Check my profile: Leveraging static analysis for fast and accurate detection of ROP gadgets

Blaine Stancill[1], Kevin Z. Snow[1], Nathan Otterness[1], Fabian Monrose[1], Lucas Davi[2], and Ahmad-Reza Sadeghi[2]

[1] Department of Computer Science, University of North Carolina at Chapel Hill,
[2] CASED/Technische Universität Darmstadt, Germany,
email:{stancill,kzsnow,otternes,fabian}@cs.unc.edu,
{lucas.davi,ahmad.sadeghi}@trust.cased.de

Abstract. Return-oriented programming (ROP) offers a powerful technique for undermining state-of-the-art security mechanisms, including non-executable memory and address space layout randomization. To mitigate this daunting attack strategy, several in-built defensive mechanisms have been proposed. In this work, we instead focus on detection techniques that do not require *any* modification to end-user platforms. Specifically, we propose a novel framework that efficiently analyzes documents (PDF, Office, or HTML files) and detects whether they contain a return-oriented programming payload. To do so, we provide advanced techniques for taking memory snapshots of a target application, efficiently transferring the snapshots to a host system, as well as novel static analysis and filtering techniques to identify and profile chains of code pointers referencing ROP gadgets (that may even reside in randomized libraries). Our evaluation of over 7,662 benign and 57 malicious documents demonstrate that we can perform such analysis accurately and expeditiously — with the vast majority of documents analyzed in about 3 seconds.

Keywords: return-oriented programming, malware analysis

1 Introduction

Today, the wide-spread proliferation of document-based exploits distributed via massive web and email-based attack campaigns is an all too familiar event. Largely, the immediate goal of these attacks is to compromise target systems by executing arbitrary malicious code in the context of the exploited program. Loosely speaking, 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. Code injection attacks date as far back as the Morris Worm [42] and were later popularized by the seminal work of Aleph One [3] on stack vulnerabilities. However, with the introduction and wide-spread deployment of the non-executable memory principle [29] (DEP), conventional code injection attacks have been rendered ineffective by ensuring the memory that code is injected into is no longer directly executable.

However, as defenses were fortified with DEP, attackers began to adapt by perfecting the art of creating practical code reuse attacks. In a so-called return-into-libc attack, for example, rather than redirect execution flow to injected code, the adversary simply redirects flow to a critical library function such as WinExec(). However, while return-into-libc attacks have been shown to be powerful enough to enable chained function calls [32], these attacks suffer from a severe restriction compared to conventional code injection attacks: that is, they do not enable *arbitrary* code execution. Instead, the adversary is dependent on library functions, and can only call one function after the other. That shortcoming, however, was later shown to be easily addressed. In particular, Shacham [38] introduced return-oriented programming (ROP), wherein short sequences of instructions are used to induce arbitrary program behavior.

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 [39] or memory disclosures [37, 45]. Sadly, even more advanced fine-grained ASLR schemes [19, 22, 35, 46] have also been rendered ineffective in the face of just-in-time return-oriented programming attacks where instructions needed to create the payload are dynamically assembled at runtime [41]. Therefore, it is our belief that until more comprehensive preventive mechanisms for code injection and reuse attacks take hold, techniques for *detecting* code reuse attacks remain of utmost importance [43].

In this paper, we provide 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, we focus on *detection* (as a service) rather than in-built *prevention* on end-user systems. In doing so, we fill an important gap in recent proposals for defenses against code reuse attacks. More specifically, preventive defenses have yet to be widely deployed, mostly due to performance and stability concerns, while the detection approach we describe 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 our goals, we 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).

2 Background and Challenges

The basic idea of return-oriented programming is depicted in Figure 1. In the first step, the adversary places the ROP payload into the program's writable area. In this case, the payload does not contain any executable code, but rather, contains a series of pointers (e.g., return addresses). Each return address points to a particular instruction sequence residing in the address space of the target program (e.g., a library segment). Typically, the instruction sequences consist of a handful of assembler instructions that terminate in a return instruction (`RET`). It is exactly the return instruction that gives return-oriented programming its name, as it serves as the mechanism for connecting all the sequences. In ROP parlance, a set of instruction sequences is called a *gadget*, where each element of the set is an atomic task (e.g., a load, add, or invocation of a system call). Shacham [38] showed that common libraries (such as libc) provide enough sequences to construct a Turing-complete gadget set, thereby allowing an adversary to perform arbitrary operations.

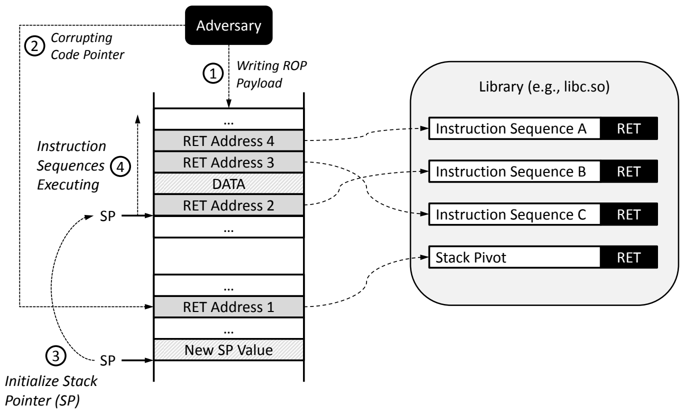

Adversary
② Corrupting Code Pointer
① Writing ROP Payload
Library (e.g., libc.so)
Instruction Sequences Executing ④
...
RET Address 4
RET Address 3
DATA
RET Address 2
SP
RET Address 1
New SP Value
③ Initialize Stack Pointer (SP)
Instruction Sequence A RET
Instruction Sequence B RET
Instruction Sequence C RET
Stack Pivot RET

Fig. 1: Basic Principle of Return-Oriented Programming

From a practical point of view, all the adversary needs to do in order to derive her gadget set is to statically analyze the target program and the shared libraries it links to. This step can be easily automated with the original Galileo algorithm [38], or performed using freely available exploit tools[3]. Once a vulnerable entry point is discovered, the adversary constructs the malicious payload by carefully combining the found gadgets in a manner that subverts the target program's intended execution flow. Typically, this is achieved by exploiting the vulnerable entry point (*e.g.,* the buffer overflow) to manipulate a code pointer (Step ②). For example, in Figure 1, the code pointer is overwritten with `RET` Address 1 which points to a special sequence, the stack pivot [48]. This sequence — identified during static analysis — is required to correctly set-up the return-oriented programming attack. Specifically, upon invocation (Step ③), the stack pivot sequence adjusts the program's stack pointer to point to the beginning of the return-oriented programming payload. A typical stack pivot sequence might look like `POP EAX; XCHG ESP,EAX; RET`. Afterwards, the return-oriented payload gets executed (Step ④), starting with Instruction Sequence B (pointed to

[3] See, for example, the `mona` (`http://redmine.corelan.be/projects/mona`) or `ropc` (`http://github.com/pakt/ropc`) tools.

by Return Address 2). The return instruction of Instruction Sequence B ensures that the next return address is loaded from the stack to invoke Instruction Sequence C. This procedure can be repeated as many times as the adversary desires. The DATA tag in Figure 1 simply highlights the fact that instruction sequences can also process attacker-supplied data, such as arbitrary offsets or pointers to strings (*e.g.,* a pointer to /bin/sh).

Lastly, one might argue that since return instructions play a pivotal role in these attacks, a natural defense is simply to monitor and protect return instructions to mitigate return-oriented programming, *e.g.,* by deploying a shadow stack to validate whether a return transfers the execution back to the original caller [1, 13, 16]. Even so, return-oriented programming without returns is possible where the adversary only needs to search for instruction sequences that terminate in an indirect jump instruction [4]. Indeed, Checkoway et al. [7] recently demonstrated that a Turing-complete gadget set can be derived for this advanced code reuse attack technique. To date, return-oriented programming has been adapted to numerous platforms (*e.g.,* SPARC [5], Atmel AVR [15], ARM [25]), and several real-world exploits (e.g., against Adobe reader [20], iOS Safari [17], and Internet Explorer [45]) have been found that leverage this ingenious attack technique. Hence, ROP still offers a formidable code reuse strategy.

Peculiarities of Real-World Code Reuse Attacks: In the course of applying our approach to a large data set on real-world exploits (see §4), we uncovered several peculiarities of modern code reuse attacks. To our surprise, several exploits include stack push operations that partly overwrite the ROP payload with new pointers to instruction sequences at runtime. Although return-oriented programming attacks typically overwrite already used pointers with local variables when invoking a function, the peculiarity we discovered is that some exploits overwrite parts of the payload with a new payload and adjust the stack pointer accordingly. As far as we are aware, this challenge has not been documented elsewhere, and makes detection based on analyzing memory snapshots particularly difficult — since the detection mechanism has to foresee that a new payload is loaded onto the stack after the original payload has been injected.

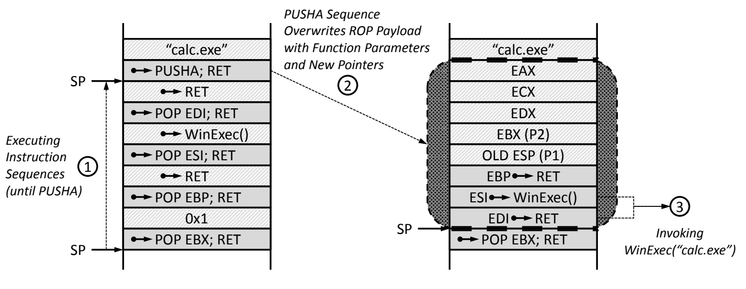


Fig. 2: Peculiarities of real-world return-oriented programming attack.

For pedagogical reasons, Figure 2 illustrates this particular challenge. In this case, the attacker's goal is to execute the function *WinExec("calc.exe")* by means of return-oriented programming. 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).

We return to this example later in §3.1, and demonstrate how our approach is able to detect this, and other, dynamic behavior of real-world attacks.

3 Our Approach

The design and engineering of a system for detecting and analyzing code reuse attacks embedded in various file formats posed significant challenges, not the least of which is the context-sensitivity of recent code reuse attacks. That is, today's exploit payloads are often built dynamically (e.g., via application-supported scripting) as the file is opened and leverage data from the memory footprint of the particular instance of the application process that renders the document[4]. 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. Obviously, for practical reasons, the end-to-end analysis of each file must complete as quickly as possible.

The approach we took to achieve these goals is highlighted in Figure 3. 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. We envision a use-case for these steps wherein documents are either extracted from an email gateway, parsed from network flows, harvested from web pages, or manually submitted to our system for analysis. In what follows, we discuss the challenges and solutions we provide for each step of our system.

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

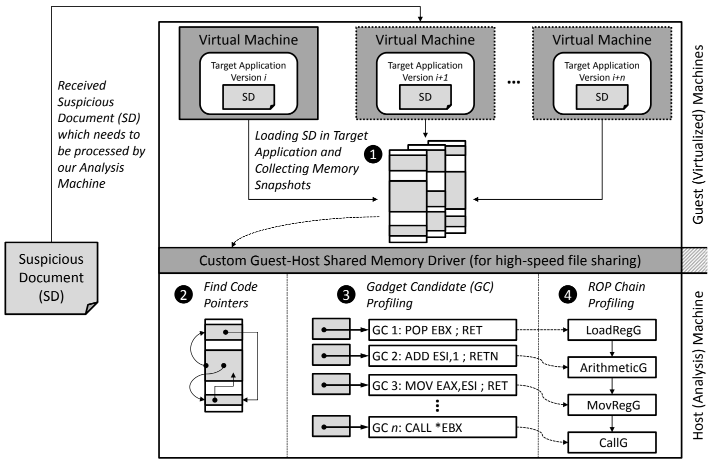

Received Suspicious Document (SD) which needs to be processed by our Analysis Machine
Virtual Machine
Target Application Version i
SD
Virtual Machine
Target Application Version i+1
SD
...
Virtual Machine
Target Application Version i+n
SD
Guest (Virtualized) Machines
Loading SD in Target Application and Collecting Memory Snapshots
1
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
Host (Analysis) Machine
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

Fig. 3: High-level abstraction of our detection approach

3.1 Step ❶: Fast Application Snapshots

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 hardcoding 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 [37, 45] or dynamically compile a payload on-the-fly [41]. 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., [10, 14, 44]). 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 [34].

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, we adapt the complementary approach of Snow et al. [40], wherein an in-guest application uses the `dbghelp` library to generate a rich application snapshot, called a `minidump`[5]. 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*.

We generate a snapshot once the `cpu` goes idle, or a time or memory threshold is exceeded. As with any snapshot-based approach, we rely 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 2). 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 all real-world exploits we have observed due to either heap spraying the payload, or pass-by-value function parameters.

Similarly to Lindorfer et al. [27], we simultaneously launch the document in different versions of the target appplication. While doing so may seem like a heavyweight operation, we 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. [24].

A significant bottleneck of the in-guest snapshot approach in past work was 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 paravirtualization) can add tens of seconds of overhead. To solve the problem of the fast transfer of memory snapshots, we developed a custom guest-host shared memory driver 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 our 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, our memory snapshot utility and fast transfer suite implementation is about 4,600 lines of `C/C++` code, and our virtual machine manager is about 2,200 lines of `python` code that fully automates document analysis. Thus, we use our fast-transfer driver to pull the application snapshot out of the guest, and onto the host system for further analysis.

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

3.2 Step ❷: Efficient Scanning of Memory Snapshots

With a memory snapshot of the target application (with document loaded) in-hand, we now scan the snapshot to identify content characteristic of `ROP`. To do so, we first traverse 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, we 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 was dynamically allocated, but excludes the static variable regions and relocation data used by each module[6]. The application snapshots from step ❷ provide all the necessary meta-information about memory regions. In short, executable memory is considered gadget space, while writeable memory is considered payload space. Note that memory that is both writeable and executable is considered in both spaces.

As we traverse the payload space, we 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. We do so because the initial alignment of a payload is unknown. For each 4-byte window, we 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 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, we take advantage of the fact that memory is partitioned into at least 4KB pages. We 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 us 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*), we treat it as the start of a potential gadget chain and start by profiling the behavior of the first gadget candidate in the chain.

3.3 Step ❸: 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, we label any instruction sequence

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

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 (hardcoded) 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[7], or (vi) the end of the code region segment. If any of these conditions are encountered, we 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, we also profile the behavior of the gadget. Gadgets are labeled by the atomic operation they perform (§2). In practice, individual gadgets usually adhere to the concept of atomic operations due to the difficulty of accounting for side effects of longer sequences. While we experimented with many types of gadget profiles, only a few proved useful in reliably distinguishing actual `ROP` payloads from benign `ROP`-like data. These profiles are `LoadRegG`, and `JumpG/CallG/PushAllG/PushG` (we also refer to this entire set as `CallG`) which precisely map to `pop`, `jmp` and `jmpc`, `call`, `pusha`, and `push` instruction types. Thus, if we observe a `pop`, for example, the gadget is labelled as a `LoadRegG`, ignoring any other instructions in the gadget unless one of the `CallG` instructions is observed, in which case the gadget is labelled with `CallG`. More instructions could be considered (*i.e.* `mov eax, [esp+10]` is another form of `LoadRegG`), but we leave these less common implementations as future work. Note that if a gadget address corresponds directly to an API call[8], we label it as such, and continue to the next gadget. The usefulness of tracking these profiles should become apparent next.

3.4 Step ❹: `ROP` Chain Profiling

In the course of profiling individual gadgets, we 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, we profile each gadget as in step ❸, then select the next gadget using the stack offset produced by the previous gadget. We 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, our task is to determine if it represents a malicious `ROP` payload or random (benign) data. In the former case, we trigger an alert and provide diagnostic output; in the latter, we return to step ❶ and advance the sliding window by one byte.

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

[8] We also consider calls that jump five bytes into an API function to evade hooks.

Unfortunately, the distinction between benign and malicious `ROP` chains is not immediately obvious. For example, contrary to the observations of Polychronakis and Keromytis [36], 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.

That said, chains of length 2 are difficult to use in real-world exploits. The difficulty arises because a useful `ROP` payload will often 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 2. 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 our `LoadRegG` profile, while a gadget that actually calls the API fits either the `JumpG`, `CallG`, `PushAllG`, or `PushG` profiles. Our 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. We found this intuitive heuristic to be sufficient to reliably detect all real-world malicious `ROP` chains. However, by itself, the above strategy would lead to false positives with very short chains, and hence we apply a final filter. When the total number of unique gadgets is ≤ 2, we require that the `LoadRegG` gadget loads the value of a system API function pointer. Assuming individual gadgets are discrete operations (as in §2), there is no room for the adversary to obfuscate the API pointer value between the load and call gadgets. On the otherhand, if the discrete operation assumption is incorrect we may miss payloads that are only 1 or 2 unique gadgets, which we have not actually observed in real-world payloads. Empirical results showing the impact of varying the criteria used in our 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 Evaluation

We now turn our attention to a large-scale empirical analysis where our static ROP chain profiling technique is used to effectively distinguish malicious documents from benign documents. Our benign dataset includes a random subset of the Digital Corpora collection[9] provided by Garfinkel et al. [18]. We analyzed 7,662 benign files that included 1,082 Microsoft Office, 769 Excel, 639 PowerPoint, 2,866 Adobe Acrobat, and 2,306 `html` documents evaluated with Internet Explorer. Our 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 2, 47 `pdf` documents collected in the wild that exploit CVE-2010-{0188,2883}, two payloads compiled using the `jit-rop` framework [41] 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 were served via the `metasploit` framework.

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

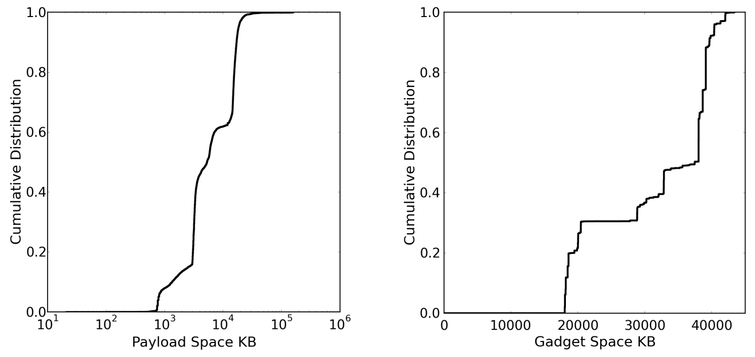
4.1 Results

Figures 4(a) and 4(b) 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 our dataset, 98% of the snapshots have a payload size less than 21 MB, and the largest payload space was 158 MB. We remind the reader that the number of bytes in the payload space is directly related to the number of gadget space lookups we 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(b). 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). We found that 98% of benign application snapshots contain less than 42 MB of code. Note that if a malicious ROP payload were present, all of it's gadgets must be derived from the gadget space of that particular application instance.

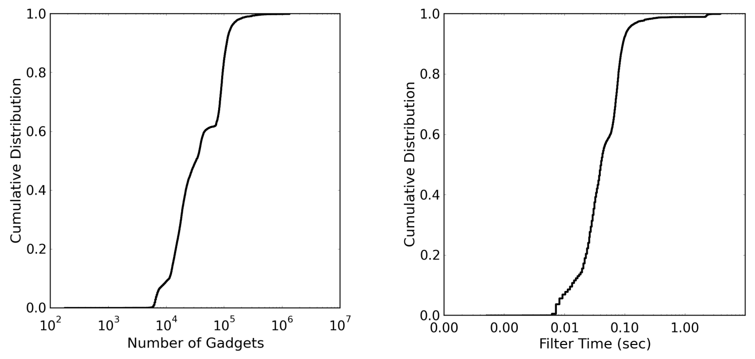
Our static ROP chain profiling captures the interaction between the payload and gadget spaces of an application snapshot. Each 4-byte chunk of data in the payload space that happens to correspond to a valid address in the gadget space triggers gadget and chain profiling. Figure 5(a) depicts the cumulative distribution of the number of times gadget candidate profiling was triggered over all benign snapshots. Not surprisingly, we observed that even within benign

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



(a) Payload space size. (b) Gadget space size.

Fig. 4: Payload and gadget space size for the benign dataset.



(a) Number of gadget candidates. (b) Execution time.

Fig. 5: Number of candidates and the corresponding runtime for the benign dataset.

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, typically contains many pointers into gadget space in the form of return addresses that were pushed by function calls. The heap(s) of an application may also contain function pointers used by the application—for example, an array of function pointers that represent event handlers.

Figure 5(b) depicts the cumulative distribution of the total time to apply static ROP chain profiling steps ❷ to ❹, which closely correlates with the total number of gadget candidates shown in Figure 5(a). The runtime demonstrates the efficiency of our technique, with 98% of documents taking 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.

Using the heuristic described in §3, we experienced no false positives on any of the 7,662 benign documents. However, we find it instructive to provide a deeper analysis on the benign `ROP` chains we did encounter that were not flagged as malicious. This analysis helps us understand *why* we did not have false positives in relation to the rules used by our heuristic. To do so, we relax some of our criteria from steps ❸ and ❹ to gauge the adverse impact on false positives that these criteria are meant to prevent.

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 1: An analysis of our profiling rules that significantly impact false positives.

First, we relax our 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 §3 step ❸), deemed the *assignment rule*. Second, we discard the requirement of having a system call pointer used by `LoadRegG` in 2-gadget chains (see §3 step ❹). We also test 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 our overall profiling criteria (e.g., some real-world `ROP` chains may assume specific values are pre-loaded into registers). The results are organized in Table 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 our entire dataset are composed of 5 or less gadgets per chain, thus taking advantage of both these rules to filter benign chains.

There be dragons: We now turn our focus to the malicious document samples in our dataset. Our heuristic precisely captures the behavior of our ideal 2-gadget `ROP` payloads and the simple `pusha` example, which are all identified successfully.

To see why, consider that our technique is used to analyze the `ROP` chain given in Figure 6. Clearly, 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 our heuristic, real-world examples are much more interesting.

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

Fig. 6: 2-gadget `ROP` chain (from a malicious document) that calls the `WinExec` API

Of the 47 samples captured in the wild that exploit CVE-2010-{0188,2883} with a malicious `pdf` document, 15 caused Adobe Acrobat to present a message indicating the file was corrupt prior to loading in step ❶. Therefore, no `ROP` was identified in these application snapshots. It is possible that an untested version of Adobe Acrobat would have enabled opening the document; however, selecting the correct environment to run an exploit in is a problem common to any approach in this domain. We discarded these 15 failed document snapshots. Our heuristic triggered on all of the 32 remaining document snapshots. Traces of portions of the `ROP` chain that triggered our heuristic are given in Appendix §A. The two `jit-rop` payloads triggered our 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.

5 Limitations

The astute reader will recognize that our criteria for labeling a gadget as valid in Step ❷ is quite liberal. For example, the instruction sequence `mov eax,0; mov [eax],1; ret;` would produce a memory fault during runtime. However, since our static analysis does not track register values, this gadget is considered valid. We acknowledge that although our approach for labeling valid gadgets could potentially lead to unwanted false positives, it also ensures we do not accidentally mislabel real `ROP` gadgets as invalid.

We note that while our static instruction analysis is intentionally generous, there are cases that static analysis can not handle. First, we can not track a payload generated by polymorphic `ROP` [28] with purely static analysis. To the best of our knowledge, 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 [30] to confuse static analysis; however, application of these techniques is decidedly more difficult when only reusing existing code. Regardless, static analysis alone cannot handle all cases of `ROP` payloads that make use of register context setup during live exploitation. In addition, our gadget profiling assumes registers must be assigned before they are used, but only when used in memory operations. Our results (in §4) show we could relax this assumption by only applying the assignment rule on small ROP chains.

6 Other Related Work

Most germane is the work of Polychronakis and Keromytis [36], 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 our work, ROPScan only analyzes pointers to non-randomized modules which is quite limiting since today's exploits place no restriction on the reliance of non-randomized modules; instead they exploit memory leakage vulnerabilities and calculate code pointers on-the-fly, thereby circumventing detection mechanism that only focus on non-randomized modules. Moreover, the fact that execution must be performed from each code pointer leads to poor runtime performance.

Similarily, Davi et al. [12] and Chen et al. [8] offer rudimentary techniques for detecting the execution of a return-oriented programming payload 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. Clearly, these techniques are fragile and can easily be circumvented by invoking longer sequences in between return instructions.

Arguably, one of the most natural approaches for thwarting code reuse attacks is to simply prevent the overwrite of code pointers in the first place. For instance, conventional stack smashing attacks rely on the ability to overflow a buffer in order to overwrite adjacent control-flow information [3]. Early defense techniques attempted to prevent such overwrites by placing so-called stack canaries between local variables and sensitive control-flow information [11]. Unfortunately, stack canaries only provided protection for return addresses, but not for function pointers. Subsequently, more generic approaches were developed, including buffer bounds checking, type-safety enforcement [31], binary instrumentation [13], as well as data-flow integrity (DFI) [2, 6]. Unfortunately, these advanced techniques either focus on non-control data attacks [9] or impose high runtime overhead. Additionally, DFI solutions require access to source code to determine the boundaries of variables and to determine which code parts are allowed to write into a specific variable.

A recent line of inquiry (e.g., return-less kernels [26] and G-Free [33]) for mitigating the threat of return-oriented programming relies on the ability to eliminate the presence of so-called unintended instruction sequences, which can be executed by jumping into the middle of an instruction. Moreover, G-Free mitigates both return- and jump-oriented programming by encrypting return addresses and ensuring that indirect jumps/calls can only be issued from a function that was entered from it originally. Unfortunately, both approaches require access to source code and re-compilation of programs — i.e., factors that limit their widespread applicability.

Yet another line of defense is to monitor and validate the control-flow of programs at runtime. In particular, program shepherding uses binary-based instrumentation to dynamically rewrite and check control-flow instructions [23]. For instance, return instructions are forced to transfer control to a valid call site, *i.e.,* an instruction that follows a call instruction. Control-flow integrity (CFI) goes a step further and enforces fine-grained control-flow checks for all indirect branches a program issues [1], effectively defeating conventional and advanced code reuse attacks. That said, the fact that CFI has yet to be shown practical for COTS binaries remains one limiting factor in its adoption. While some CFI-based follow-up work (*e.g.,* [21, 47]) have attempted to tackle this deficiency, the deployed CFI policies are usually too coarse-grained in practice.

7 Conclusion

In this paper, we introduce a novel framework for detecting code reuse attacks lurking within malicious documents. Specifically, we show how one can efficiently capture memory snapshots of applications that render the target documents and subsequently inspect them for ROP payloads using newly developed static analysis techniques. Along the way, we shed light on several challenges in developing sound heuristics that cover a wide variety of ROP functionality, all the while maintaining low false positives. Our large-scale evaluation spanning thousands of documents show that our approach is also extremely fast, with most analyses completing in a few seconds.

8 Acknowledgments

We thank the anonymous reviewers for their insightful comments. This work is funded in part by the National Science Foundation under award numbers 0915364 and 1127361.

A Example detection and diagnostics

Once `ROP` payloads are detected, we are able to provide additional insight on the behavior of the malicious document by analyzing the content of the `ROP` chain. Figure 7 depicts sample output provided by our static analysis utility when our heuristic is triggered by a `ROP` chain in an application snapshot.

```
==== 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]
```

Fig. 7: 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.

The first trace (top 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. We detect this payload due to 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.

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

Bibliography

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

[2] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro. Preventing memory error exploits with wit. In *IEEE Symposium on Security and Privacy*, 2008.

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

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

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

[6] M. Castro, M. Costa, and T. Harris. Securing software by enforcing data-flow integrity. In *USENIX Symposium on Operating Systems Design and Implementation*, 2006.

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

[8] P. Chen, H. Xiao, X. Shen, X. Yin, B. Mao, and L. Xie. Drop: Detecting return-oriented programming malicious code. In *Intl. Conf. on Information Systems Security*, 2009.

[9] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer. Non-control-data attacks are realistic threats. In *USENIX Security Symposium*, 2005.

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

[11] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX Security Symposium*, 1998.

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

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

[14] 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 & Vulnerability Assessment*, 2009.

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

[16] M. Frantzen and M. Shuey. Stackghost: Hardware facilitated stack protection. In *USENIX Security Symposium*, 2001.

[17] Gadgets DNA. 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/`.

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

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

[20] jduck. The latest adobe exploit and session upgrading. `https://community.rapid7.com/community/metasploit/blog/2010/03/18/the-latest-adobe-exploit-and-session-upgrading`, 2010.

[21] M. Kayaalp, M. Ozsoy, N. A. Ghazaleh, and D. Ponomarev. Efficiently securing systems from code reuse attacks. *IEEE Transactions on Computers*, 99 (PrePrints), 2012.

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

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

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

[25] T. Kornau. Return oriented programming for the ARM architecture. Master's thesis, Ruhr-University, 2009.

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

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

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

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

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

[31] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer. Ccured: type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems*, 2005.

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

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

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

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

[36] M. Polychronakis and A. D. Keromytis. ROP payload detection using speculative code execution. In *MALWARE*, 2011.

[37] F. J. Serna. The info leak era on software exploitation. In *Black Hat USA*, 2012.

[38] H. Shacham. 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*, 2007.

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

[40] K. Z. Snow, S. Krishnan, F. Monrose, and N. Provos. Shellos: enabling fast detection and forensic analysis of code injection attacks. In *USENIX Security Symposium*, 2011.

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

[42] E. H. Spafford. The Internet worm: Crisis and aftermath. *Communications of the ACM*, 32(6):678–687, 1989.

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

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

[45] P. Vreugdenhil. Pwn2Own 2010 Windows 7 Internet Explorer 8 exploit. , 2010.

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

[47] Y. Xia, Y. Liu, H. Chen, and B. Zang. Cfimon: Detecting violation of control flow integrity using performance counters. In *IEEE/IFIP International Conference on Dependable Systems and Networks*, 2012.

[48] D. D. Zovi. Practical return-oriented programming. RSA Conference, 2010.