ROP Payload Detection Using Speculative Code Execution

Michalis Polychronakis
Columbia University
mikepo@cs.columbia.edu

Angelos D. Keromytis
Columbia University
angelos@cs.columbia.edu

Abstract

The prevalence of code injection attacks has led to the wide adoption of exploit mitigations based on non-executable memory pages. In turn, attackers are increasingly relying on return-oriented programming (ROP) to bypass these protections. At the same time, existing detection techniques based on shellcode identification are oblivious to this new breed of exploits, since attack vectors may not contain binary code anymore. In this paper, we present a detection method for the identification of ROP payloads in arbitrary data such as network traffic or process memory buffers. Our technique speculatively drives the execution of code that already exists in the address space of a targeted process according to the scanned input data, and identifies the execution of valid ROP code at runtime. Our experimental evaluation demonstrates that our prototype implementation can detect a broad range of ROP exploits against Windows applications without false positives, while it can be easily integrated into existing defenses based on shellcode detection.

1 Introduction

The exploitation of memory corruption vulnerabilities in server and client applications has been one of the prevalent means of system compromise and malware infection. By supplying a malicious input to the target application, an attacker can inject and execute arbitrary code, known as shellcode, in the context of the vulnerable process. Fortunately, the wide adoption of non-executable memory page protections like Data Execution Prevention (DEP) [11] in recent versions of popular OSes has reduced the impact of conventional code injection attacks.

In turn, attackers have started adopting a new exploitation technique, widely known as *return-oriented programming* (ROP) [17], which allows the execution of arbitrary code on a victim system without the need to inject any code. In the same spirit as in the return-to-libc exploitation technique [19], return-oriented programming relies on the execution of code that already exists in the address space of the process. In contrast to return-to-libc though, instead of executing the code of a whole library function, return-oriented programming is based on the combination of tiny code fragments, dubbed *gadgets*, scattered throughout the code segments of the process. The execution order of the gadgets is controlled through a sequence of gadget addresses that is part of the attack payload. This means that an attacker can execute arbitrary code on the victim system by injecting only control *data*.

Besides the effective circumvention of non-executable page protections, return-oriented programming also poses significant challenges to a broad range of defenses that are based on shellcode detection [4, 13–15, 18, 21, 24, 25]. The main idea behind these approaches is to execute valid instruction sequences found in the inspected data on a CPU emulator, and identify characteristic behaviors exhibited by different shellcode types using runtime heuristics. Besides the detection of code injection attacks at the network level [13–15, 18, 25], shellcode identification has been used for in-browser detection of drive-by download attacks [8, 9], as well as malicious document scanning [18, 22].

In a ROP exploit, however, in place of the shellcode, the attack vector contains just a chunk of data—to which we refer as the *ROP payload*—comprising the addresses of the gadgets to be executed along with any necessary instruction arguments. Since there is no injected binary code to identify, existing emulation-based shellcode detection techniques are ineffective against ROP attacks. At the same time, return-oriented programming is increasingly used in the wild to broaden the targets of exploits against Acrobat Reader and other popular applications, extending the infection coverage of recent exploit packs [5].

As a step towards filling this gap, we present a new technique for the detection of ROP exploits based on the identification of the ROP payload that is contained in the attack vector. *ROPscan*, our prototype implementation, uses a code emulator to speculatively execute code fragments that already exist in the address space of a targeted process. The execution is driven by valid memory addresses that are found in the injected payload, and which could possibly

point to the actual gadgets of a malicious ROP code. We have evaluated ROPscan using an array of publicly available ROP exploits against Windows applications, as well as with a vast amount of benign data. Our results show that ROPscan can accurately detect existing ROP exploits without false positives, while it achieves an order of magnitude higher throughput compared to Nemu [13, 14], an existing shellcode detector with which ROPscan shares the code emulation engine.

Current exploits use ROP code only as a first step to bypass memory protections and to enable the execution of a second-level conventional shellcode, which is included in the same attack vector and thus can be identified by existing shellcode detectors. However, the embedded shellcode can easily be kept unexposed through a simple packing scheme, and get dynamically decrypted by a tiny ROP-based decryption routine, similarly to simple polymorphic shellcode engines. It has also been demonstrated that return-oriented programming can be used to execute arbitrary code [17], and thus future exploits may rely solely on ROP-based malicious code.

In any case, the ability to identify the presence of ROP code can increase the detection accuracy of current defenses that rely only on shellcode detection. ROPscan can inspect arbitrary data, which allows its easy integration into existing detectors—we present two case studies in which we have used ROPscan as part of a network-level attack detector and a malicious PDF scanner.

2 Background and Related Work

The ability to identify the presence of shellcode in arbitrary data inputs, such as network traffic [12–15, 18, 21, 23–25], process buffers [8, 9, 22], or memory dumps [18], offers an effective way to detect a broad range of code injection attacks. This alleviates the need to take into consideration the specifics of the exploitation method used, or the actual vulnerability being exploited—the mere presence of shellcode in a network request, a memory buffer, or a malicious file denotes suspicious activity.

Initial shellcode detection approaches used code disassembly on network streams to identify the NOP sled [21] or the shellcode itself [12, 23]. Static code analysis though is not effective in the presence of code obfuscation or self-modifying code, techniques that are widely used for shellcode packing and polymorphism. Dynamic code analysis using emulation can effectively handle even highly obfuscated code, and therefore has been used extensively for shellcode detection [4, 13–15, 18, 24, 25].

As ROP code has started replacing conventional shellcode in recent exploits, in this work we build on the concept of dynamic code analysis with the goal to detect the presence of ROP payloads in arbitrary inputs. Our prototype

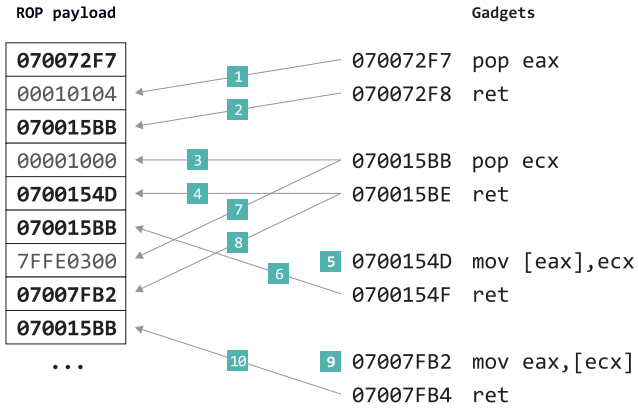

Figure 1. Example of ROP code taken from an exploit against Adobe Reader (CVE-2010-0188). The execution of the gadgets (right) is driven by the arrangement of gadget addresses and embedded data in the ROP payload (left). Arrows denote read accesses to payload data, and numbers correspond to the order of the executed instructions.

system is based on Nemu [13, 14], a shellcode detector that uses a CPU emulator to identify the execution behavior of various shellcode types using different runtime heuristics.

During execution, the shellcode may access data that already exist in the address space of the vulnerable process. To execute shellcode correctly, Nemu uses a fully-blown virtual memory subsystem that can be initialized with a snapshot of the complete address space of a real process. We take advantage of this feature to speculatively trigger the execution of gadgets that already exist in the executable memory segments of the vulnerable application we aim to protect, according to the ROP payload in the attack vector.

Each gadget of the ROP code transfers control to the next one through an indirect control transfer instruction—the final one in its sequence of "useful" instructions. The target addresses are read sequentially from the sequence of gadget addresses contained in the injected ROP payload, as shown in the example of Figure 1.

The gadgets usually end with a `ret` instruction—hence the name of the technique [17]—although any other indirect jump instruction can be used [6]. The `ret` instruction is a perfect fit for transferring control to the next gadget because it actually performs two operations at once: sets the instruction pointer (`EIP`) to the address contained in the memory location pointed to by the stack pointer (`esp`), and increments the stack pointer by four bytes (assuming an address size of 32 bits). This allows `esp` to be used as an "index" register for transferring control to the desired gadget according to the list of addresses in the ROP payload.

3 Approach

Our goal is to identify the presence of a ROP payload in arbitrary data, such as network traffic streams or process memory buffers. Each input is simply treated as a sequence of bytes, without any knowledge about the actual type or structure of the data. Consequently, if an input actually contains a ROP payload, its location is initially unknown. ROPscan searches the whole input to identify sequences of valid addresses that, when treated as a ROP payload, yield an actual execution path that spans several gadgets.

3.1 Setting up the Environment

To execute the sequence of gadgets used in an exploit, ROPscan should have access to the executable memory segments of the targeted process in which the gadgets reside. For this reason, the virtual address space of the emulator is initialized with a snapshot of the process memory from a real instance of each application we aim to protect. Multiple address spaces can coexist at the same time by maintaining a set of different page tables. This is useful in case an input needs to be checked in the context of more than one vulnerable processes.

Consider for example the case of a PDF file scanner, as the one described in Section 5.2. Different versions of Acrobat Reader may have the same DLL mapped into different addresses, and more than likely there will be differences in the actual code of some segments. The construction of ROP code is based on a particular static memory layout of the targeted application, and even a slight variation in one of the gadgets may break its execution. Exploits also may load on demand DLLs or executable components that are not loaded by default by the application. It is thus desirable to be able to check the same buffer for ROP payloads that would be valid in the context of different versions of Acrobat Reader, i.e., different memory layouts. Besides different versions of the same application, in other settings, such as the network-level detector described in Section 5.1, an input may also be inspected in the context of several different applications.

3.2 Speculative Execution

A working ROP exploit should contain a sequence of valid memory addresses in its ROP payload, each pointing to an actual gadget in the executable address space of the targeted process. A key characteristic of ROP code is that it relies on gadgets that exist in the non-ASLR code segments of the process, which remain static across different process instances or system configurations.

These segments are often a small subset of all allocated pages, which in turn are a subset of the whole virtual address space of a process (2GB for the default configurations of 32-bit Windows). We collectively refer to all the non-volatile memory segments of a process that have execute permission as its *gadget space*.

For randomized processes, the gadget space is even smaller, usually comprising the code segments of just a few non-ASLR DLLs. For the application and OS combinations we tested, the gadget space ranges from 28KB to just 17.71MB, as shown in Table 1. This means that the probability of an arbitrary address to fall within the gadget space is significantly low. Note that for 64-bit systems, this probability is even smaller due to the massive size of the available address space. In this work, we focus on 32-bit processes, since even in 64-bit versions of Windows the most commonly exploited applications are still 32-bit.

Based on the above observation, the first step of the detection algorithm is to identify potential gadget addresses within the scanned input. This is achieved by advancing a 4-byte sliding window one byte at a time, and checking whether the 32-bit address that corresponds to the current location of the window falls within the gadget space of any of the protected applications. In the common case, a random address will fall either into an unmapped or non-executable memory page, or in the kernel address space (upper 2GB of the total 4GB), as shown in Figure 2(a). Address `0072F741` is not mapped, and the sliding window advances to the next byte of the input.

If the address falls into the gadget space of a process, then this may denote the beginning of a ROP payload. In that case, ROPscan assumes that the address corresponds to the first gadget of the ROP code, and speculatively starts executing the code that exists at that address. Figure 2(b) illustrates the moment at which the sliding window reaches the first gadget address of the payload shown in the example of Figure 1. The address falls into the gadget space of the process (specifically, in the code segment of Adobe Reader's `BIB.dll`, which is the sole source of the gadgets used in this particular exploit), so `EIP` is loaded with address `070072F7` (the bytes of which are in reverse order in the payload due to endianness).

As discussed in Section 2, for the proper execution of the ROP code, the attacker needs to control both the `EIP` and `esp` registers. The latter is crucial for the correct transfer of control to the second gadget after the first one has completed. For this reason, before the beginning of a new execution, the `esp` register is set to point right *after* the four bytes of the first gadget's address in the input buffer, as shown in Figure 2(b). This corresponds to the state of the vulnerable process right after the flow of control has been hijacked, and is usually the outcome of a stack pivot instruction sequence [10, 26] (for exploits in which the stack pointer does not happen to point right at the beginning of the ROP payload).

In this example, the first gadget pops the next 4-byte

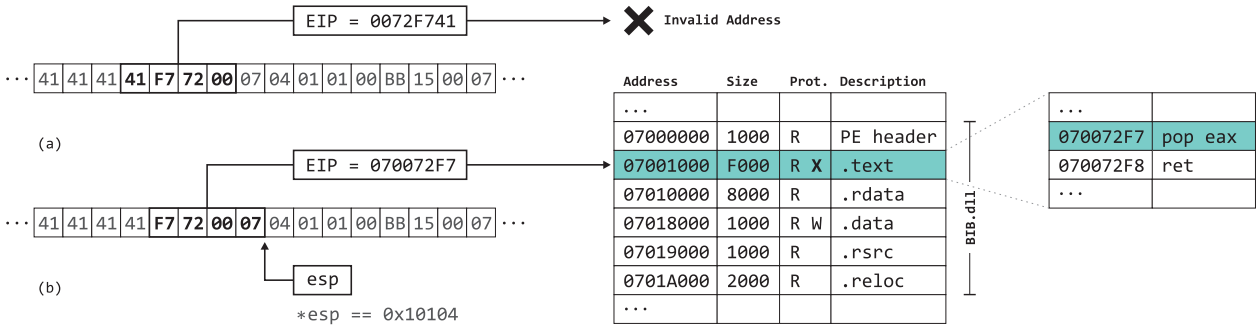

Figure 2. Overview of the scanning process. If the 4-byte value at the current position does not correspond to a mapped executable memory page, the sliding window advances one byte (a). When a valid address is found, `EIP` and `esp` are initialized appropriately and a new execution begins (b).

value from the ROP payload into `eax`, and transfers control to the next gadget through the `ret` instruction. The execution continues normally as long as each gadget manipulates the stack pointer correctly, and may terminate for one of the following reasons: i) a gadget transfers control to an invalid address, ii) the emulator encounters an invalid or privileged instruction, iii) the number of executed instructions in the current gadget reaches a certain threshold, or iv) the total number of executed instructions reaches an overall execution threshold.

The second condition is possible due to the variable-length instruction set of the x86 architecture. For example, a random address in a benign input may fall into the middle of an actual instruction in one of the code segments. That byte may correspond to the opcode of a privileged instruction that only the kernel is allowed to execute.

The third condition helps distinguishing between random code and actual ROP code. The typical size of the gadgets used in Turing-complete implementations [6, 17], as well as in the exploits we tested, ranges between 2–5 instructions, while the largest number of executed instructions in a single gadget that we observed is 10 instructions (EDB-ID 16619 in Table 1). We have conservatively set a gadget threshold of 32 instructions.

The final execution threshold ensures that the execution will stop in case the flow of control has been "trapped" into a loop or an overly long straight-through code path. Although the largest number of executed instructions in the ROP exploits we have encountered so far is less than 500, we have set a conservative threshold of 4096 instructions.

3.3 Runtime Detection

It is common for a totally benign input to contain one or more 4-byte values that fall within the gadget space, and which consequently point to valid instruction sequences. Depending on their arrangement in the input buffer and the final instruction of each sequence, a benign input may result in an execution chain of "accidental" gadgets that exhibits a ROP-like behavior. For the accurate detection of real ROP payloads, we need to be able to distinguish between the accidental execution of random instruction sequences and the actual execution of real gadgets. This is achieved using a runtime heuristic that precisely matches the execution behavior of ROP code.

We observe that the transfer of control to a subsequent gadget is always achieved through an indirect branch instruction, and its control data is always derived from the injected ROP payload. That is, the branch instruction itself (in case of `ret`, as shown in Figure 1), or some previously executed instruction in the same gadget (in case of indirect `jmp` or `call`), *reads* the destination address from the ROP payload. For example, gadgets that end with a non-`ret` instruction [6] use a sequence like `pop eax; jmp eax;` to first read the destination address from the payload and then jump to it. Therefore, we consider that the execution of an instruction sequence corresponds to an actual gadget if it *ends with an indirect control transfer instruction that uses control data derived from the original input buffer.*[1]

During the execution of an instruction sequence, if a `jmp eax` instruction transfers control to another valid location in the gadget space, but the value of `eax` has not been loaded from the input buffer, then this sequence is clearly not a gadget. Similarly, consider a relative `call` instruction that transfers control a few bytes further from the current location of `EIP`, followed at some point by a `ret` instruction. In this case, `ret` does not denote the end of a gadget although it reads an address from the payload and jumps to it, because the value read is not the original value that ex-

[1] In case a dispatcher gadget is used [6], gadgets first transfer control to it using a previously initialized register. Only the dispatcher gadget reads the next destination address from the payload and jumps to it. This does not pose any problem to our definition of a gadget's execution because the dispatcher gadget will be considered as part of the previous gadget.

isted at that location of the input buffer, but the return address pushed at runtime by the `call` instruction (each execution starts with a "clean" version of the original buffer using copy-on-write).

The above definition captures an essential property of the execution behavior of a gadget, which is rarely encountered during the execution of random code. Indeed, although it is very common for a benign input to contain addresses that correspond to random gadget-like instruction sequences, it is much less probable that one of these sequences will happen to *read* another valid destination address from the original input *and transfer* control to it—but sometimes, this may happen as well.

Fortunately, the ROP code of an exploit will rely on a set of several different gadgets, and there should be an uninterrupted flow of control from one to the other. Given that the same gadget can be executed several times (e.g., the second gadget at address `070015BB` in Figure 1) our detection heuristic is based on the number of *unique* gadgets that are encountered during the same execution chain. Although it is possible that a benign input will result to the execution of a few consecutive gadgets, setting a higher *detection threshold* provides for a robust detection heuristic that precisely captures the runtime behavior of ROP code. In the following section, we discuss how we can set this threshold so as to accurately detect existing ROP exploits, while practically eliminating the possibility of false positives.

4 Experimental Evaluation

We begin our evaluation by focusing on the resilience of the detection heuristic against false positives through stress-testing with benign data. We then test the detection effectiveness of ROPscan using existing ROP payloads, and finally discuss runtime performance and optimization issues.

4.1 Tuning the Detection Threshold

To assess the accuracy of ROPscan's detection heuristic, we tested our prototype implementation using a large and diverse set of benign data. Our aim is to verify the intuition that the execution patterns of the random code that can be triggered by valid addresses—which unavoidably occur in benign inputs—will not match the runtime behavior of the ROP code used in current exploits. This is crucial for ensuring that benign data are not falsely identified as containing a ROP payload.

The test inputs consist of randomly generated data, as well as real benign data. Specifically, we used a simple program that continuously generates inputs of varying size between 4–16KB with uniformly random binary and ASCII content. The data were fed directly to ROPscan, which inspected 100 million inputs of each type, totalling about 1.86TB of data. We also used traces of real network traffic captured at the access link of two production networks. The data set consists of about 7 million reassembled TCP streams with a maximum size of 64KB, totalling more than 196GB. Finally, we analyzed 923 benign PDF files with embedded JavaScript code using MDscan [22], and dumped the contents of the memory buffers allocated by the JavaScript interpreter that had a size larger than 128 bytes.

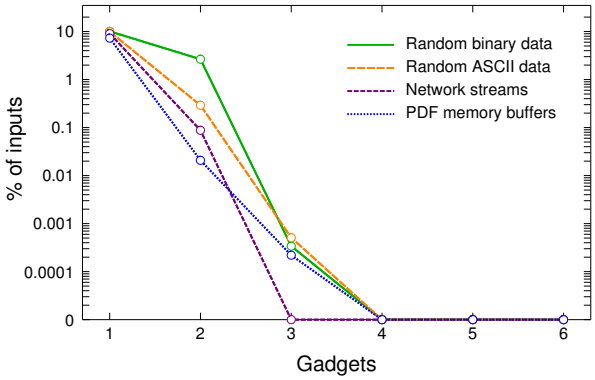

Figure 3. Percentage of benign inputs with a given maximum number of unique gadgets in the same execution chain.

For each input, we measure the maximum number of *unique* gadgets that happen to be executed as part of a single execution chain, according to the runtime pattern definition discussed in the previous section. To stress the detection algorithm, the emulator has been initialized with snapshots of multiple processes, which correspond to the applications listed in Table 1. This slightly increases the probability that a random address will fall into the combined gadget space of all processes.

Figure 3 shows the percentage of inputs with a given maximum number of unique gadgets in the same execution. For all kinds of data, about 7–10% of the inputs cause the execution of a single gadget. As discussed, a random instruction sequence is considered as a gadget only if it ends with an indirect branch with control data derived from the payload. Of course, the vast majority of the inputs trigger many other execution chains, but most of the time these do not end with a valid indirect branch, or are terminated due to the execution thresholds. The percentage of inputs with two gadgets ranges from 0.02% for memory buffers to 2.7% for random binary data, while an extremely small amount of inputs resulted to the execution of three gadgets.

In these experiments, as well as previous preliminary tests, we never observed a benign input with more than three unique gadgets. This means that, for the data sets we have used so far, setting a detection threshold of four gadgets

will never result in a false positive. Although we can never rule out the possibility of a false identification in another set of data or an actual long-term deployment, it is possible to raise the detection threshold even higher in order to increase the robustness or ROPscan against false alarms. Based on the results of the analysis of real ROP exploits that follows, the minimum number of unique gadgets used in the publicly available exploits we tested is eight, which allows for an execution threshold of up to eight gadgets.

We should note that there are several ways in which the detection heuristic could be strengthened even further in terms of accuracy. For instance, by manually analyzing the instructions of the instances with three unique gadgets, we observed that more than two thirds of them were due to three identical (but located in different addresses, and thus unique) single-instruction gadgets, each consisting solely of a `ret` instruction. Single-instruction gadgets alone cannot achieve anything useful other than advancing the stack pointer, so we could strengthen the heuristic by requiring the execution of a certain amount of gadgets with at least two or more instructions.

Furthermore, the current allowable maximum gadget length of 32 instructions is a quite conservative value, as discussed in Section 3.2, and could be lowered. About one third of the random gadgets in the tested benign inputs were overly long, between 16–32 instructions, and their execution could have been avoided by setting a lower maximum gadget length.

4.2 Detection Effectiveness

We evaluated the detection effectiveness of ROPscan using a set of eight publicly available ROP exploits against Windows applications. All exploits use a first-stage ROP code to bypass DEP and execute an embedded second-stage shellcode. Details about the exploits are listed in Table 1. The exploits are available through the Exploit Database [2] using the corresponding EDB-ID, and most of them are also included in Metasploit [3]. We also used four generic ROP payload implementations for bypassing Windows DEP [1, 7]. Two of them are based on gadgets from `msvcr71.dll`, a DLL that is included in (and remains static across) many popular applications [1].

For each exploit, we isolated the attack vector that contains the ROP payload, and fed it to ROP scan, which in all cases identified the beginning of the payload correctly. The last two columns in the table correspond to the total number of executed gadgets and the number of unique gadgets, respectively. When considering the detection heuristic used in ROPscan, in the worst case, one of the exploits against Adobe Reader uses just eight gadgets for its ROP code. When combined with the results of Section 4.1, this gives us a range of possible values for the detection threshold between 4–8 gadgets. A median value of six gadgets strikes a good balance between increased resilience to false positives, and the ability to detect even smaller ROP code implementations.

Note that in these exploits ROP code is used only to circumvent DEP, and the actual malicious functionality is carried out by conventional shellcode. A fully-blown ROP-based implementation of the same functionality or the addition of a decryption routine would probably require a larger number of gadgets.

4.3 Runtime Performance and Optimizations

The most CPU-intensive operation in ROPscan is the emulated execution of the code that is triggered whenever a new address from the input falls within the gadget space. Fortunately, the total size of the gadget space even when multiple process images are used is usually just a few tens of megabytes, as shown in Table 1, which is a fraction of the 4GB of addressable space using a 32-bit address.

Even whenever an execution chain is spawned, it usually ends very soon, as the occurrence of long valid instruction sequences is quite rare. This allows ROPscan to achieve a high raw processing throughput, despite the reliance on CPU-intensive interpretive emulation, which in our experiments exceeded 120Mbit/s on average. This allows it to be easily used in tandem with the legacy shellcode detection heuristics of Nemu, which achieve about an order of magnitude lower throughput [13].

Implementing the detection algorithm of ROPscan in a shellcode detection system like ShellOS [18], which executes the inspected code using native execution through virtualization, would allow for a much higher processing throughput. Additionally, there is room for further performance optimizations in the detection approach itself. For instance, not all addresses in the gadget space correspond to actual gadgets. In fact, usually just a fraction of them point to useful instruction sequences. Assuming a given maximum gadget length, potential valid gadget addresses can be pre-marked in the address space of the emulator, e.g., with the aid of a gadget discovery tool [17, 20]. Then, instead of blindly attempting an execution whenever an address from the input happens to fall anywhere within the gadget space, ROPscan will consider for execution only the addresses that point to actual pre-marked gadgets, reducing significantly the cycles spent on code emulation.

5 Use Cases

The main detection engine of ROPscan can inspect and identify the presence of ROP payloads in arbitrary inputs.

Exploit/Payload	CVE	EDB-ID	Tested Platform	Gadget Space	Executed Gadgets	Unique Gadgets
Adobe Reader v9.3.0	2010-0188	16670	Windows XP SP3	17.7MB	47	8
Adobe Reader v9.3.0	2010-1297	16687	Windows XP SP3	17.7MB	60	12
Adobe Reader v9.3.4	2010-2883	16619	Windows 7 SP1	864KB	33	10
Adobe Reader v9.3.4	2010-3654	16667	Windows XP SP3	17.7MB	60	12
Winamp v5.572	-	14068	Windows 7 SP1	5.7MB	126	21
Integard Pro v2.2.0	-	15016	Windows 7 SP1	724KB	165	16
Mplayer Lite r33064	-	17124	Windows 7 SP1	6.4MB	179	16
All to MP3 Converter v2.0	-	17252	Windows XP SP3	9.4MB	388	16
msvcr71.dll [1]	-	-	Windows 7 SP1	228KB	11	9
msvcr71.dll [7]	-	-	Windows 7 SP1	228KB	12	11
mscorie.dll [1]	-	-	Windows 7 SP1	28KB	9	9
mfc71u.dll [7]	-	-	Windows 7 SP1	872KB	15	10

Table 1. Details of the tested ROP exploits [2,3] and generic ROP payloads [1,7].

This allows it to be used in a broad range of attack detection and analysis systems. In this section, we discuss two different settings in which we have used ROPscan to detect network-level attacks and malicious documents. We expect that ROPscan will be easy to incorporate in other shellcode detectors as well [4, 18, 24].

5.1 Network-level Detection

Shellcode identification has been widely used for code injection attack detection at the network level [13–15, 18, 25]. The ability to identify ROP payloads can extend the range of attacks that these systems can detect, especially for next-generation attacks that may rely on ROP-only implementations of their malicious code.

ROPscan has been implemented on top of the detection engine of Nemu [13, 14], which already has a network-level detection component based on passive network monitoring. In this setting, ROPscan can detect ROP payloads in the raw network data that are transmitted through a TCP stream. For instance, the attack vector of the exploit against Integard Pro (a filtering proxy server) is just a POST request to the web interface of the application that triggers a buffer overflow. Similarly, the exploits against the media player applications in Table 1 are based on malicious media files that take control of the application when opened. The ROP payload is contained in the raw data of the file, which can easily be transmitted to potential victims over the network.

For all above exploits, ROPscan was able to detect the ROP payload by scanning the attack traffic. Actually, each input is inspected twice, since Nemu also applies its runtime shellcode detection heuristics, which are based on the execution of network data itself. Shellcode detection using emulation is much more CPU-intensive compared to ROP payload detection, and thus the additional overhead due to ROPscan is negligible.

5.2 PDF Scanning

Return-oriented programming has been widely used in exploits against Adobe Reader, which has full DEP support since version 9.2.0. As shown in Table 1, one version can be successfully exploited even in Windows 7, since a few third-party DLLs do not support ASLR.

We incorporated ROPscan in MDScan [22], a malicious PDF scanner based on shellcode detection. MDScan extracts any JavaScript code contained in the scanned document and executes it on a JavaScript emulator. Most of the exploits against Adobe Reader use JavaScript code to trigger a memory corruption vulnerability and execute the embedded code. MDScan inspects each newly allocated memory buffer in the context of the JavaScript interpreter for the presence of shellcode. As with in the case of the network-level detector, with the addition of ROPscan each buffer is also scanned for the presence of ROP payloads.

We generated malicious PDFs of all four Adobe Reader exploits using Metasploit [3]. From these exploits, only CVE-2010-0188 does not rely on JavaScript, and thus its malicious payload is not exposed to MDScan. The ROP code of all three other exploits was successfully detected. Our preliminary tests with actual in-the-wild malicious PDFs have also been positive.

MDScan inspects only JavaScript buffers, which limits its detection capabilities against malicious PDFs that do not rely on JavaScript code. However, ROPscan can easily be used in other types of detectors that either scan *all* memory buffers of a process at runtime using library interposition [16], or scan raw dumps of specific memory areas [18].

6 Conclusion

Attackers always seek new ways to evade detection systems and bypass protection mechanisms. Return-oriented

programming is increasingly used in the wild in exploits against Windows applications to circumvent DEP, facilitated in part by the lack of full support for address space layout randomization in many vulnerable applications. The detection algorithm of ROPscan can identify the presence of ROP payloads in arbitrary inputs, and the results of our experimental evaluation demonstrate that it can easily extend the detection capabilities of existing defenses that are based solely on the detection of conventional shellcode. As part of our future work, we plan to explore possible optimizations in the performance and accuracy of the core detection algorithm, and incorporate it in other existing detectors.

Acknowledgments

This work was supported in part by the US Air Force, DARPA, and the NSF through Contracts AFRL-FA8650-10-C-7024 and DARPA-FA8750-10-2-0253, and Grant CNS-09-14312, respectively, and by the FP7-PEOPLE-2009-IOF project MALCODE, funded by the European Commission under Grant Agreement No. 254116. Any opinions, findings, conclusions, or recommendations expressed herein are those of the authors, and do not necessarily reflect those of the US Government, the Air Force, the NSF, or DARPA.

References

[1] http://www.whitephosphorus.org/

[2] http://www.exploit-db.com/

[3] http://www.metasploit.com/

[4] P. Baecher and M. Koetter. libemu. http://libemu.carnivore.it/

[5] K. Baumgartner. The ROP pack. In *Proceedings of the 20th Virus Bulletin International Conference (VB)*, 2010.

[6] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy. Return-oriented programming without returns. In *Proceedings of the 17th ACM conference on Computer and Communications Security (CCS)*, 2010.

[7] Corelan Team. Corelan ROPdb. https://www.corelan.be/index.php/security/corelan-ropdb/

[8] M. Cova, C. Kruegel, and G. Vigna. Detection and analysis of drive-by-download attacks and malicious JavaScript code. In *Proceedings of the 19th International World Wide Web Conference (WWW)*, 2010.

[9] M. Egele, P. Wurzinger, C. Kruegel, and E. Kirda. Defending browsers against drive-by downloads: Mitigating heap-spraying code injection attacks. In *Proceedings of the 6th international conference on Detection of Intrusions and Malware, & Vulnerability Assessment (DIMVA)*, 2009.

[10] Ú. Erlingsson. Low-level software security: Attack and defenses. Technical Report MSR-TR-07-153, Microsoft Research, 2007. http://research.microsoft.com/pubs/64363/tr-2007-153.pdf

[11] R. Hensing. Understanding DEP as a mitigation technology. 2009. http://blogs.technet.com/b/srd/archive/2009/06/12/understanding-dep-as-a-mitigation-technology-part-1.aspx

[12] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Polymorphic worm detection using structural information of executables. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection (RAID)*, Sept. 2005.

[13] M. Polychronakis, K. G. Anagnostakis, and E. P. Markatos. Comprehensive shellcode detection using runtime heuristics. In *Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC)*, December 2010.

[14] M. Polychronakis, E. P. Markatos, and K. G. Anagnostakis. Network-level polymorphic shellcode detection using emulation. In *Proceedings of the Third Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, July 2006.

[15] M. Polychronakis, E. P. Markatos, and K. G. Anagnostakis. Emulation-based detection of non-self-contained polymorphic shellcode. In *Proceedings of the 10th International Symposium on Recent Advances in Intrusion Detection (RAID)*, September 2007.

[16] P. Ratanaworabhan, B. Livshits, and B. Zorn. NOZZLE: A defense against heap-spraying code injection attacks. In *Proceedings of the* 18^{th} *USENIX Security Symposium*, Aug. 2009.

[17] H. Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and Communications Security (CCS)*, 2007.

[18] K. Z. Snow, S. Krishnan, F. Monrose, and N. Provos. ShellOS: Enabling fast detection and forensic analysis of code injection attacks. In *Proceedings of the 20th USENIX Security Symposium*, 2011.

[19] Solar Designer. Getting around non-executable stack (and fix). http://seclists.org/bugtraq/1997/Aug/63

[20] P. Solé. Hanging on a ROPe. http://www.immunitysec.com/downloads/DEPLIB20_ekoparty.pdf

[21] T. Toth and C. Kruegel. Accurate buffer overflow detection via abstract payload execution. In *Proceedings of the 5th Symposium on Recent Advances in Intrusion Detection (RAID)*, Oct. 2002.

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

[23] X. Wang, C.-C. Pan, P. Liu, and S. Zhu. Sigfree: A signature-free buffer overflow attack blocker. In *Proceedings of the USENIX Security Symposium*, Aug. 2006.

[24] G. Wicherski. libscizzle. http://code.mwcollect.org/projects/libscizzle

[25] Q. Zhang, D. S. Reeves, P. Ning, and S. P. Lyer. Analyzing network traffic to detect self-decrypting exploit code. In *Proceedings of the 2nd ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2007.

[26] D. A. D. Zovi. Practical return-oriented programming. SOURCE Boston, 2010.