Code-Stop: Code-Reuse Prevention By Context-Aware Traffic Proxying

Terrence OConnor and William Enck
Department of Computer Science
North Carolina State University
Raleigh, North Carolina, USA
Email: tjoconno@ncsu.edu, enck@cs.ncsu.edu

***Abstract*—This paper introduces a network and host-based cooperative system for defending against code-reuse attacks that bypass exploit mitigation strategies. While the combination of address space layout randomization (ASLR) and data execution prevention (DEP) provide the means for mitigating exploitation, attackers routinely bypass these mechanisms by borrowing code from shared libraries that lack the same protections or by abusing memory leaks. This paper illustrates the ability to identify code-reuse attacks through cooperation between the traffic proxy and destination host. With the context of the host, the network has the ability to prevent code-reuse, and ultimately, exploitation. Through experimentation, we demonstrate that our cooperative system can effectively defeat a wide variety of code-reuse attacks, including newer attack vectors such as Just-in-Time-Flash or jump-oriented gadgets. Our experiments indicate our prototype is compatible with popular software such as Internet Explorer, Adobe Reader, and Microsoft Office applications and proved successful mitigating code-reuse attacks.**

Keywords–code-reuse attacks; return-oriented programming; intrusion prevention system; proxy.

I. INTRODUCTION

Identifying and defending against exploits in the wild is an ongoing challenge. While system protections such as Address Space Layout Randomization (ASLR) and Data Execution Prevention (DEP) make vulnerabilities more difficult to exploit, they can be bypassed using code-reuse attacks (e.g., Return Oriented Programming [ROP], Jump Oriented Programming [JOP], SigReturn Oriented Programming [S-ROP], and Just In Time Return Oriented Programming [JIT-ROP]).

In 2012, Microsoft's BlueHat Challenge [1] awarded over a quarter of a million dollars to three solutions that defended against ROP. Within a year, Shacham et al. [2] successfully bypassed every ROP protection that had been awarded a prize. Additionally, after Microsoft integrated the BlueHat Challenge winner solutions into their commercial product The Enhanced Mitigation Experience Toolkit (EMET), DeMott [3] demonstrated separate methods for bypassing all twelve system protections included in EMET. Most defense mechanisms have focused exclusively on the host by either compiling gadget-free binaries, protecting critical functions [4], or performing runtime randomization [5], control flow analysis [6], or system caller checking [7] [8] [9]. However, all have shown weaknesses by making the potential victim responsible for managing its own safeguards.

In contrast to these prior approaches, we propose a system for defending against code-reuse attacks that pushes the defense to the network but still relies on the host to provide the active context of the attack. We focus exclusively on preventing the code-reuse attacks (e.g., ROP, JOP, S-ROP, JIT-ROP) used to bypass system protections (e.g., ASLR, DEP). The key idea behind our solution is the concept of *parameter suspicion*. We make the assumption that a code-reuse attack will be used to bypass memory protections by allocating a protected region of memory as executable. Further, we make the assumption that a code-reuse attack will borrow code in order to execute a memory-related function. Instead of monitoring the specific function calls that may allocate or change memory protections, we attempt to identify the parameters used by the function call. Parameter suspicion identifies the generic behavior that occurs prior to a code-reuse attack. It identifies when the attacker abuses memory to load data into registers, functioning as parameters for a call to a critical system call or function.

In this paper, we propose and implement Code-Stop, a collaborative system that protects an application from code-reuse attacks that bypass system protections. Code-Stop relies on parameter suspicion to identify a code-reuse attack. We implement Code-Stop on top of an existing traffic proxy in communication with destination hosts. We observe that Code-Stop can prevent modern client-side attacks at the network layer only with the emulated context being provided by the host. We demonstrate that prevention can occur with minimal overhead by reducing the critical area of traffic that must be tested. Our prototype scans for potential code-reuse attacks in PDF documents, JavaScript, Adobe Flash, and Microsoft Office documents, which account for 72.9% of vulnerable file types [10].

This paper makes the following contributions:

- We design and implement Code-Stop to protect against the broader threat of client-side attacks that use code-reuse attacks. Code-Stop allows the traffic proxy to make context-aware decisions about malicious traffic based on the emulated impact on the destination host.
- We propose the technique of *parameter suspicion* to identify code-reuse attacks with low false positives. Parameter suspicion emulates potential gadgets to determine the impact on the general purpose registers used as parameters when calling Windows API functions. Specifically, parameter suspicion identifies the parameters used for a Windows API function that allocates or changes memory as executable.
- We evaluate the accuracy, performance overhead, scalability and coverage of Code-Stop. We observe Code-Stop's ability to prevent code-reuse attacks without producing false positives with a large set of known malware-free files. Code-Stop's performance overhead and delay scales with typical proxy configurations and anti-virus scanning proxy solutions. We evaluate that Code-Stop can detect a wide range of code-reuse attacks without modification.

ISBN: 978-1-61208-475-6

The remainder of this paper is as follows. Section II provides a background on memory protection mechanisms and code-reuse attacks. Section III examines the challenges with preventing code-reuse attacks. Section IV provides an overview of our solution. Section V examines the design of our prototype solution. Section VI evaluates our prototype, Code-Stop. Section VII discusses the limitations and future work. Section VIII discusses recent related work in the field of preventing code-reuse attacks, and Section IX concludes.

II. Background And Motivation

To understand the defense system presented in this paper, we must review common mitigation strategies, including ASLR, DEP, and compile-time, run-time, and network-layer exploit prevention mechanisms.

Address Space Layout Randomization: Implemented in early 2002, ASLR provided one of the earliest means to decrease the effectiveness of an exploit. ASLR prevents an attacker from using a predictable and pre-calculated virtual address for code reuse in an exploit. ASLR can randomize the location of code by randomizing the starting address of dynamic-linked libraries, the base address of the heap, or the location of routines and static data in the executable [11] [12] [13]. Initial research targeting ASLR implementations studied the effectiveness of defeating the entropy of code randomization [2]. However, attackers found it far more useful to bypass ASLR entirely. Notably, the initial release of Windows Vista Service Pack 0 only randomized the base address of executables and dynamic link libraries [14]. The poorly implemented Vista design effectively allowed attackers to bypass ASLR by partially overwriting the address offset without overwriting the randomized base address. Another common means for bypassing ASLR borrows code from dynamic link libraries (DLL) that lack ASLR. Several recent attacks in the wild have relied upon using DLLs without ASLR [15] [16]. Attackers routinely execute these attacks by forcing the application to load a DLL that implements extra functionality.

Data Execution Prevention: The 2004 release of Windows XP Service Pack 2 introduced the DEP security feature [17] [18]. In Windows OS, hardware DEP works similar to Linux W$\oplus$X, which uses the non-executable (NX) bit to mark memory as executable. Under W$\oplus$X or DEP, memory may be executable or writable, but not both [19]. This isolation of memory mitigates control flow hijacking by preventing stack-based buffer overflows. Combined with ASLR, DEP defeated control flow hijacking on the Windows OS until Shacham [20] and Litchfield [17] proposed the first code-reuse attacks.

Code-Reuse Attacks: In 2005, Litchfield proposed the first means of defeating DEP by returning to the VirtualAlloc() function [17]. Litchfield borrowed heavily from a Linux technique known as to return to libc, which replaced the return address on the stack with the address from a function call borrowed from the libc library. Return-Oriented-Programming (ROP) expands upon return to libc by chaining a series of borrowed code snippets together to execute a specific purpose. Under Windows, ROP often overwrites the return address with a chain of addresses that point to borrowed code inside shared libraries. ROP chains these borrowed fragments of code together to disable the DEP security mechanisms [21]. While the application's sandbox may implement memory protections, attackers often dynamically load shared libraries in order to borrow code and escape the sandbox of protection.

Under ROP, each small fragment (gadget) borrows a small piece of code that is followed by a return. The variable x86 instruction length eases the difficulty of gadget discovery, since gadgets can be borrowed from the offset of a logical address. The assembly of gadgets typically disables protection mechanisms in order to allow malicious shell-code to execute. This commonly involves placing specific values into general purpose registers before calling a Windows API function that disables the security protection of DEP.

Our solution to preventing code-reuse relies on the fact that an attacker must use gadgets to load these values into registers before calling the API function. In this way, we can observe the generic behavior in order to identify attacks. The next section examines the challenges in preventing against code-reuse attacks.

Host and Network Based Cooperative Defense: In Section VIII, we discuss the shortcomings of previous host-only or network-only defense mechanisms. We argue that the shortcomings of the host-layer and network-layer defenses can be addressed by using the network to defend with the context of the host. Previous work has examined emulating arbitrary data at the network to determine if it is part of a payload of an attack. However, code-reuse attacks rely on borrowing code from specific memory addresses. Since these addresses vary between application and operating system versions, the network must be aware of the dynamic context of the host to successfully identify a code-reuse attack.

III. Challenges

Our prototype, Code-Stop, hardens the security of the host under protection and provides new opportunities to identify attacks in progress. However, enabling this protection introduces challenges that we address:

C-1 *Ability to identify dynamic sandbox escapes.* One method commonly used by attackers forces an application to load a library lacking the same protection mechanisms as the application, so that the attacker escapes the sandbox of protection. This can be seen in recent attacks against Internet Explorer (with the `hxds.dll` bypass) and Adobe Reader (with `icuncnv36.dll` bypass.) [22], [23] Content that forces dynamic loading presents an interesting challenge for a host-based context emulator. In Section V-C, we introduce the concept of *disarmed reading*, which removes the content for a code-reuse attack and allows the application to determine whether the suspected file forces the loading of a shared library without ASLR. Preventing dynamic attacks is where our approach notably differs from previous approaches [24].

C-2 *Ability to seamlessly protect different application and operation system versions and configurations.* The address space layout protection mechanisms and fixed address space used in various bypass mechanisms differ within versions and configurations. Therefore, the proxy must dynamically construct the context of the attack to detect attacks and not be overwhelmed with a range of false positives. In Section V-B, we outline the design of the *Parameter suspicion* technique that uses the context gained

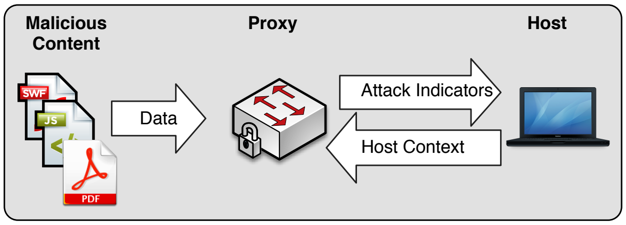


Figure 1. Overview of Code Stop

from the host to make an informed decision about the malicious nature of content.

C-3 *Impact on application performance.* Security and performance must always be closely balanced. Code-Stop protects client-side applications (Adobe Reader, Internet Explorer, Microsoft Office) from attacks where the content is rendered over the network. In Section V-A, we introduce the concept of parsing and scanning only the most absolutely necessary critical-space in affected file-formats (e.g., HTML, SWF, PDF, and DOCX).

IV. Approach

Our goal is to prevent the exploitation of a client side application by identifying the presence of code-reuse attacks in arbitrary data, such as an HTML document, PDF document stream, or Flash object. We focus on these document types since they account for 72.9% of malicious documents used in exploit kits in 2014 [10]. To successfully identify an attack, each input must be inspected with regard to the actual file format and structure to reconstruct how that data would be allocated into application memory. Figure 1 depicts the high level overview of our Code-Stop prototype. Our prototype is implemented on a network proxy, that scans suspected files to identify valid addresses that correspond to code-reuse attacks. Our approach differs from previous approaches because it includes dynamic context from the host under attack to determine if suspected data is part of a code-reuse attack.

Our hybrid approach combines the benefits of both the network layer and host layer to effectively identify and mitigate code-reuse attacks. We discuss the benefits of several host-based defenses [7] [8] [25] [26] and network-based defenses [24] [27] [28] [29] in Section VIII. However, few works have examined the benefits of combining the network and host together in defense of code-reuse attacks. Tzermias et al. [30] presented a method for the identification of ROP payloads in arbitrary data such as network traffic. However, their work failed to address the dynamic context of the host. The dynamic context of the host proves extremely important to monitor as an application can be forced to load shared libraries by processing an arbitrary document. Our work improves upon the design of Tzermias et al. by adding dynamic context of the host. Several optimizations arise out of our shared approach from Tzermias et al. By adding the network layer into the host defense, we store the record of known malicious documents. Further, caching the result of known-malicious documents allows the network layer to extend protection to hosts without our prototype software.

Assumptions and Threat Model:

We make two general assumptions in our prototype design:

1) We implement our Code-Stop prototype on the 32-bit architecture instruction set. We make the assumption that expanding our prototype to support a 64-bit architecture will only decrease the probability of false alarms. Section VI-B discusses the probability of false alarms and further explains this assumption. Further, the vast availability of exploits and ROP Chains for 32-bit applications provided for better testing of our prototype.
2) We do not address de-obfuscation as a topic for this paper. Rather, our prototype relies upon pdf-parser [31] and jsunpack [32] as means for de-obfuscating content. We make the general assumption that the proxy can de-obfuscate content or simply block heavily obfuscated content as already malicious in nature. Previous works have addressed de-obfuscating malicious code from PDF documents [33] and browser downloads [34]. Further, Section V-A discusses Code Stop's design for parsing content, supporting this assumption. Further, we expand on the limitation of de-obfuscation in Section VII. Future work may examine de-obfuscation of malicious content and the likelihood obfuscated content is benign.

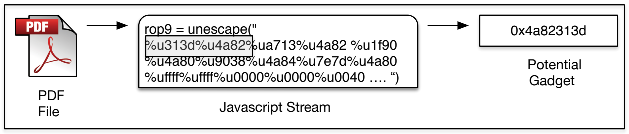
We make the following assumptions in our threat model. The adversary can exploit (i.e., control the flow of execution) of a client application (under our protection). Further, the adversary has the ability to read or infer randomized memory from those binary and shared libraries. However, we assume the attacker must bypass both DEP and ASLR to complete their exploit and execute a payload (e.g., download a remote access toolkit, add a user, or disable processes.) The trusted computing base (TCB) includes the network proxy software that parses the potential gadgets and the host application that determines the context of the gadgets. We trust that the context determined by the host application has not been altered by a malicious administrator. Finally, we assume that a trusted network channel exists between the host and the network proxy. The channel is available and preserves the traffic integrity between the proxy and the host. Given the above-described challenges, assumptions and threat model, the next section examines the design in detail.

V. Design

The following section describes the design of our prototype, Code-Stop. In Section V-A, we address how the network proxy parses the critical space of files to identify gadgets used in Code-Reuse attacks. Section V-B proposes the concept of *parameter suspicion* to identify code-reuse attacks in progress by using the emulated context of the host. Section V-C details *disarmed reading*, which disarms a malicious file in order to safely identify mitigation bypass techniques. Section V-D provides an example to illustrate how our prototype prevents an attack. Finally, Section V-E discusses how to extend Code-Stop to identify other generic exploit behaviors and operating systems.

A. Parsing Potential Gadgets

In our prototype design, the network proxy runs an application that parses arbitrary data for indicators of a potential code-reuse attack. By de-obfuscating content and further de-compiling and de-constructing specific file formats, the prototype looks for arbitrary data that represents 32-bit memory



```
rop9 = unescape("
%u313d%u4a82%ua713%u4a82 %u1f90
%u4a80%u9038%u4a84%u7e7d%u4a80
%uffff%uffff%u0000%u0000%u0040 …. “)
```

Figure 2. PDF streams containing ROP gadgets

addresses or references to a variable that would contain a 32-bit address. A 32-bit memory address can be constructed many ways in memory and our parser matches regular expressions for several methods for allocating arbitrary data as 32-bits.

Consider Figure 2 for how our prototype parses a potential gadget out of a PDF File. In the example, our network proxy parser application uses pdf-parser [31] to read the embedded JavaScript inside the document. The parser then matches a regular expression for a unicode string containing two characters. Since unicode strings are 16-bits wide, two unicode characters are commonly used by attacks to represent a 32-bit memory address. The parser matches the potential gadget 0x4a82313d against the unicode string %u313d %u4a82 and passes the potential gadget to the paramater suspicion classifier.

For our prototype, we implemented the parser to examine browser traffic, including Internet Explorer Javascript, Adobe Flash Vector Objects (Actionscript) and PDF Files containing Javascript. While there are several other client side applications - we implemeted our prototype to cover the client applications that presented the largest surface area for recent exploits seen in the wild. In 2015, Trend Micro observed that Internet Explorer, Flash, and Adobe PDFs vulnerabilities accounted for 72.9% of the exploits used in the top nine exploit kits [10].

Within those specific file formats, the parser matches JavaScript's binary strings (BSTR) and Adobe Flash Objects containing ActionScript code. As illustrated with the PDF JavaScript, an attacker can cleanly construct a gadget for Internet Explorer JavaScript with a binary string of two unicode characters that refer to a 32-bit address. Although Internet Explorer 9 removed BSTR functionality, Yu [35] discovered a method for calling the earlier version of JavaScript on newer browsers by adding an HTML compatibility tag. Our prototype also decompiles Adobe Flash files and matches the embedded ActionScript code for gadgets. ActionScript code offers similar means to allocate arbitrary data as 32-bit addresses [36]. With this understanding of how our Code-Stop prototype parses allocated gadgets, the next section examines how Code-Stop detects the generic behavior of code-reuse attacks.

B. Parameter Suspicion Classifier

In order for a code-reuse attack to allocate or change memory protections, it must make a call to a Windows API function that manages memory (e.g., VirtualAlloc, VirtualProtect, SetInfoProcess.) Calling functions that allocate or change memory require parameters such as the location of memory, size, and bitwise value for new memory protections. These parameters must be placed into data registers in order to be pushed onto the call stack as parameters to the function. Rather than targeting specific API function calls, we propose the idea of a *parameter suspicion* classifier. The parameter suspicion classifier runs entirely on the host to emulate potential gadgets. Parameter suspicion determines if the instructions from the suspected gadgets load values into multiple general purpose registers. Parameter suspicion overcomes the limitations of most code-reuse defenses by identifying the anomalous characteristics of a bypass, rather than looking for specific dangerous function or system calls. We expand upon how and why it works, and the probability for false positives.

TABLE I. REGISTER VALUES FOR COMMON ROP CHAINS

Register	VirtualAlloc()	SetInfoProcess()	VirtualProtect()
EAX	Ptr to VirtualAlloc	SizeOf 0x00000004	Ptr to VirtualProtect
EBX	dwSize 0x00000001	NtCurrentProcess 0xffffffff	dwSize 0x00000001
ECX	flProtect 0x00000040	&ExecuteFlags Ptr to 0x00000002	Writable Address
EDX	flAllocationType 0x00001000	ProcessExecuteFlags 0x00000022)	NewProtect 0x00000040

To understand how parameter suspicion works, consider Figure 3. The ROP gadgets placed onto the stack refer to instructions in the `msvcr71.dll` used by Java 1.6. The gadgets are part of a larger chain used to execute the VirtualProtect() function. After the proxy has parsed and sent potential gadgets to the host, the host emulates the effects and determines the chain loaded values into the `ECX, EBX`, and `EDX` registers. The key insight of parameter suspicion is that we do not need to trap the exact jump or call to the VirtualProtect() function. Rather, we can detect the gadgets placing parameter values into ECX, EBX, and EDX, respectively, to match the lpAddress, dwSize, and flNewProtect parameters required by VirtualProtect(). Using our technique we can identify any generic function call used to bypass DEP/ASLR.

We expand this understanding to other critical functions used to bypass DEP. Table I depicts the register values allocated for the VirtualAlloc(), SetInfoProcess(), and VirtualProtect() functions from ROP Chains generated by the Mona.py toolkit (a commonly used tool to automatically build ROP Chains) [37]. Note that each function requires a minimum of three parameters in addition to a pointer to the critical function. In fact, most of the 50 critical functions checked by EMET require a minimum of three parameters. Functions that allocate new memory and mark it as executable (e.g., VirtualAlloc, VirtualProtect) commonly use parameters of the following form: (1) a bitwise value for new memory protections, (2) an address to which one might write shellcode, and (3) a size of the memory allocated to the new region. Ultimately, all functions that must accomplish anything of value require multiple parameters.

Parameter suspicion does not produce significant false positives, because the probability that a data value matches a gadget address is very small. Consider the protection of Adobe Reader as an example. Assume that `icucnv36.dll` is the only current library available to bypass ASLR and DEP for the Adobe Reader application. The probability that an arbitrary eight character string corresponds to a single `POP EAX; RET` sequence of instructions is represented in Fig. 1.

$$P(R_{EAX}) = \left(\frac{22}{94}\right)^8 \cup \left(\frac{294,912}{2^{32}}\right) \cup \left(\frac{28}{294,912}\right) \qquad (1)$$

We calculate this probability given that only 94 print-

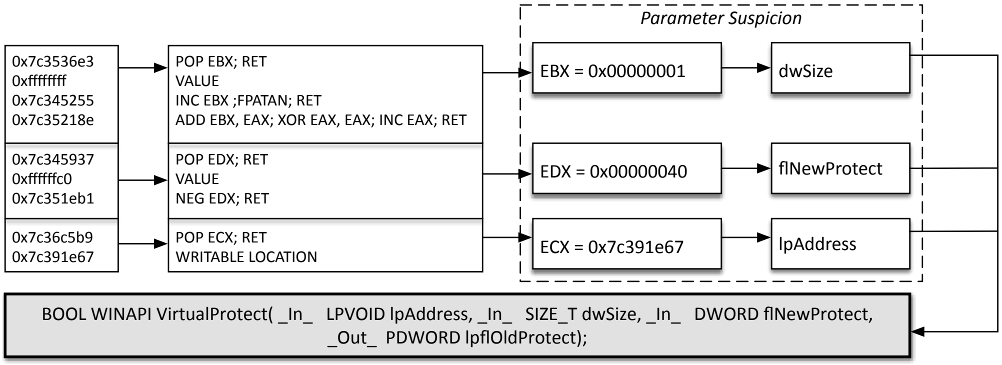


Figure 3. Parameter Suspicion used to identify msvcr71.dll ROP chain

TABLE II. LOADCONSTANT GADGETS FROM COMMON DLLS

DLL	Application	Size (Bytes)	Pop EAX	Pop EBX	Pop ECX	Pop EDX
icucnv36.dll	Adobe Reader	294,912	28	455	237	1
vgx.dll	Adobe Flash	732,672	55	699	146	3
msvcrt.dll	Visual C++ Runtime	184,320	86	358	164	22
msvcr71.dll	Java	233,472	46	234	258	21
hxds.dll	MS Office	564,224	33	481	345	7
PEhelper.dll	IBM Forms	103,936	6	78	74	0

able ASCII characters exist and only 22 of them (0-9,A-F,a-f) correspond to a memory address space. Further, the `Icucnv36.dll` only has 294,912 unique memory locations that point to executable code. And finally, only 28 of those unique locations point to a `POP EAX; RET` gadget. Our parameter suspicion classifier only matches when a chain of gadgets loads constants into three or more separate memory registers. While other operands exist (`XCHG, MOV`), the probability remains extremely small that three of these instructions will be randomly constructed from an arbitrary data stream.

To understand how this applies to other applications, examine the results in Table II. These results show the frequency of `POP REG; RET` instructions in some common shared libraries discovered by the the Metasploit *msfrop* tool (a common tool used by hackers to find instructions for code-reuse attacks.) Next, we examine how *disarmed reading* augments our classifier to determine when exploits attempt to bypass standard mitigations by loading shared libraries at runtime.

C. Disarmed Reading

One method commonly used by attackers forces an application to load a library lacking the same protection mechanisms as the application, so that the attacker escapes the sandbox of protection. Content that forces dynamic loading presents an interesting challenge for a host-based context emulator. The emulator must be aware of the addresses of all shared libraries that can be loaded dynamically by arbitrary data. *Disarmed reading* allows parameter suspicion to determine when an arbitrary file forces a protected application to load a shared library without the same protection mechanisms (e.g., a library without ASLR). To achieve this effect, disarmed reading removes suspected gadgets from the formatted file and allows the host to render the file hidden to the user. This removal includes PDF streams, JavaScript memory allocations of BSTRs, and ActionScript dynamic content. Disarmed reading allows the host to safely inspect a disarmed file to understand if it loads any shared libraries that lack the protection mechanisms of the application. We expand upon disarmed reading using two recent mitigation bypasses that highlight its necessity.

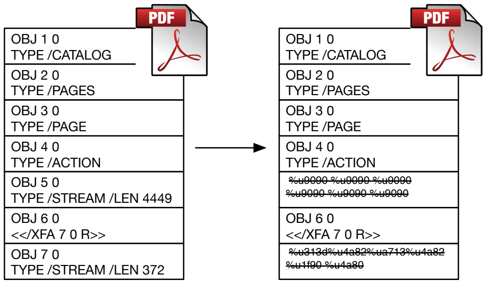
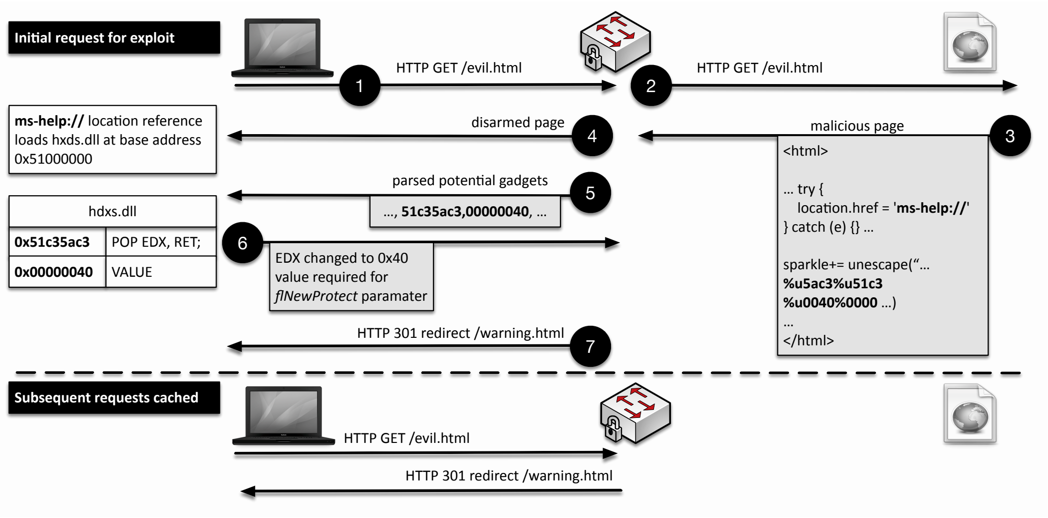


Figure 4. Disarmed reading of a malicious PDF

The first example considers the case where Adobe Reader suffers from a similar bypass technique. A properly crafted XFA tag within a PDF document can force the Adobe Reader application to load `icucnv36.dll`, which lacks ASLR [38]. For parameter suspicion to identify the gadget, it must be aware that the formatted file has loaded the additional shared library. Figure 4 depicts how disarmed reading handles loading a malicious PDF. The suspected file contains seven PDF objects: a catalogue, two pages, an action, two streams, and an object containing the XFA tag. Code-Stop removes the two streams when disarming the file, since streams prove to be common locations for ROP chains and shellcode. However, Code-Stop leaves the other objects intact. Object 6 0 contains the XFA tag that forces the Adobe Reader application to load `icucnv36.dll`. This results in disarmed reading learning of the base address of the shared library without ASLR. It further provides this information to parameter suspicion in order for it to have the full context of addresses that code exists at for the application under protection.

The second example considers the protection mechanisms of Internet Explorer, which can be bypassed by loading the `hxds.dll` by making a location reference to ms-help [22]. Because `hxds.dll` lacks ASLR, exploits can use fixed addresses within `hxds.dll` to construct an ROP chain capable of bypassing DEP. Parameter suspicion requires knowledge of what code exists at specific memory locations. Without knowing that an exploit forced Internet Explorer to load `hxds.dll` at the base address of 0x51BD0000, parameter suspicion cannot determine the emulated effect of any gadget. Ultimately, Code-Stop allows a disarmed version of an HTML document through the proxy such that `hxds.dll` loads, but does not contain any dynamic content that could be used to exploit the application.


Initial request for exploit
HTTP GET /evil.html
1
HTTP GET /evil.html
2
ms-help:// location reference loads hxds.dll at base address 0x51000000
disarmed page
4
malicious page
3
parsed potential gadgets
5
..., 51c35ac3,00000040, ...
hdxs.dll
0x51c35ac3
POP EDX, RET;
0x00000040
VALUE
6
EDX changed to 0x40 value required for flNewProtect paramater
HTTP 301 redirect /warning.html
7
Subsequent requests cached
HTTP GET /evil.html
HTTP 301 redirect /warning.html

```
<html>

... try {
   location.href = 'ms-help://'
} catch (e) {} ...

sparkle+= unescape("...
%u5ac3%u51c3
%u0040%0000 ...)
...
</html>
```

Figure 5. Example Detection of the Dynamically Loaded hxds.dll ROP Chain

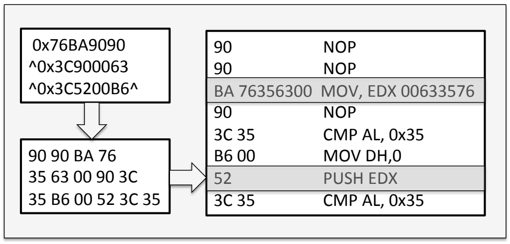
D. Detection of an Example Attack

Figure 5 depicts how our prototype prevents an example attack. (1) First, the victim requests the webpage evil.html that contains an exploit with a VirtualProtect() ROP chain that loads at runtime. (2) The proxy requests the webpage on behalf of the client and (3) receives the html page (4) The proxy then sends a disarmed form of the page to the client to determine what additional libraries might be loaded when the page loads. In this case, the location reference for ms-help:// loads a shared library without ASLR protection. (5) The proxy parses values that may be addresses of gadgets or part of a ROP Chain (e.g., - 0x51c35ac3,0x00000040). These potential gadgets are sent to the host to determine their context. (6) The host replies with the impact of the ROP Chain on the memory registers. In this case, we demonstrate the part of the ROP Chain that loads the value 0x40 into EDX as a parameter for VirtualProtect(). (7) After classifying the malicious impact of the ROP Chain, the proxy replies with a HTTP 301 redirect to a warning page. On subsequent requests - the proxy replies with the HTTP 301.

E. Extending Code-Stop

We now examine how the design of Code-Stop allows us to identify the distinct JOP code-reuse attack and other generic exploit behavior.

JOP Classifier: Jump-Oriented Programming (JOP) presents a unique code-reuse attack [39]. Instead of using gadgets ending in return instructions, JOP uses register-indirect jumps to chain together gadgets. JOP's design contains two types of gadgets: functional gadgets and dispatcher gadgets. Dispatcher gadgets essentially maintain a virtual program counter, advancing the attack and allowing functional gadgets to execute. A dispatcher gadget may prove as simple as `ADD EDX, 4; JMP [EDX]`, which repeatedly advances the virtual program counter by a constant value. Since dispatcher gadgets must alternate functional gadgets, we implement a classifier that identifies JOP similar to parameter suspicion. To detect JOP, Code-Stop identifies the alternating dispatcher gadgets by manipulation of a single register, and then a jump instruction. As with parameter suspicion, there is a negligible likelihood that a random string contains alternating addresses that point to instructions that happen to manipulate a single register and then jump to that register.



```
0x76BA9090
^0x3C900063
^0x3C5200B6^

90 90 BA 76
35 63 00 90 3C
35 B6 00 52 3C 35

90            NOP
90            NOP
BA 76356300   MOV, EDX 00633576
90            NOP
3C 35         CMP AL, 0x35
B6 00         MOV DH,0
52            PUSH EDX
3C 35         CMP AL, 0x35
```

Figure 6. Identifying just-in-time code spraying

JIT Code-Spraying Classifier: We now discuss how to extend Code-Stop to detect other types of exploitation attack vectors, including JIT-Flash. JIT-Flash sprays executable code directly into memory. Consider the example depicted in Figure 6. A JIT-Flash attack writes a suspected string into memory. When Code-Stop translates the string to raw bytes, it replaces the XOR character with the ASCII encoding value 0x35. Next, we evaluate the bytes as variable-length x86 instructions. Emulating the instructions determines that the attack moves the value 0x00633576 into EDX and subsequently pushed that value onto the stack. To detect JIT Code-Spraying, we extend parameter suspicion to examine potential code and determine if the value has intentionally been placed onto the stack for malicious purposes. It is very unlikely that a benign arrangement of bytes would accomplish the same effect.

To identify JIT Code Spraying, Code-Stop examines potential JIT bytes to determine if they: 1) pop an address, 2) push an address to the flow of execution, or 3) store values at our heap-spray. In the case of Figure 6, the highlighted section of code clearly pushes an address to the flow of execution and is detected as an attack. Next, we describe our evaluation.

VI. Evaluation

We evaluate Code-Stop by answering the following research questions.

- **RQ1:** What is the accuracy of detecting gadgets?

- **RQ2:** What is the performance overhead on the client host?
- **RQ3:** What is the scalability of the network proxy?
- **RQ4:** What set of attacks can Code-Stop detect?

The following sections answers these questions and describes the configuration of our prototype used in the evaluation.

A. Experimental Setup

We tested our prototype using the following systems, which were configured as described below:

Network Proxy: *DansGuardian 2.10.1.1, Squid Version 3.3.8 on Ubuntu 14.04 LTS.* We implement our gadget parser as a DansGuardian content-scanner via a python script that parses suspect gadgets. The script communicates with the vulnerable host over TCP sockets to understand the effect of the emulation of the suspected gadgets.

Vulnerable Host: *Windows 7 Service Pack 1.* We installed applications that are known to be vulnerable, including Adobe Reader 9.0, Internet Explorer 8.0 and 11.0, JRE-1.6, and Microsoft Office 2010. It is necessary to test using these specific application versions to ensure we can properly test ROP chains from `icucnv36.dll`, `msvcr71.dll`, and `hxds.dll`. Additionally, we installed a third-party browser help object IBM Forms Viewer 4.0.0, which installed the `pehelper.dll` that is compiled without ASLR support. The host runs a Python script that determines the emulated impact of potential gadgets and communicates with the network proxy.

B. RQ1: Accuracy of Detecting Gadgets

The first part of our evaluation investigates the accuracy of our prototype to detect gadgets in PDF documents. We compare three cases: (a) matching a string that contains a hexadecimal address; (b) matching a string that contains a hexadecimal address corresponding to the address space of Adobe Acrobat Reader and its shared libraries; and (c) matching using parameter suspicion. In doing so, we show the benefit of Code-Stop over naive approaches.

Datatasets: We utilize the following datasets to illustrate the accuracy of our prototype.

- Contagio Datasets: The Contagio Benign Dataset consists of 9,000 known benign PDF documents from March 2013. The Contagio Malware Repository Team collected the dataset and published it for the specific purpose of testing security products for false positives. We used the dataset to ensure our prototype did not falsely detect benign documents as malicious in nature. In addition, the Contagio Team published a smaller dataset of 109 complex PDF documents that contained shockwave flash. This dataset was included for the specific testing of larger PDF files that contained large amounts of arbitrary data.
- VirusTotal Dataset: The VirusTotal Dataset includes 1,000 known benign documents from September 2015. The VirusTotal team made their private API available for testing our prototype. Using their private API, we queried for known benign documents that had been uploaded to their website within the last thirty days.
- Metasploit Dataset: We extended the Metasploit_adobe_toolbutton.rb exploit to include five additional ROP Chains for shared libraries without ASLR. We then randomly generated 550 unique malicious PDF documents with different payloads and different ROP Chains. Note that this is the only controlled dataset in our experiment evaluation as all PDFs contain known ROP chains. Using a controlled dataset, we can test Code-Stop's ability to detect code-reuse attacks since the application and operating system versions must match that of the attack. Alternatively, a wild dataset would encompass attacks against multiple versions of applications and operating systems.

Results: Table III depicts the results of executing the string matching algorithms on our dataset. The first row demonstrates that simply matching strings containing hexadecimal addresses produces a significant number of false positives. Refining the matching to values that are valid address ranges significantly reduces the number of false positives, but there are still some. However, using parameter suspicion's heuristic of identifying three register changes; we do not detect false positives.

C. RQ2: Host Performance Overhead

Next, we measured the performance overhead on the client to determine any negative impact when rendering content. We focused on JavaScript performance, since Code-Stop heavily parses and examines JavaScript for potential code-reuse attacks. We measured JavaScript performance using the Sun Spider JavaScript Benchmarking Suite [40]. Sun Spider measures the performance of the host executing core JavaScript language, focusing on the typical code implemented in real-world situations. We compared the Sun Spider results for the following scenarios: (a) a normal host (baseline), (b) a host using a Squid Proxy, (c) a host using the SquidClamAV Proxy with Avast anti-virus scanning, and (d) our Code-Stop solution. Sun Spider reports 95% confidence intervals as percentages.

Results: Table IV shows that Code-Stop suffers a minimal performance overhead penalty, comparable to the results of the SquidClamAV Proxy. Ultimately, the performance overhead is negligible, since the difference between the baseline and Code-Stop is less than 8ms, which is imperceptible to the end user.

D. RQ3: Network Proxy Scalability

In the previous experiment, we investigated the performance overhead on a client protected by Code-Stop. However, in a typical deployment, there will be many clients protected by the Code-Stop proxy. Therefore, it is important to characterize the scalability of Code-Stop to many clients. The experiment measured the average time to download a 10MB PDF file through the same four scenarios considered in Section VI-C.

We measured the average time to initiate and complete a download of the PDF given 1,5,10,15,20 and 25 concurrent users repeatedly downloading an uncached copy of the PDF over a period of 5 minutes. During the Code-Stop test, this resulted in the sum users downloading the 10MB file over 7,500 times over a period of 300 seconds.

Results: Figure 7 demonstrates that Code-Stop scales similar to a typical proxy configuration and an anti-virus solution. In an environment with 25 concurrent users, the time to download a 10MB document averaged 0.978±0.129 seconds. In contrast, hosts under Code-Stop protection averaged 1.968±0.262

TABLE III. CODE-STOP PARAMETER SUSPICION DETECTION IN BENIGN AND MALICIOUS DATASETS

	Total Files Tested	Total String Matches	Files with String Matches	Total Strings in Address Space	Files with Strings in Address Space	Files Matched by Parameter Suspicion
Contagio Benign Dataset	9,000	3,249,338	8,977	5,286	494	0
Contagio Complex Dataset	109	8,853	104	35	24	0
VirusTotal Benign Dataset	1,001	3,096,287	987	3,309	191	0
Metasploit Malicious Dataset	550	63,170	550	8,651	550	550

TABLE IV. RESULTS OF SUNSPIDER PERFORMANCE TEST

	Without Proxy	Standard Proxy	SquidClamAV Proxy	Code-Stop Proxy
Time (ms)	105.9ms ± 1.2%	106.5ms ± 1.2%	112.3ms ± 3.7%	113.8ms ± 2.8%

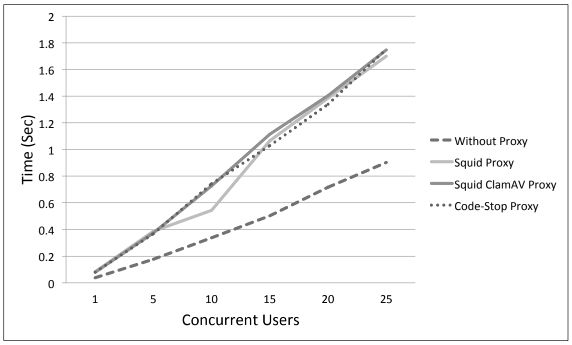


Figure 7. Performance impact with concurrent users

seconds. Both an anti-virus proxy and our prototype roughly double the time to complete the request to download a document. We argue that our solution proves superior to anti-virus scanning since the anti-virus scanning done on the proxy cannot take into account the full context of the host.

E. RQ4: Coverage

Finally, we evaluated the coverage of our Code-Stop prototype. Coverage defines the set of attacks our system can detect under ideal conditions. We evaluated parameter suspicion alone, as well as its use in combination with disarmed reading to detect code-reuse attacks that rely upon loading shared libraries without protection. Further, we measured Code-Stop against a set of five unique shared libraries that lack ASLR. We argue that Code-Stop is not limited to the method, the library, or the individual gadgets used in the code-reuse attack but is a technique that is applicable to a broader set of attacks.

Dataset: We evaluated how parameter suspicion detected ROP chains from different shared libraries. Specifically, we sought libraries that had one or more of the following characters: (1) compiled without ASLR (2) used with multiple different applications (3) contains the necessary instructions to generate an ROP chain, and (4) has been seen in use in the wild in attacks. Thus we selected the following libraries `icucnv36.dll`, `cryptocme2.dll`, `grooveutil.dll`, `pehelper.dll`, and `msvcr71.dll` respectively. Section VI-B previously demonstrated that our prototype detected malicious PDFs with ROP chains from the first three libraries with a malicious PDF. To further demonstrate the coverage, we modified the script for the Metasploit ie_cgenericelement_uaf.rb exploit for Internet Explorer to include new ROP chains from `pehelper.dll`, `msvcr71.dll`, and `grooveutil.dll`. In doing this, we demonstrate that Code-Stop is applicable to a broader set of applications since the technique detects the general behavior of code-reuse attacks and is not unique to any one application or shared library.

Results: Code-Stop detected the gadgets from the shared libraries of `pehelper.dll`, `msvcr71.dll`, and `grooveutil.dll`. Since these libraries are from a third-party application, the operating system, and a separate Microsoft application we can argue that the results indicate that parameter-suspicion covers the broad spectrum of shared libraries used to implement code-reuse attacks.

VII. Limitations and Future Work

Limitations of 32-Bit Architecture: For simplicity, we implemented the Code-Stop prototype against the 32-bit architecture. Extending Code-Stop to 64-bit increases the accuracy of the system, by decreasing the probability that a random string would refer to an address in the 64-bit address space.

Limitations of De-obfuscation: Additionally, we do not address obfuscation in the design of Code-Stop. Other works have examined the de-obfuscation of malicious JavaScript, PDF document streams, or Adobe Flash ActionScript [32] [41]. Implemented as a proxy, Code-Stop can prevent access to files it is unable to de-obfuscate. Future work may examine the concept of a quarantined machine that can emulate the full effect the result of rendering the obfuscated file and determine if the file attempts to allocate potential gadgets into memory.

Extending to Other Operating Systems: Further, we implemented the Code-Stop host software only as a Microsoft Windows application. Extending the host software to other platforms allows the possibility to protect against platform-unique attacks, such as S-ROP. In this paper, we addressed classifiers for generic code-reuse attacks and specific cases for ROP and JOP. Future work may extend classifiers to identify other attack vectors outside of the scope of code-reuse attacks as demonstrated with our JIT Code-Spraying classifier.

Allowing the Host to Proxy Content: Last, we implemented the gadget parser on the proxy and rely on the host to deliver context. Both could be implemented on the host. Splitting the design allows both the network and the host to share the performance overhead and prevents the attacker from opting out of both defense mechanisms. Future work may examine the host both parsing gadgets and examining context.

VIII. Related Work

In the following section, we describe the related work that addresses the shortcomings of defense strategies against code-reuse attacks. First, we examine the recent work into mitigating code-reuse using compiler- and operating system- based mechanisms. Next, we examine defenses utilizing the network layer. We believe our solution can combine the benefits of both the host and network layer defenses to mitigate code-reuse attacks.

A. Host-Level Code-Reuse Prevention

Because of ROP's success in bypassing DEP, several papers have examined means for defending against ROP [4] [42] [43]. Onarlioglu et al. [5] presented G-Free to counter ROP gadgets by removing unaligned gadgets altogether, and by removing a portion of aligned gadgets from binaries at compile-time. Alignment checking proved to be one of the more trivial but successful mechanisms for defeating ROP gadgets. Gadgets can consist of different offsets inside valid instructions. G-Free extends GCC to ensure a gadget-free binary. However, G-Free fails to protect already compiled binaries or shared libraries.

Other mechanisms such as kBouncer, ROPGuard, and ROPecker use the processor's Last Branch Recording (LBR) functionality to heuristically examine control flow for gadgets [25]. kBouncer [26] examines the LBR for fifty-two WinAPI functions considered harmful. RopGuard [7] expands upon this by performing past and future control flow analysis and static checks. By preceding protected API calls, it ensures that an attacker cannot divert into a protected function. Furthermore, RopGuard performs checks to ensure a process does not attempt to make the stack executable by disabling DEP. ROPecker [8] attempts to combine the benefits of RopGuard and kBouncer by examining the LBR control-flow simulation. However, the LBR defenses have proven trivial to bypass by clearing the LBR. Schustere et al. [25] demonstrated that both i-long-jumps and LBR flushing gadgets can defeat the effectiveness of any mitigation strategy that relies on the LBR.

Several recent defenses have prevented memory disclosures that could be used to create gadgets just-in-time (JIT). HideM [44] uses a split translation look-aside buffer to fetch read and executable memory separately. However, Crane et al. [45] noted that the split TLB technique does not work on recent x86 processors, since most processors released after 2008 contain a unified second level TLB. In contrast, Crane et al. [45] presented Redactor that successfully disassembles code pages and identifies JIT-ROP gadgets dynamically at runtime. Additionally, De Groef et al. [9] presented a countermeasure for JIT-ROP gadgets. When faced with the difficult problem that JIT gadgets can bypass ASLR and W$\oplus$X policies, De Groef and his team implemented a run-time monitor to prevent JIT ROP. The monitor uses a series of checks against any system call generated from the stack or heap, in order to determine the original calling function. However, DeMott [3] demonstrated bypassing caller check monitors by borrowing from valid code that makes calls to protected APIs.

B. Network-Level Exploit Prevention

Several checks can be performed at the network-layer to prevent the successful exploitation of hosts. Early, signature-based checks identified specific x86 instructions commonly used in malicious shellcode. However, the vast abundance of code-obfuscation techniques made these early checks easy for an attacker to bypass. One of the earlier methods for detection improved on such techniques by using a NIDS-embedded CPU emulator [27]. This emulator executed potential instructions with the intent of identifying polymorphic shellcode that evaded signature-based detection. In a similar approach, SigFree [28] presented a model for implementing a proxy-based firewall. This firewall successfully identified and filtered client-side exploits by detecting code and examining instructions using a process of code abstraction. With the extensive amount of client-side code executing in the context of the modern browser, the 2006 results appear to be only applicable in theory.

In 2010, researchers developed the JSAND toolkit [29] to emulate JavaScript and reliably identify malicious code based on machine-learning. However, this approach is limited to JavaScript and does not affect the large volume of exploits delivered via modern plug-ins, such as Adobe Flash. Support Vector Machines (SVMs) shellcode detection egingines use modern emulation to identify key instructions commonly used in shellcode [46]. Polychronakis and Keromytis [24] specifically proposed a network tool that could identify potential ROP Gadgets, but did so without dynamic context from the host. We argue that the shortcomings of the host-layer and network-layer defenses can be addressed by using the network to defend with the context of the host.

IX. Conclusion

This paper presented Code-Stop, a network and host-based cooperative system for defending against client-side attacks that bypass exploit mitigation strategies using code-reuse. We introduced the concept of *parameter suspicion* classifier to identify code-reuse attacks. With parameter suspicion, we introduced the idea of identifying when an attacker attempts to call a Windows API function with specific parameters to allocate memory or change memory protections. Rather than identifying the specific call to a particular function, parameter suspicion identifies code-reuse attacks by emulating the behavior of suspected instructions, gained from the context of the destination host. Parameter suspicion identifies when instructions place parameters onto the stack as part of a code-reuse attack to call a Windows API function to allocate or change memory. By examining the host context, Code-Stop can effectively prevent client-side attacks from reaching the intended victim. We implemented a prototype of Code-Stop, and verified the ability to mitigate exploits against Internet Explorer, Adobe Reader, and Microsoft Office applications. Our evaluation showed that Code-Stop successfully prevented code-reuse attacks without risk of false-positives. With the ability to detect code-reuse attacks and mitigate against previously unseen attack vectors, Code-Stop is a practical solution for enhancing the security of client-side applications.

References

[1] Microsoft Corporation, "BlueHat Prize Winners Announced," Aug 2012, Retrieved: April 10, 2016. [Online]. Available: http://www.microsoft.com/security/bluehatprize/

[2] H. Shacham et al., "On the Effectiveness of Address-Space Randomization," in Proc. of the ACM conference on Computer and communications security, Oct 2004, pp. 298–307.

[3] J. DeMott, "Bypassing EMET 4.1," Bromium Labs, Feb 2014, Retrieved: April 10, 2016. [Online]. Available: http://labs.bromium.com/2014/02/24/bypassing-emet-4-1/

[4] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Smashing the Gadgets: Hindering Return-Oriented Programming Using In-Place Code Randomization," in Proc. of the IEEE Symposium on Security and Privacy, 2012, pp. 601–615.

[5] K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda, "G-Free: Defeating Return-oriented Programming Through Gadget-less Binaries," in Proc. of the Annual Computer Security Applications Conference (ACSAC), 2010, pp. 49–58.

[6] V. Pappas, M. Polychronakis, and A. D. Keromytis, “Transparent ROP Exploit Mitigation Using Indirect Branch Tracing,” in Proc. of the USENIX Conference on Security, 2013, pp. 447–462.

[7] I. Fratric, “Runtime Prevention of Return-Oriented Programming Attacks,” in Microsoft’s BlueHat Prize - Black Hat USA 2012. June, 2012. [Online]. Available: https://github.com/ivanfratric/ropguard

[8] Y. Cheng, Z. Zhou, M. Yu, X. Ding, and R. H. Deng, “ROPecker: A generic and practical approach for defending against ROP attacks,” in Proc. of the Symposium on Network and Distributed System Security (NDSS), 2014, pp. 763–780.

[9] W. De Groef, N. Nikiforakis, Y. Younan, and F. Piessens, “Jitsec: Just-in-time security for code injection attacks,” in Benelux Workshop on Information and System Security (WISSEC), Nov 2010, pp. 1–15.

[10] J. C. Chen and B. Li, “Evolution of exploit kits : Exploring past trends and current improvements,” Trend Micro, Tech. Rep., 2015.

[11] S. Bhatkar, D. C. DuVarney, and R. Sekar, “Address Obfuscation: An Efficient Approach to Combat a Broad Range of Memory Error Exploits,” in Proc. of the 12th USENIX security symposium, vol. 120. Washington, DC., Aug 2003, pp. 105–120.

[12] PaX Team, “Pax address space layout randomization (aslr),” Mar 2003, Retrieved: April 10, 2016. [Online]. Available: http://pax.grsecurity.net/docs/aslr.txt

[13] C. Kil, J. Jun, C. Bookholt, J. Xu, and P. Ning, “Address Space Layout Permutation (ASLP): Towards Fine-Grained Randomization of Commodity Software,” in Proc. of the Annual Computer Security Applications Conference (ACSAC), 2006, pp. 339–348.

[14] A. Sotirov and M. Dowd, “Bypassing browser memory protections in windows vista,” in Blackhat USA, Aug 2008.

[15] T. Haq, “Internet explorer 8 exploit found in watering hole campaign targeting chinese dissidents,” Mar 2013, Retrieved: April 10, 2016. [Online]. Available: https://www.fireeye.com/blog/threat-research/2013/03/internet-explorer-8-exploit-found-in-watering-hole-campaign-targeting-chinese-dissidents.html

[16] T. OConnor, “Nuclear Scientists, Pandas and EMET Keeping Me Honest,” SANS Internet Storm Center, Tech. Rep., May 2013, Retrieved: April 10, 2016.

[17] D. Litchfield, “Buffer Underruns, DEP, ASLR and improving the Exploitation Prevention Mechanisms (XPMs) on the Windows platform,” in Next Generation Security Software, Sep 2005.

[18] E. J. Schwartz, T. Avgerinos, and D. Brumley, “Q: Exploit Hardening Made Easy,” in Proc. of the USENIX Security Symposium, Aug 2011, pp. 25–41.

[19] P. Team, “Pax non-executable pages design & implementation,” May 2003, Retrieved: April 10, 2016. [Online]. Available: http://pax.grsecurity.net/docs/noexec.txt

[20] H. Shacham, “The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86),” in Proc. of the ACM conference on Computer and communications security, Oct 2007, pp. 552–561.

[21] D. Dai Zovi, “Practical return-oriented programming,” 2010, Retrieved: April 10, 2016. [Online]. Available: https://www.trailofbits.com/resources/practical_rop_slides.pdf

[22] P. Anwar, “Bypassing Microsoft Windows ASLR with a little help by MS-Help,” Aug 2012, Retrieved: April 10, 2016. [Online]. Available: https://www.greyhathacker.net/?p=585

[23] N. Sikka, “Cve 2013 3893: Fix it workaround available,” Microsoft Corporation, Tech. Rep., Sep 2013, Retrieved: April 10, 2016. [Online]. Available: http://blogs.technet.com/b/srd/archive/2013/09/17/cve-2013-3893-fix-it-workaround-available.aspx

[24] M. Polychronakis and A. D. Keromytis, “ROP Payload Detection Using Speculative Code Execution,” in Proc. of the International Conference on Malicious and Unwanted Software (MALWARE), 2011, pp. 58–65.

[25] F. Schuster et al., “Evaluating the Effectiveness of Current Anti-ROP Defenses,” in Research in Attacks, Intrusions and Defenses, vol. 8688, 2014, pp. 88–108.

[26] L. Davi, P. Koeberl, and A.-R. Sadeghi, “Hardware-Assisted Fine-Grained Control-Flow Integrity: Towards Efficient Protection of Embedded Systems Against Software Exploitation,” in Proc. of the Annual Design Automation Conference, 2014, pp. 1–6.

[27] M. Polychronakis, K. G. Anagnostakis, and E. P. Markatos, “Network-Level Polymorphic Shellcode Detection Using Emulation,” in Proc. of Detection of Intrusions and Malware & Vulnerability Assessment. Springer, Jul 2006, pp. 21:1–21:3.

[28] X. Wang, C.-C. Pan, P. Liu, and S. Zhu, “Sigfree: A Signature-Free Buffer Overflow Attack Blocker,” IEEE Transactions on Dependable and Secure Computing, vol. 7, Jun 2010.

[29] M. Cova, C. Kruegel, and G. Vigna, “Detection and Analysis of Drive-by-download Attacks and Malicious JavaScript Code,” in Proc. of the International Conference on World Wide Web (WWW), 2010.

[30] Z. Tzermias, G. Sykiotakis, M. Polychronakis, and E. P. Markatos, “Combining static and dynamic analysis for the detection of malicious documents,” in Proc. of the Fourth European Workshop on System Security. ACM, 2011, p. 4.

[31] D. Stevens, “Malicious pdf documents explained,” Security & Privacy, IEEE, vol. 9, no. 1, 2011.

[32] B. Hartstein, “Jsunpack: An automatic javascript unpacker,” in ShmooCon convention, 2009.

[33] N. Šrndic and P. Laskov, “Detection of malicious pdf files based on hierarchical document structure,” in Proc. of the 20th Annual Network & Distributed System Security Symposium. Citeseer, 2013.

[34] M. Egele, P. Wurzinger, C. Kruegel, and E. Kirda, “Defending browsers against drive-by downloads: Mitigating heap-spraying code injection attacks,” in Detection of Intrusions and Malware, and Vulnerability Assessment. Springer, 2009, pp. 88–106.

[35] Y. Yu, “Rops are for the 99 percent,” Mar 2014, Retrieved: April 10, 2016. [Online]. Available: https://cansecwest.com/slides/2014/ROPs_are_for_the_99_CanSecWest_2014.pdf

[36] M. S. Xiaobo Chen, Dan Caselden, “New zero-day exploit targeting internet explorer versions 9 through 11 identified in targeted attacks,” Apr 2014, Retrieved: April 10, 2016. [Online]. Available: https://www.fireeye.com/blog/threat-research/2014/04/new-zero-day-exploit-targeting-internet-explorer-versions-9-through-11-identified-in-targeted-attacks.html

[37] P. V. Eeckhoutte, “Corelan team exploit development swiss army knife,” 2015, Retrieved: April 10, 2016. [Online]. Available: https://github.com/corelan/mona/blob/master/mona.py

[38] J. V. Soroush Dalili, sinn3r, “Adobe reader toolbutton use after free,” Nov 2013, Retrieved: April 10, 2016. [Online]. Available: http://www.rapid7.com/db/modules/exploit/windows/browser/adobe_toolbutton

[39] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, “Jump-oriented Programming: A New Class of Code-reuse Attack,” in Proc. of the ACM Symposium on Information, Computer and Communications Security (ASIACCS), 2011, pp. 30–40.

[40] J. K. Martinsen, H. Grahn, and A. Isberg, “A Comparative Evaluation of JavaScript Execution Behavior,” in Proc. of the International Conference on Web Engineering, 2011, pp. 399–402.

[41] J. M. Esparza, “Obfuscation and (Non-) Detection of Malicious PDF Files,” in CARO 2011 Workshop in Prague, Czech Republic, 2011.

[42] L. Davi, A.-R. Sadeghi, D. Lehmann, and F. Monrose, “Stitching the Gadgets: On the Ineffectiveness of Coarse-grained Control-flow Integrity Protection,” in Proc. of the USENIX Conference on Security Symposium, 2014, pp. 01–416.

[43] L. Davi, A.-R. Sadeghi, and M. Winandy, “ROPdefender: A Detection Tool to Defend Against Return-oriented Programming Attacks,” in Proc. of the ACM Symposium on Information, Computer and Communications Security, 2011, pp. 40–51.

[44] J. Gionta, W. Enck, and P. Ning, “Hidem: Protecting the contents of userspace memory in the face of disclosure vulnerabilities,” in Proc. of the ACM Conference on Data and Application Security and Privacy (CODASPY), 2015, pp. 325–336.

[45] S. Crane and et al., “Readactor: Practical Code Randomization Resilient to Memory Disclosure,” in Proc. of the IEEE Symposium on Security and Privacy, 2015, pp. 763–780.

[46] Y. Hou, J. W. Zhuge, D. Xin, and W. Feng, “SBE : A Precise Shellcode Detection Engine Based on Emulation and Support Vector Machine,” in Proc. of the International Conf. on Information Security Practice and Experience, 2014, pp. 159–171.