice does not have access to the binary. In this case, Alice will attempt to overflow a buffer to place her own malicious return address on the stack. Our program will have masked that return address (*mel xor* operation) and continued program

execution to the *callee*. As soon as that return address is needed to resume program flow at the location following the *caller*, Alice’s address will be deobfuscated. This will provide an illegitimate program address, causing the program to crash.

Next, consider a situation in which Alice does have access to the binary or is somehow able to determine that defensive return address obfuscation is in place. Alice can still theoretically discover the mask. If Alice knows that the masking function uses an *xor* operation, she can change the return address so that after it is obfuscated, it becomes the address that she intends to jump to. In defense of the information leak for the compromised binary, the defender, Bob, would use custom mask values and custom mask functions, rather than constant ones (e.g., *42d* and *xor*). This requires regular changes to the Nano editor script. Instead of performing an operation, such as *xor 42d* as we have implemented in this work, the mask can be changed to any other number (e.g., *1337d*, *4444d*) and the masking operation(s) can be changed (e.g., *add 42d*, *mult 7d*, *xor 128d*). Deobfuscation is the inverse of these operations. To ensure its reliability, an automated program could change the binaries on a schedule that implements new masks and new obfuscation and deobfuscation functions.

### C. ADAPTING BEYOND ELF

Since 58.4% [28] of U.S. government systems run the Windows operating systems, we realize a real-world application of our tool would be best adapted to these platforms. We demonstrated the overall idea of protecting return addresses and extending executables in a Linux OS, but we know the approach is applicable to other OSes. Windows portable executables (PEs) are not that different in their addressing concepts. Therefore, Windows executables differ in name, but at a granular level, their headers still dictate structure. This commonality of dedicated fields that tell the OS how to layout the file, where to begin execution, and what resources to use, are still contained in the binary of a Windows PE. We encoded our tool with a means to traverse the binary headers to find this data. It is trivial to make our tool work in this same manner on a Windows system. Our tool relies solely on the offsets within the ELF Header that contain structural metadata to run the executable. The Windows PE Header provides exactly the same and

this information is publicly available. With this similarity, research could be done into building a Windows-specific tool.

One other modification to usher our tool into the real world would be amending it to work on Apple computers. Although not as pervasive, 9.2% [28] of the U.S. government sector utilizes Apple products and that number is continuously rising. Therefore, this is another arena that needs protection and, again due to similar header structure, research could be done into developing a tool to fit the Mach-O format.

#### **D. FUTURE WORK**

Further research can be done into adapting the tool we have built here into one that is automated and generalized, providing optimal protection with nearly no user interaction. This tool could scour the binary for each individual function. Once functions are found, the tool adds our protective technique. This would provide the obfuscation security to each return address of each function in the executable. This will add obfuscation functionality to functions that may not be susceptible to buffer overflow, but the added overhead is minimal to the overall size increase of the executable. The size of our added functionality is at most 14 bytes. If a program contained 100 functions, this would be at most an additional 1400 bytes. Our Nano executable was 204,000 bytes. This type of extension accounts for 0.6% of the application.

Another area of future work is to perform additional testing and experimentation. Benchmarks may be performed to see what kind of performance overhead is incurred by the added functionality described above. Depending on whether these protections are implemented at every function or only specific functions, a significant performance impact may occur.

Additionally, work can be done on developing and testing attacks against our protection mechanism. The algorithm developed in this proof-of-concept is relatively straightforward and may be susceptible to brute-forcing. Future work can be done into utilizing more complex obfuscation algorithms, as discussed in Chapter IV, Section A.



THIS PAGE INTENTIONALLY LEFT BLANK

## APPENDIX A. MMAP EDITOR

```
1 import mmap
2 import sys
3
4 ## Code adapted from user 0x90 on Stack Overflow, found at:
5 ## https://stackoverflow.com/questions/5917047/delete-insert-data-in-mmaped-file
6
7 def openfile():
8     global VDATA
9     global f
10    f = open(sys.argv[-1], "r+b")
11    VDATA = mmap.mmap(f.fileno(), 0)
12    return f
13
14 def deleteFromMmap(start, end):
15     global VDATA
16     global f
17     length = end - start
18     size = len(VDATA)
19     newsize = size - length
20
21     VDATA.move(start, end, size - end)
22     VDATA.flush()
23     VDATA.close()
24     f.truncate(newsize)
25     VDATA = mmap.mmap(f.fileno(), 0)
26
27 def insertIntoMmap(offset, data):
28     global VDATA
29     global f
30     length = len(data)
31     size = len(VDATA)
32     newsize = size + length
33
34     VDATA.flush()
35     VDATA.close()
36     f.seek(size)
37     f.write(data)
38     f.flush()
39
40     VDATA = mmap.mmap(f.fileno(), 0)
41     VDATA.move(offset + length, offset, size - offset)
42     VDATA.seek(offset)
43     VDATA.write(data)
44     VDATA.flush()
45
46 ## insert instructions by shifting down all bytes after them
47 def insert_instructions(address, instructions):
48     if address >= 0x400000:
49         offset = address - 0x400000 # calculate offset into file from virtual address
50     else:
51         offset = address
52     if type(instructions) == str:
53         instructions = bytes.fromhex(instructions) # convert string instructions into byte form
54
55     insertIntoMmap(offset, instructions) # perform insertion
56
57 ## replace instructions by deleting old and inserting new
58 def replace_instructions(address, instructions):
59     if address >= 0x400000:
60         offset = address - 0x400000 # calculate offset into file from virtual address
61     else:
62         offset = address
63     if type(instructions) == str:
64         instructions = bytes.fromhex(instructions) # convert string instructions into byte form
65
66     deleteFromMmap(offset, offset + len(instructions)) # perform deletion
67     insertIntoMmap(offset, instructions) # perform insertion
68
```

```

69 ## delete extra bytes in extended text section to account for new bytes added
70 def delete_extra_bytes(address, num_bytes):
71     if address >= 0x400000:
72         offset = address - 0x400000          # calculate offset into file from virtual address
73     else:
74         offset=address
75
76     deleteFromMmap(offset, offset+num_bytes)
77
78 def clean_up():
79     global VDATA
80     global f
81     VDATA.flush()
82     VDATA.close()
83     f.flush()
84     f.close()

```

## APPENDIX B. INSTRUCTION INSERTION

```
1 from mmap_editor import *
2 from shutil import copyfile
3 import os
4
5
6 ## call insertion function by specifying virtual program address (hex int) and instructions (string)
7 ## instructions will be inserted at address and push all following bytes down
8
9
10 ## call replace function by specifying program address (hex int) and instructions (string)
11 ## function will delete the bytes at address and replace them with instructions
12
13
14 ## ----> START HERE FOR NANO <---- ##
15
16 ## to be used on nano.new.NEW.GOLDEN to add functionality for me1 and me2.
17
18 copyfile(sys.argv[-2], sys.argv[-1]) # create a copy of the input file to work with
19
20 openfile()
21
22 '''
23 Preparation for call to me2.
24 Change return to jump.          jump
25 Insert jump address to me2.     423049
26 '''
27 replace_instructions(0x4088fb, "e9")
28 insert_instructions(0x4088fc, "49a70100")
29
30 '''
31 Preparation for call to me1.
32 Save real call address.          push 4087e0
33 Call me1.                        call 42303c
34 '''
35 insert_instructions(0x408e47, "68e0874000")
36 replace_instructions(0x408e4c, "e8eba10100") # added 5 to address to account for inserted instructions before
37
38 '''
39 Write me1.
40     pop r8          Save real return address.    (408e48)
41     pop r9          Save real call address.      (4087e0)
42     xor r8, 0x2a    Mask return address.
43     push r8         Replace return address.
44     jump r9         Jump to real call address.
45 '''
46 replace_instructions(0x42303c, "415841594983f02a415041ffe1") # shifted left by 1 byte
47
48 '''
49 Write me2.
50     pop r8          Retrieve masked return address.
51     xor r8, 0x2a    Unmask return address.
52     push r8         Replace real return address.
53     ret             Return to real return address.
54 '''
55 replace_instructions(0x423049, "41584983f02a4150c3")
56
57 '''
58 Delete extra bytes near end of text section to account for those added with insert.
59 '''
60 delete_extra_bytes(0x423052, 9)
61
62
63 #####
64 # Run objColumnsPLT.py (should use the same command line arguments)
65 command2 = 'python3 SLEAKobjColnoPrint.py ' + sys.argv[-2] + ' ' + sys.argv[-1]
66 command3 = 'chmod +x ' + sys.argv[-1]
67
68 #perform the objdump
69 os.system(command2)
70 os.system(command3)
```

THIS PAGE INTENTIONALLY LEFT BLANK

## APPENDIX C. UHAUL

```
1 #-----
2 #-----
3 ##imports
4
5 import sys
6 import os
7 import time
8
9 ##will take the array of changes and insert/overwrite into extended file
10 from mmap_editor import *
11
12 #-----
13 #-----
14 ##initializations
15
16 ##dictionary of operation codes (Opcodes) as keys with amount of bytes as values (aids with offset math)
17 opCodes = {"e8": "1",
18            "70": "1",
19            "71": "1",
20            "72": "1",
21            "73": "1",
22            "74": "1",
23            "75": "1",
24            "76": "1",
25            "77": "1",
26            "78": "1",
27            "79": "1",
28            "7a": "1",
29            "7b": "1",
30            "7c": "1",
31            "7d": "1",
32            "7e": "1",
33            "7f": "1",
34            "e3": "1",
35            "e5": "1",
36            "eb": "1",
37            "0f80": "2",
38            "0f81": "2",
39            "0f82": "2",
40            "0f83": "2",
41            "0f84": "2",
42            "0f85": "2",
43            "0f86": "2",
44            "0f87": "2",
45            "0f88": "2",
46            "0f89": "2",
47            "0f8a": "2",
48            "0f8b": "2",
49            "0f8c": "2",
50            "0f8d": "2",
51            "0f8e": "2",
52            "0f8f": "2"}
53
54 ##location in nano where additional code will be inserted in the .text section
55 ##at change 1 addresses below will have +5 added
56 ##at change 2 addresses below will have an additional +4 added
57 change1 = 0x4088fc
58 change2 = 0x408e47
59
60 ##initialization of arrays
61 fileArray = []
62 hexArray = []
63 addrArray = []
64 instructionArray = []
65 toChangeArray = []
66 offsetOGArray = []
67 weirdMaths = []
68 addrOGArray = []
69 forPrint = []
70
71 ##initialization of dictionaries
72 commentArray = {}
73 ryansDict = {}
74
75 ##initialization of counters
76 counter = 0
77 changeCounter = 0
78 printSet = 0
79 arrayCtr = 0
80 readCtr = 0
81 weirdCtr = 0
82
83 ##initialization of strings
84 textLine = "Disassembly of section .text:"
85 stopLine = "Disassembly"
86
87 ##initialization of sets
88 weirdOps = set()
89 weirdInst = set()
90
91 ##initialization of flags
92 szOpodeFlag = 0x00
93
94 ##argument passed in from command
95 #file to be objdump'd given as arg ORIGINAL FILE because it uses original address to calc
96 fileObjdOG = sys.argv[1]
97
98 #-----
99 #-----
100 ##functions
101
102 def weirdOffsets(instcAddr, weirdCtr, commentAddr):
103     #this works with instructions where address is "dir/hard" coded rather than offset/rel
104     #comment addr what is found in comment what is written in instro
105     #instcAddr = offset in prog of instruction
106     #all those that need to be changed are kept in dict: commentArray
107     #go through each and create array of changes could add to other change array
108     #key addr of inst, value is addr in plt
109     #need to get instruct to do work
```

```

110     #get corresponding hex instruction
111     #need to know index that instruction addr is cuz that matches index of cores hex instruct
112     #this is full instruction
113     makeHex = hexArray[addrArray.index(instcAddr)]
114
115     #strips it down to just the last bytes of offset
116     #takes offset listed in assembly instruction and removes all else
117     #this is what is used to strip out opcode
118     #this what will be reduced, used for math
119     if int(commentAddr,16) >= 0x600000:
120         getInstrc = (((instructionArray[addrArray.index(instcAddr)]).split("+"))[1].split(" ")) [0]
121
122     else:
123         getInstrc = "0x" + commentAddr
124
125     #flip them so they match hex
126     getInstrcFlip = endianness(int(getInstrc,0))
127
128     #match spacing, preserve og
129     getInstrcSpod = " ".join(getInstrcFlip[i:i+2] for i in range(0, len(getInstrcFlip), 2))
130
131     #this and sz are used to make changes to correct location in instruc
132     #get out opcode, no spaces
133     getJustOpcode = (makeHex.split(getInstrcSpod)[0]).replace(" ", "")
134
135     #if doesnt find in string
136     if makeHex.replace(" ", "") == getJustOpcode:
137         #did not match on assembly instc, need next line; is str
138         restInstrc = hexArray[addrArray.index(instcAddr)+1]
139         lenXInstrc = len(restInstrc)
140         wholeInstrc = makeHex + " " + restInstrc
141         getJustOpcode = (wholeInstrc.split(getInstrcSpod)[0]).replace(" ", "")
142
143
144     #need to rid of what is being moved into that offset, when follows, pre taken care of by dir of find
145     opCodeSz = int(len(getJustOpcode)/2)
146
147     #account for address extension
148     if int(instcAddr,16) >= change2:
149         instcAddr = int(instcAddr,16) + 0x09
150
151     elif int(instcAddr,16) >= changel:
152         instcAddr = int(instcAddr,16) + 0x04
153
154     else:
155         instcAddr = int(instcAddr,16)
156
157     #int here
158     if int(commentAddr,16) >= 0x600000:
159         #2nd value string, 1st int
160         #if above (less than changel) will have offseted +9 away so -no
161         if instcAddr >= change2:
162             updatedOffset = int(getInstrc,0) - 0x09
163
164             #get into 2d array that will eventually be put in whole changes array
165             weirdMaths.append([])
166             weirdMaths[weirdCtr].append(instcAddr+opCodeSz)
167             weirdMaths[weirdCtr].append(endianness(updatedOffset))
168             weirdCtr += 1
169
170         #between
171         elif instcAddr >= changel:
172             updatedOffset = int(getInstrc,0) - 0x04
173
174             #get into 2d array that will eventually be put in whole changes array for ryan
175             weirdMaths.append([])
176             weirdMaths[weirdCtr].append(instcAddr+opCodeSz)
177             weirdMaths[weirdCtr].append(endianness(updatedOffset))
178             weirdCtr += 1
179
180     elif int(commentAddr,16) >= change2 and int(commentAddr,16) < 0x423040 and len(commentAddr) == 6:
181
182         updatedOffset = int(getInstrc,0) + 0x09
183
184         #get into 2d array that will eventually be put in whole changes array for ryan
185         weirdMaths.append([])
186         weirdMaths[weirdCtr].append(instcAddr+opCodeSz)
187         weirdMaths[weirdCtr].append(endianness("00"+format(updatedOffset,"x")))
188         weirdCtr += 1
189
190     elif int(commentAddr,16) >= changel and int(commentAddr,16) < change2 and len(commentAddr) == 6:
191
192         updatedOffset = int(getInstrc,0) + 0x04
193
194         #get into 2d array that will eventually be put in whole changes array for ryan
195         weirdMaths.append([])
196         #add offset but need to append zeros
197         weirdMaths[weirdCtr].append(instcAddr+opCodeSz)
198         weirdMaths[weirdCtr].append(endianness("00"+format(updatedOffset,"x")))
199         weirdCtr += 1
200
201     elif int(commentAddr,16) >= 0x423040 and int(commentAddr,16) <= 0x42341f and len(commentAddr) == 6:
202
203         commentAddr = int(commentAddr,16) + 0x400
204         commentAddr = format(commentAddr, 'x')
205         updatedOffset = int(getInstrc,0) + 0x400
206
207         #get into 2d array that will eventually be put in whole changes array for ryan
208         weirdMaths.append([])
209         #add offset but need to append zeros
210         weirdMaths[weirdCtr].append(instcAddr+opCodeSz)
211         weirdMaths[weirdCtr].append(endianness("00"+format(updatedOffset,"x")))
212         weirdCtr += 1
213
214     return weirdCtr
215
216
217
218
219 #-----

```

```

220
221 def checkChange(chgdAddr, way):
222
223     #need to see if new address falls in a nop zone, if so add more
224     #this shouldnt work
225     #if new change falls on inserted bytes needs more change
226     if chgdAddr >= changel and chgdAddr <= (changel + 0x03):
227         if way == "ADD":
228             chgdAddr = chgdAddr + 0x04
229             checkChange(chgdAddr, "ADD")
230         if way == "SUB":
231             chgdAddr = chgdAddr - 0x04
232             checkChange(chgdAddr, "SUB")
233
234
235     if chgdAddr >= change2 and chgdAddr <= (change2 + 0x04):
236         if way == "ADD":
237             chgdAddr = chgdAddr + 0x05
238             checkChange(chgdAddr, "ADD")
239         if way == "SUB":
240             chgdAddr = chgdAddr - 0x05
241             checkChange(chgdAddr, "SUB")
242
243     return chgdAddr
244
245
246 -----
247
248 def twosComp(negNum, instroc, szOpCode):
249     getFs = "f"
250     eightCodes = {"eb", "e3", "70", "71", "72", "73", "74", "75", "76", "77", "78", "79", "7a", "7b", "7c", "7d", "7f"}
251     three2Codes = {
252         "0f80",
253         "0f81",
254         "0f82",
255         "0f83",
256         "0f84",
257         "0f85",
258         "0f86",
259         "0f87",
260         "0f88",
261         "0f89",
262         "0f8a",
263         "0f8b",
264         "0f8c",
265         "0f8d",
266         "0f8e",
267         "0f8f",
268         "e8",
269         "es"}
270
271     #process:
272     #get offset listed in instruction, comes in striped of spcs, opcode, and as string
273     #change to binary rep
274     #is bin aligned 3/16/32/64
275     #if aligned, is it negative
276     #if neg, gets twos compl, use that num as offset
277     #if not aligned or is positive, pass through as is, offset is already in form
278
279     #get bin
280     binNum = int(negNum, 16)
281
282     #see what first index is only those aligned
283     if len((bin(binNum))[2:]) == 8 or len((bin(binNum))[2:]) == 16 or len((bin(binNum))[2:]) == 32 or len((bin(binNum))[2:]) == 64:
284
285         #in bin form str
286         binNum = bin(binNum)
287         #if starts with 1 need twos
288         #is binary rep of offset here as str
289         if instroc[5].replace(" ", "") in three2Codes or instroc[2] in three2Codes:
290             #need to be 32/64 bits, if not then auto not neg
291             if len(binNum) < 34:
292                 binNum = int(binNum, 2)
293                 return binNum
294
295         #accounts for "0b", is it negative
296         if binNum[2] == "1":
297             #do work#flip bits#ff=8 1's#ffff = 16 1's; make f string to xor with
298             getFs = "0x" + getFs.rjust((int(len(binNum)/4)), 'f')
299
300             #is str
301             getFs = int(getFs, 0)
302             binNum = int(binNum, 2)
303
304             #xor in hex
305             binNum = bin(binNum ^ getFs)
306
307             #add 1
308             binNum = int(binNum, 2) + 0x1
309             binNum = binNum * (-1)
310
311         else:
312             binNum = int(negNum, 16)
313
314     #else:
315     binNum = int(binNum, 16)
316
317     #if no work offset should be as is
318     #can pass on thru
319     #this is an int need to make hex for math
320     return binNum
321
322 -----
323
324
325 def endianness(change):
326     #switch endianness, comes in as int
327
328     if type(change) == int:
329         change = format(change, "x")

```



```

330
331     if len(change) == 1 or len(change) == 3 or len(change) == 5 or len(change) == 7:
332         change = "0"+change
333
334     #for 2 byte offset
335     if len(change) == 4:
336         change = change[2:4] + change[0:2]
337
338     #for 3 byte
339     elif len(change) == 6:
340         change = change[4:6] + change[2:4] + change[0:2]
341
342     #for 4 byte
343     elif len(change) == 8:
344         change = change[6:8] + change[4:6] + change[2:4] +change[0:2]
345
346     #for 5 byte
347     elif len(change) == 10:
348         change = change[8:10] + change[6:8] + change[4:6] + change[2:4] +change[0:2]
349
350     #for 6 byte
351     elif len(change) == 12:
352         change = change[10:12] + change[8:10] + change[6:8] + change[4:6] + change[2:4] +change[0:2]
353
354     #for 6 byte
355     elif len(change) == 14:
356         change = change[12:14] + change[10:12] + change[8:10] + change[6:8] + change[4:6] + change[2:4] +change[0:2]
357
358     return(change)
359
360
361 #-----
362 #-----
363 #BEGIN PROGRAM
364 #-----
365
366 #####extend file objdump need for offset addr
367 command = 'objdump -h --syms -D -M intel ' + fileObjDOG + ' > OGFile.txt'
368
369 #perform the objdump
370 os.system(command)
371 time.sleep(1)
372
373 arrayCtr=0
374 #open hex file and read into array
375 fFrom = open("OGFile.txt", "r+")
376
377 for line in fFrom.readlines():
378
379     print("reading\n")
380     if textLine in line:
381         printSet += 1
382
383     #dont need to check for changes after 0x423024, specific to nano, 42302c is a ret
384     #this should prevent both verid changes and normal jmp/calls to write past here
385     elif "42302c" in line:
386         #finished checking up to 42302c"
387         break
388
389     #for nano no ext
390     elif "422fb0" in line:
391         #
392         # finished checking up to 422fb0"
393         # break
394
395     elif stopLine in line and printSet == 1:
396         break
397
398     elif printSet == 1:
399         #if line[0] == str(0):
400         if line[0:2] == "00":
401             print ("not appending symbol", line)
402
403         elif line[0] == "\n":
404             print ("found newline")
405
406         else:
407             #rid line of formatting characters
408             cleanLine = line[:-1].split('\t', 5)
409
410             #parse line into appropriate arrays
411             addrArray.append((cleanLine[0].strip()).strip(':'))
412             hexArray.append(cleanLine[1].strip())
413
414             #when just 00 on a line there is not instruction
415             if len(cleanLine) == 2:
416                 instructionArray.append("just zeros")
417             #space formatting still there, just spaces
418             else:
419                 instructionArray.append(cleanLine[2].strip())
420
421             if len(cleanLine) > 2:
422
423                 #needs to collect all above changes cuz those offsets will be greater
424                 #this goes up to not change 2 inserted cuz we know those dont use plt
425                 #but the four that shifted down may
426                 if int(addrArray[readCtr],16) <= 0x408e4a:
427                     if "$" in cleanLine[2]:
428                         #get address to jmp to out of assembly instrc, remains is just addr
429                         pltAddr = ((cleanLine[2].split('$'))[-1].strip())[:6]
430                         #add addr where this instruction w/# occurred, and the plt address
431                         #key is addr of instrc, value is plt addr
432                         commentArray[addrArray[readCtr]] = pltAddr
433                         weirdOps.add(hexArray[readCtr])
434                         weirdInst.add(instructionArray[readCtr])
435                         #just helper for visual aid
436                         forPrint.append(str(hexArray[readCtr]+"\\t\\t"+instructionArray[readCtr]))
437
438                     elif ("mov" in cleanLine[2] or "cmp" in cleanLine[2]) and ",0x4" in cleanLine[2]:
439                         #get address to jmp to out of assembly instrc, remains is just addr
440                         pltAddr = ((cleanLine[2].split(',0x'))[-1].strip())[:6]

```

```

440                                     #add addr where this instruction v/# occured, and the plt address
441                                     #key is addr of instc, value= is plt addr
442                                     commentArray[addrArray[readCtr]] = pltAddr
443                                     weirdOps.add(hexArray[readCtr])
444                                     weirdInst.add(instructionArray[readCtr])
445                                     #just helper for visual aid
446                                     forPrint.append(str(hexArray[readCtr])+"\t\t"+instructionArray[readCtr]))
447
448                                     readCtr += 1
449
450 #prints key value of all the plt's that will need change
451 #dont think need second arg
452 for item in sorted(commentArray):
453     if int(item,16) > 0x42302c:
454         break
455
456     if len(commentArray[item]) != 6:
457         continue
458
459     weirdCtr = weirdOffsets(item, weirdCtr, commentArray[item])
460
461 itemCtr = 0
462
463
464 #WORKING OFF OG FILE-----
465
466 #close opened file after finished using
467 ffrom.close()
468
469 #loop thru by checking hex instruction, if enters next inner loop contains jump or call opcode
470 for hexBytes in hexArray:
471     #look up in dict is no spcs, dict takes out spc
472     if hexBytes[:2] in opCodes.keys() or hexBytes[:5].replace(" ", "") in opCodes.keys():
473         szOpcodeFlag = 0x00
474
475         #-----worked to here, finds call/jmp codes, below is offset maths
476         #get the address current at, the address this instruction is at
477
478         #get current address from array, indexing by counter of hex loop
479         #this is instruction addr after changes
480         currentAddr = addrArray[arrayCtr]
481         lenAddr = len(currentAddr)
482
483         #add hex characteristics to string
484         currentAddr = "0x"+currentAddr
485
486         #make a hex integer, address read as int now
487         currentAddr = int(currentAddr,0)
488
489         #-----clean up offset-----
490         #get opcode, same as hexBytes?
491         #this is the offset from og next addr
492         #this is full instruction
493         ogOffset = hexArray[arrayCtr]
494
495         #get offset out, dif if 1 byte or 2 byte opcode, remove spaces
496         #flag tells how much to add to address/what addr offset change occurs
497         if hexBytes[:2] in opCodes:
498             if opCodes[hexBytes[:2]] == "1":
499                 ogOffset = ogOffset[3:].replace(" ", "")
500                 szOpcodeFlag = 0x01
501
502         else:
503             ogOffset = ogOffset[6:].replace(" ", "")
504             szOpcodeFlag = 0x02
505
506         if len(ogOffset) == 0:
507             arrayCtr += 1
508             continue
509
510         #switch endianness
511         #for 2 byte offset, offset it at next inst so acct for after this instr
512         #symtbl1 prop: 2cuz ct ea indx here
513         #num only accounts for offset bytes here, opcode will be handled w/sz flag
514         if len(ogOffset) == 4:
515             ogOffset = ogOffset[2:4] + ogOffset[0:2]
516
517         #for 3 byte
518         elif len(ogOffset) == 6:
519             ogOffset = ogOffset[4:6] + ogOffset[2:4] + ogOffset[0:2]
520
521         #for 4 byte
522         elif len(ogOffset) == 8:
523             ogOffset = ogOffset[6:8] + ogOffset[4:6] + ogOffset[2:4] + ogOffset[0:2]
524
525         #for 5 byte
526         elif len(ogOffset) == 10:
527             ogOffset = ogOffset[8:10] + ogOffset[6:8] + ogOffset[4:6] + ogOffset[2:4] + ogOffset[0:2]
528
529         #for 6 byte
530         elif len(ogOffset) == 12:
531             ogOffset = ogOffset[10:12] + ogOffset[8:10] + ogOffset[6:8] + ogOffset[4:6] + ogOffset[2:4] + ogOffset[0:2]
532
533         #for 7 byte
534         elif len(ogOffset) == 14:
535             ogOffset = ogOffset[12:14] + ogOffset[10:12] + ogOffset[8:10] + ogOffset[6:8] + ogOffset[4:6] + ogOffset[2:4] + ogOffset[0:2]
536
537         #at this point ogOffset has been flipped, stripped, and stripped, og opcode is hexBytes, str
538         #needs to happen to get into int
539         #is a str at this pt, will not add with hex
540         #ogOffset = "0x"+ogOffset
541
542         #everyone needs to be two's compliment!!!
543         #comes back here as str, humanendianess
544         #is it a dec or two's compl give decimal so we need hex of that num from 2s w/neg
545         #passed in flag cuz offset dif for 2 byters
546         #should be int and possibly negative
547         ogOffset+Math = twosComp(ogOffset, hexBytes, szOpcodeFlag)
548
549         if arrayCtr == len(addrArray) - 1:

```

```

550         break
551
552     #get next addr
553     offsetAsAddr = addrArray[arrayCtr+1]
554     offsetAsAddr = "0x"+offsetAsAddr
555     #make a hex integer, address read as int now
556     offsetAsAddr = int(offsetAsAddr,0)
557     #this begins work on math to change offset in opcode
558
559     offsetAsAddr = ogOffset4Math + offsetAsAddr
560
561     if len('0x{:x}'.format(offsetAsAddr)) > lenAddr+2:
562         #just get last chars to match addr size of prog, make str do chgs bk to hex
563         offsetAsAddr = (int("0x"+ ('0x{:x}'.format(offsetAsAddr)[-lenAddr:])) ,0)
564         #finishes as int
565
566     ogOffset = int(ogOffset, 16)
567
568     if currentAddr > 0x42302c:
569         break
570
571     #between
572     if currentAddr >= changel and currentAddr < change2:
573         extendedAddr = currentAddr + 0x04
574         #ck if changed into nop, with + change
575         checkChange(extendedAddr, "ADD")
576
577         #need to stop from writing on Ryan, one specific case, other chgs are touched
578         if extendedAddr >= 0x408e44 and extendedAddr <= 0x408e4c:
579             arrayCtr += 1
580             continue
581
582         #offset after change 2
583         if offsetAsAddr >= changel and offsetAsAddr >= change2:
584             extendedOffset = ogOffset + 0x05
585             checkChange(extendedAddr, "ADD")
586
587         elif offsetAsAddr < changel:
588             extendedOffset = ogOffset - 0x04
589             checkChange(extendedAddr, "SUB")
590
591         else:
592             extendedOffset = ogOffset
593
594         toChangeArray.append([])
595         toChangeArray[changeCounter].append(extendedAddr+szOpcodeFlag)
596         extendedOffset = (endianess(extendedOffset))
597
598         toChangeArray[changeCounter].append((extendedOffset))
599         changeCounter += 1
600
601     #below
602     elif currentAddr >= change2:
603         extendedAddr = currentAddr + 0x09
604         checkChange(extendedAddr, "ADD")
605
606         #need to stop from writing on Ryan, one specific case, other chgs are touched
607         if extendedAddr >= 0x408e44 and extendedAddr <= 0x408e4c:
608             arrayCtr += 1
609             continue
610
611         if offsetAsAddr < changel:
612             extendedOffset = ogOffset - 0x09
613             checkChange(extendedAddr, "SUB")
614
615         elif offsetAsAddr < change2:
616             extendedOffset = ogOffset - 0x05
617             checkChange(extendedAddr, "SUB")
618
619         else:
620             extendedOffset = ogOffset
621
622         toChangeArray.append([])
623         toChangeArray[changeCounter].append(extendedAddr+szOpcodeFlag)
624         extendedOffset = (endianess(extendedOffset))
625
626         toChangeArray[changeCounter].append((extendedOffset))
627         changeCounter += 1
628
629     #above
630     elif currentAddr < changel:
631         #no change
632         extendedAddr = currentAddr
633
634         #need to stop from writing on Ryan, one specific case, other chgs are touched
635         if extendedAddr >= 0x408e44 and extendedAddr <= 0x408e4c:
636             arrayCtr += 1
637             continue
638
639         if offsetAsAddr >= change2:
640             extendedOffset = ogOffset + 0x09
641             checkChange(extendedAddr, "ADD")
642
643         elif offsetAsAddr >= changel:
644             extendedOffset = ogOffset + 0x04
645             checkChange(extendedAddr, "ADD")
646
647         else:
648             extendedOffset = ogOffset
649
650         toChangeArray.append([])
651         toChangeArray[changeCounter].append(extendedAddr+szOpcodeFlag)
652         extendedOffset = (endianess(extendedOffset))
653
654         toChangeArray[changeCounter].append((extendedOffset))
655         changeCounter += 1
656
657     #need to keep track of what index, all array need to be aligned
658     arrayCtr += 1
659

```

## APPENDIX D. EXTEND AND PATCH

```
1 import argparse
2 import sys
3 from mmap editor import *
4 from shutil import copyfile
5
6
7 copyfile(sys.argv[-2], sys.argv[-1]) # create a copy of the input file to work with
8 f = openfile()
9
10 bytes_extended = 64 # extend by N bytes
11 sec_headers_dict = {}
12
13 ### Binary Patcher ###
14
15 ## Actions performed:
16 # Patch program headers that have addresses after text section (e.g. eh_frame_header)
17 # Find text section header; patch size
18 # Find all section headers with start addresses between end of text section and start of VOID (after eh_frame)
19 ## to include (for example): rodata, eh_frame_header, eh_frame
20
21 file_contents = f.read()
22
23 ## Get header info.
24 elf_header = file_contents[0:64]
25
26 prog_header_offset = elf_header[32:40]
27 prog_header_offset = int.from_bytes(prog_header_offset, byteorder='little')
28
29 sec_header_offset = elf_header[40:48]
30 sec_header_offset = int.from_bytes(sec_header_offset, byteorder='little')
31
32 prog_header_size = elf_header[54:56]
33 prog_header_size = int.from_bytes(prog_header_size, byteorder='little')
34
35 num_prog_headers = elf_header[56:58]
36 num_prog_headers = int.from_bytes(num_prog_headers, byteorder='little')
37
38 sec_header_size = elf_header[58:60]
39 sec_header_size = int.from_bytes(sec_header_size, byteorder='little')
40
41 num_sec_headers = elf_header[60:62]
42 num_sec_headers = int.from_bytes(num_sec_headers, byteorder='little')
43
44 sec_header_table_index = elf_header[62:64]
45 sec_header_table_index = int.from_bytes(sec_header_table_index, byteorder='little')
46
47
48 ## Iterate through and find desired sections.
49 for j in range(num_sec_headers):
50     current_sec_header = file_contents[sec_header_offset+sec_header_size*j:sec_header_offset+sec_header_size*j+sec_header_size]
51     sec_name_index = current_sec_header[0:4]
52     sec_name_index = int.from_bytes(sec_name_index, byteorder='little')
53     sec_headers_dict[j] = dict()
54     sec_headers_dict[j].update({"sec_name_index":sec_name_index})
55     sec_addr = current_sec_header[16:24]
56     sec_headers_dict[j].update({"sec_addr":sec_addr})
57     sec_offset = current_sec_header[24:32]
58     sec_offset = int.from_bytes(sec_offset, byteorder='little')
59     sec_headers_dict[j].update({"sec_offset":sec_offset})
60     sec_size = current_sec_header[32:40]
61     sec_size = int.from_bytes(sec_size, byteorder='little')
62     sec_headers_dict[j].update({"sec_size":sec_size})
63
64     # Find section header string table.
65     if j == sec_header_table_index:
66         shstrtab_contents = file_contents[sec_offset:sec_offset+sec_size]
67
68 ## Update section headers dictionary with section names.
69 for k in sec_headers_dict:
70     cur_index = 0
71     for byte in shstrtab_contents[sec_headers_dict[k]["sec_name_index"]]:
72         if byte == 0:
73             break
74         cur_index+=1
75     sec_name = shstrtab_contents[sec_headers_dict[k]["sec_name_index"]:sec_headers_dict[k]["sec_name_index"]+cur_index]
76     sec_name = sec_name.decode()
77     sec_headers_dict[k].update({"sec_name":sec_name})
78
79 # Locate the text section.
80 if sec_headers_dict[k]["sec_name"] == '.text':
81     print("I found the text section!")
82     text_sec_addr = sec_headers_dict[k]["sec_addr"]
83     text_sec_addr_dec = int.from_bytes(text_sec_addr, byteorder='little')
84     text_sec_start = sec_headers_dict[k]["sec_offset"]
85     text_sec_end = sec_headers_dict[k]["sec_offset"]+sec_headers_dict[k]["sec_size"]
86     print("Text section start offset: ", text_sec_start)
87     print("Text section end offset: ", text_sec_end)
88     text_sec_header = file_contents[sec_header_offset+sec_header_size*k:sec_header_offset+sec_header_size*k+sec_header_size]
89     text_sec_size = text_sec_header[32:40]
```

```

90     text_sec_size = int.from_bytes(text_sec_size, byteorder='little')
91     print("Text section size to update: ", text_sec_size)
92     new_sec_size = text_sec_size+bytes_extended
93     print("Updated text section size: ", new_sec_size)
94     new_sec_size = new_sec_size.to_bytes(8, byteorder='little')
95     replace_instructions(sec_header_offset+sec_header_size*k+32, new_sec_size)
96
97 ## Find the end of the VOID
98 for k in sec_headers_dict:
99     dec_sec_addr = int.from_bytes(sec_headers_dict[k]["sec_addr"], byteorder='little')
100     if dec_sec_addr >= text_sec_addr_dec+2097152: # 0x200000
101         print("I found the end of the VOID")
102         end_of_void = dec_sec_addr - 6291456
103         print("End of VOID: ", end_of_void)
104         break
105
106
107 ## Iterate through program headers.
108 for i in range(num_prog_headers):
109     current_prog_header = file_contents[prog_header_offset+prog_header_size*i:prog_header_offset+prog_header_size*(i+1)]
110     offset_into_file = current_prog_header[8:16]
111     offset_into_file = int.from_bytes(offset_into_file, byteorder='little')
112     vaddr = current_prog_header[16:24]
113     vaddr = int.from_bytes(vaddr, byteorder='little')
114     paddr = current_prog_header[24:32]
115     paddr = int.from_bytes(paddr, byteorder='little')
116
117     p_type = current_prog_header[0:4]
118     p_type = int.from_bytes(p_type, byteorder='little')
119     p_offset = current_prog_header[8:16]
120     p_offset = int.from_bytes(p_offset, byteorder='little')
121
122     if p_type == 1 and p_offset == 0:
123         file_size = current_prog_header[32:40]
124         file_size = int.from_bytes(file_size, byteorder='little')
125         print("File size: ", file_size)
126         new_file_size = file_size + bytes_extended
127         print("New file size: ", new_file_size)
128         new_file_size = new_file_size.to_bytes(8, byteorder='little')
129         replace_instructions(prog_header_offset+prog_header_size*i+32, new_file_size)
130         mem_size = current_prog_header[40:48]
131         mem_size = int.from_bytes(mem_size, byteorder='little')
132         print("Mem size: ", mem_size)
133         new_mem_size = mem_size + bytes_extended
134         print("New mem size: ", new_mem_size)
135         new_mem_size = new_mem_size.to_bytes(8, byteorder='little')
136         replace_instructions(prog_header_offset+prog_header_size*i+40, new_mem_size)
137
138     if offset_into_file >= text_sec_end and offset_into_file < end_of_void:
139         print("Old offset: ", offset_into_file)
140         new_offset_into_file = offset_into_file+bytes_extended
141         print("New offset: ", new_offset_into_file)
142         new_offset_into_file = new_offset_into_file.to_bytes(8, byteorder='little')
143         replace_instructions(prog_header_offset+prog_header_size*i+8, new_offset_into_file)
144         new_vaddr = vaddr+bytes_extended
145         new_vaddr = new_vaddr.to_bytes(8, byteorder='little')
146         replace_instructions(prog_header_offset+prog_header_size*i+16, new_vaddr)
147         new_paddr = paddr+bytes_extended
148         new_paddr = new_paddr.to_bytes(8, byteorder='little')
149         replace_instructions(prog_header_offset+prog_header_size*i+24, new_paddr)
150
151 ## Iterate through and find desired sections.
152 for j in range(num_sec_headers):
153     current_sec_header = file_contents[sec_header_offset+sec_header_size*j:sec_header_offset+sec_header_size*(j+1)]
154     sec_addr = current_sec_header[16:24]
155     sec_addr = int.from_bytes(sec_addr, byteorder='little')
156     sec_offset = current_sec_header[24:32]
157     sec_offset = int.from_bytes(sec_offset, byteorder='little')
158     if sec_offset < end_of_void and sec_offset >= text_sec_end:
159         print("Section name: ", sec_headers_dict[j]["sec_name"])
160         new_sec_addr = sec_addr+bytes_extended
161         print("Updating address from ", sec_addr, "to ", new_sec_addr)
162         new_sec_addr = new_sec_addr.to_bytes(8, byteorder='little')
163         replace_instructions(sec_header_offset+sec_header_size*j+16, new_sec_addr)
164         new_sec_offset = sec_offset+bytes_extended
165         print("Updating section offset from ", sec_offset, "to ", new_sec_offset)
166         new_sec_offset = new_sec_offset.to_bytes(8, byteorder='little')
167         replace_instructions(sec_header_offset+sec_header_size*j+24, new_sec_offset)
168
169
170 ### Text Extension ###
171
172 # compensate by deleting N nulls from the VOID section
173 delete_extra_bytes(end_of_void-bytes_extended, bytes_extended) # use (end_of_void - N) as offset at which to delete bytes
174
175 # extend text section by adding N nulls at the end
176 insert_instructions(text_sec_end, "00"*bytes_extended) # use text_sec_end as offset at which to add bytes
177
178 clean_up()

```

## APPENDIX E. CODE HIDER

```
1 ### To be used on a *FRESH* (unmodified) Nano to hide code in the VOID between memory segments ###
2
3 from mmap_editor import *
4 from shutil import copyfile
5
6
7 copyfile(sys.argv[-2], sys.argv[-1]) # create a copy of the input file to work with
8
9
10 bytes_to_takeover = 9 # takeover by N bytes
11 file_contents = f.read() # File already opened by mmap_editor.py
12
13 ## Get ELF header info.
14
15 elf_header = file_contents[0:64]
16
17 prog_header_offset = elf_header[32:40]
18 prog_header_offset = int.from_bytes(prog_header_offset, byteorder='little')
19
20 prog_header_size = elf_header[54:56]
21 prog_header_size = int.from_bytes(prog_header_size, byteorder='little')
22
23 num_prog_headers = elf_header[56:58]
24 num_prog_headers = int.from_bytes(num_prog_headers, byteorder='little')
25
26 ## Get program header info.
27
28 for i in range(num_prog_headers):
29
30     prog_info = file_contents[prog_header_offset*prog_header_size*i:prog_header_offset*prog_header_size*i+prog_header_size]
31     prog_info_offset = prog_header_offset+prog_header_size*i
32
33     # check p_type flag for 0x01; check p_offset for 0x00
34     p_type = prog_info[0:4]
35     p_type = int.from_bytes(p_type, byteorder='little')
36     p_offset = prog_info[8:16]
37     p_offset = int.from_bytes(p_offset, byteorder='little')
38
39     if p_type == 1 and p_offset == 0: # if we find the first LOAD section
40         print("Program header: \n")
41         file_size = prog_info[32:40]
42         file_size = int.from_bytes(file_size, byteorder='little')
43         print("File size: ", file_size)
44         new_file_size = file_size + bytes_to_takeover
45         print("New file size: ", new_file_size)
46         new_file_size = new_file_size.to_bytes(8, byteorder='little')
47         mem_size = prog_info[40:48]
48         mem_size = int.from_bytes(mem_size, byteorder='little')
49         print("Mem size: ", mem_size)
50         new_mem_size = mem_size + bytes_to_takeover
51         print("New mem size: ", new_mem_size)
52         new_mem_size = new_mem_size.to_bytes(8, byteorder='little')
53         break
54
55 #####
56 def endianness(change):
57 #switch endianness, comes in as int
58
59     if type(change) == int:
60         change = format(change, "x")
61
62     if len(change) == 1 or len(change) == 3 or len(change) == 5 or len(change) == 7:
63         change = "0"+change
64
65     #for 2 byte offset
66     if len(change) == 4:
67         change = change[2:4] + change[0:2]
68
69     #for 3 byte
70     elif len(change) == 6:
71         change = change[4:6] + change[2:4] + change[0:2]
72
73     #for 4 byte
74     elif len(change) == 8:
75         change = change[6:8] + change[4:6] + change[2:4] + change[0:2]
76
77     #for 5 byte
78     elif len(change) == 10:
79         change = change[8:10] + change[6:8] + change[4:6] + change[2:4] + change[0:2]
80
81     #for 6 byte
82     elif len(change) == 12:
83         change = change[10:12] + change[8:10] + change[6:8] + change[4:6] + change[2:4] + change[0:2]
```

```

84
85     #for 6 byte
86     elif len(change) == 14:
87         change = change[12:14] + change[10:12] + change[8:10] + change[6:8] + change[4:6] + change[2:4] + change[0:2]
88
89     return(change)
90
91
92 replace_instructions(prog_info_offset+32, new_file_size)
93 replace_instructions(prog_info_offset+40, new_mem_size)
94
95
96 start_of_void = file_size + 0x400000          # start of VOID section; place to insert me1
97 me2_addr = start_of_void + 13                # place to insert me2
98 print("Start of void: ", start_of_void)
99 print("me2 address: ", me2_addr)
100
101 void_offset_me1 = start_of_void - 0x408e51    # offset to write for call to me1
102 void_offset_me1 = endianness(void_offset_me1) # stringify it
103 if len(void_offset_me1) == 6:
104     void_offset_me1 = void_offset_me1+"00"    # zero fill to 4 bytes
105 print("Void offset for me1: ", void_offset_me1)
106
107 void_offset_me2 = me2_addr - 0x408900         # offset to write for jump to me2
108 void_offset_me2 = endianness(void_offset_me2) # stringify it
109 if len(void_offset_me2) == 6:
110     void_offset_me2 = void_offset_me2+"00"    # zero fill to 4 bytes
111 print("Void offset for me2: ", void_offset_me2)
112
113 '''
114 Preparation for call to me2.
115     jump          Change return to jump.
116     423049        Insert jump address to me2.
117 '''
118 replace_instructions(0x4088fb, "e9")
119 insert_instructions(0x4088fc, void_offset_me2)
120
121 '''
122 Preparation for call to me1.
123     push 4087e0    Save real call address.
124     call 42303c    Call me1.
125 '''
126 insert_instructions(0x408e47, "68e0874000")
127 replace_instructions(0x408e4c, "e8"+void_offset_me1) # added 5 to address to account for inserted instructions before
128
129 '''
130 Write me1.
131     pop r8        Save real return address. (408e48)
132     pop r9        Save real call address. (4087e0)
133     xor r8, 0x2a   Mask return address.
134     push r8        Replace return address.
135     jump r9        Jump to real call address.
136 '''
137 replace_instructions(start_of_void, "415841594983f02a415041ffe1") # shifted left by 1 byte
138
139 '''
140 Write me2.
141     pop r8        Retrieve masked return address.
142     xor r8, 0x2a   Unmask return address.
143     push r8        Replace real return address.
144     ret           Return to real return address.
145 '''
146 replace_instructions(me2_addr, "41584983f02a4150c3")
147
148 '''
149 Delete extra bytes in VOID to account for those added with insert.
150 '''
151 delete_extra_bytes(me2_addr+9, 9)

```

## LIST OF REFERENCES

- [1] S. Cass, “The 2018 top programming languages,” *IEEE Spectrum*, July 31, 2018. [Online]. Available: <https://spectrum.ieee.org/at-work/innovation/the-2018-top-programming-languages>
- [2] D. A. M. Trejo, “After all these years, the world is still powered by C programming,” Toptal. Accessed July 23, 2019. [Online]. Available: <https://www.toptal.com/c/after-all-these-years-the-world-is-still-powered-by-c-programming>
- [3] D. Bolton, “State of C programming language in 2019,” Dice, January 14, 2019. [Online]. Available: <https://insights.dice.com/2019/01/14/state-of-c-programming-language-in-2019/>
- [4] V. Choudhary, “C programming language version history,” Developer Insider. Accessed July 23, 2019. [Online]. Available: <https://developerinsider.co/c-programming-language-version-history/>
- [5] D. M. Ritchie, “The development of the C language\*,” 2003. [Online]. Available: <https://www.bell-labs.com/usr/dmr/www/chist.html>
- [6] D. Kalev, “C11: a new C standard aiming at safer programming,” Smartbear, June 25, 2012. [Online]. Available: <https://smartbear.com/blog/test-and-monitor/c11-a-new-c-standard-aiming-at-safer-programming/>
- [7] J. P. Anderson, “Computer security technology planning study,” Hanscom AFB, Bedford, MA, ESD-TR-73-51, ESD/AFSC, 1972. [Online]. Available: <http://seclab.cs.ucdavis.edu/projects/history/papers/ande72a.pdf>
- [8] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, “Return-oriented programming: systems, languages, and applications,” *ACM*, vol. 15, no. 2, March 2012. [Online].  
doi: 10.1145/2133375.2133377
- [9] K. S. Lhee and S.J. Capin, “Buffer overflow and format string overflow vulnerabilities,” S.P. & E., August 31, 2002. [Online]. Available: <https://surface.syr.edu/cgi/viewcontent.cgi?article=1095&context=eecs>
- [10] E. Levy, “Smashing the stack for fun and profit,” *Phrack*, November 08, 1996. [Online]. Available: <http://phrack.org/issues/49/14.html#article>



- [11] D. Evtvyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "Jump over ASLR: attacking branch predictors to bypass ASLR," in *2016 49th Annual IEEE/ACM Internatl. Symp. On Microarchitecture*, 2016. [Online.] doi: 10.1109/MICRO.2016.7783743
- [12] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin, "Securing untrusted code via compiler-agnostic binary rewriting," in *ASAC '12 Proc. of the 28th Annual Comp. Sec. App. Conf.*, 2012. [Online]. doi: 10.1145/2420950.2420995
- [13] A. J. Boulton and B. J. Godwood, "Binary image stack cookie protection," September 12, 2018. [Online]. Available: <http://www.freepatentsonline.com/y2019/0205526.html>
- [14] H. Marco-Gisbert, "SSPFA: stack smashing protection for android OS," January 22, 2019. *I. J. I. S.*, vol. 18, is. 4, pp. 519–532, [Online]. Available: <https://link.springer.com/article/10.1007/s10207-018-00425-8>
- [15] H. P. Joshi, A. Dhanasekaran, and R. Dutta, "Impact of software obfuscation on susceptibility to return-oriented programming attacks," in *2015 36th IEEE Sarnoff Symposium*, 2015. [Online.] doi: 10.1109/SARNOF.2015.7324662
- [16] C. Cowan, S. Beattie, J. Johansen, and P. Wagle, "PointGuard: protecting pointers from buffer overflow vulnerabilities," *Proc. of the 12th Conf. on USENIX Sec. Symp.*, vol. 12, pp.91-104, August 8, 2003. [Online]. Available: [https://www.usenix.org/legacy/events/sec03/tech/full\\_papers/cowan/cowan.pdf](https://www.usenix.org/legacy/events/sec03/tech/full_papers/cowan/cowan.pdf)
- [17] J. Erickson, *Hacking: The Art of Exploitation*. San Francisco, CA, USA: No Starch Press, 2008.
- [18] O. Andreeva, S. Gordeychik, G. Gritsai, O. Kochetova, E. Potseluevskaya, S. I. Sidorov, and A. A. Timorin, "Industrial control systems vulnerabilities statistics," Kaspersky Lab., Woburn, MA, USA, 2016. [Online]. Available: [https://media.kasperskycontenthub.com/wpcontent/uploads/sites/43/2016/07/07190426/KL\\_REPORT\\_ICS\\_Statistic\\_vulnerabilities.pdf](https://media.kasperskycontenthub.com/wpcontent/uploads/sites/43/2016/07/07190426/KL_REPORT_ICS_Statistic_vulnerabilities.pdf)
- [19] MITRE, "Common weakness enumeration," June 20, 2019. [Online]. Available: <https://cwe.mitre.org/index.html>
- [20] MITRE, "Common vulnerabilities and exposures," August 19, 2019. [Online]. Available: <https://cve.mitre.org/index.html>
- [21] Towson University, "Buffer overflow – "data gone wild"," Accessed June 30, 2019. [Online]. Available: <http://cis1.towson.edu/~cssecinj/modules/cs0/buffer-overflow-cs0-java/>
- [22] J. R. Levine, "Linkers and loaders," San Francisco, California, USA: Morgan Kaufmann, 1996.

- [23] *Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification*, 1.2, 1995. [Online]. Available: <https://refspecs.linuxbase.org/elf/elf.pdf>
- [24] Apple Incorporate, “Apple public source license,” Version 2.0, August 6, 2003. [Online]. Available: <http://www.opensource.apple.com/apsl/>
- [25] Lawrence Livermore National Laboratory. Livermore, CA, USA. 2007. Rose Compiler Infrastructure. ver. 0.9.9.35. [Online]. Available: <http://rosecompiler.org/>
- [26] Python, “Memory-mapped file support,” August 24, 2019. [Online]. Available: <https://docs.python.org/3/library/mmap.html>
- [27] 0x90, “Delete / insert data in mmap’ed file,” Stack Overflow, May 8, 2011. [Online]. Available: <https://stackoverflow.com/questions/5917047/delete-insert-data-in-mmaped-file>
- [28] S. J. Vaughan-Nichols, “The most popular U.S. end-user operating systems, according to the federal government,” ZD Net, March 27, 2015. [Online]. Available: <https://www.zdnet.com/article/the-federal-government-on-what-are-the-most-popular-us-end-user-operating-systems/>

THIS PAGE INTENTIONALLY LEFT BLANK

## **INITIAL DISTRIBUTION LIST**

1. Defense Technical Information Center  
Ft. Belvoir, Virginia
2. Dudley Knox Library  
Naval Postgraduate School  
Monterey, California