ROPMEMU: A Framework for the Analysis of Complex Code-Reuse Attacks

Mariano Graziano
Cisco Systems, Inc.

Davide Balzarotti
Eurecom

Alain Zidouemba
Cisco Systems, Inc.

ABSTRACT

Code reuse attacks based on return oriented programming (ROP) are becoming more and more prevalent every year. They started as a way to circumvent operating systems protections against injected code, but they are now also used as a technique to keep the malicious code hidden from detection and analysis systems. This means that while in the past ROP chains were short and simple (and therefore did not require any dedicated tool for their analysis), we recently started to observe very complex algorithms – such as a complete rootkit – implemented entirely as a sequence of ROP gadgets.

In this paper, we present a set of techniques to analyze complex code reuse attacks. First, we identify and discuss the main challenges that complicate the reverse engineer of code implemented using ROP. Second, we propose an emulation-based framework to dissect, reconstruct, and simplify ROP chains. Finally, we test our tool on the most complex example available to date: a ROP rootkit containing four separate chains, two of them dynamically generated at runtime.

1. INTRODUCTION

Memory analysis and memory forensics are active research fields that have rapidly evolved over the past decade and they are now a popular, complementary approach to support modern malware analysis and inspect potentially compromised machines. The main focus on memory forensics (from a malicious code perspective) is to find intrusion evidences in the physical memory. Commonly, these evidences involve artifacts that has been created or injected in memory by malicious components. Volatility plugins like `psxview` and `malfind` are good example of tools that perform this task. Unfortunately, the current focus on *"code injection"* is unable to cope with the emerging trend of advanced threats that adopt *"code reuse"* techniques (such as return oriented programming) as a mean of obfuscation, to perform malicious computation without injected code.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ASIA CCS '16, May 30-June 03, 2016, Xi'an, China

© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ISBN 978-1-4503-4233-9/16/05…$15.00

DOI: http://dx.doi.org/10.1145/2897845.2897894

Return oriented programming (ROP) is a technique to execute code by reusing instructions already present in a program. Each sequence of instructions (called gadget) is responsible to fetch the address of the next one (typically from the stack using the `ret` instruction), thus gluing together many small gadgets to perform a predefined computation.

The ROP paradigm is in constant evolution due to the frequent release of new countermeasures against code reuse. In particular, tools like EMET [37] and kBouncer [45] considerably raised the bar and made simple techniques ineffective, forcing offensive researchers to devise more advanced forms of ROP. At the same time, software and hardware vendors introduced sandboxing mechanisms and light forms of control flow integrity in userspace, as well as several security enhancements in kernel-space. For example, UDEREF [54], SMAP [29], SMEP [17], NX regions, driver signing and KASLR [31] significantly hinder the kernel exploitation phase, and even when the attacker is able to control the instruction pointer, a considerable effort is still required to create a functional exploit. To cope with all these security improvements, attackers also adopted ROP chains as part of kernel exploits. The same trend can also be observed in other architectures. Jailbreak communities for Android and iOS are also responsible for some of the most complex public ROP chains, for instance as part of the Comex's iOS jailbreaks [26, 59]. As a last step of this constant evolution, in 2014 researchers were able to implement the first prototype of a complete and functional persistent rootkit in ROP [58].

Meanwhile, over the last decade, hundreds of academic papers and underground presentations focused on ROP. On the one hand, researchers proposed a wide range of defense mechanisms. On the other hand, they introduced improved ROP variations as well as tools to automate the chain creation phase. This constant arms race is still in progress today and resulted in a considerable increase in the *complexity* of ROP chains. While both the attack and the defense sides have been widely covered, the *analysis* of ROP chains has been completely overlooked and, in 2016, there is still not a single available framework to support their analysis.

In comparison, reverse engineering relies on a broad range of tools that have been perfected over the years, such as debuggers, disassemblers, and decompilers. Unfortunately, all these products were designed for "EIP-based"

programming and are of very little use to analyze stack-based return oriented programming payloads.

This is the problem we address in this paper. Specifically, two main observations motivate our work: the lack of public tools to analyze ROP payloads and the observation that ROP chains are rapidly growing both in size and in complexity.

To tackle these problems, we propose ROPMEMU – a framework for the automated analysis of ROP chains. We assume that, using existing techniques [55, 47, 33], a forensic investigator discovers a chain in system memory and needs to investigate its behavior. At first glance, the problem may seem trivial: it would be enough to dump the memory region containing the chain, reconstruct the entire code by appending the instructions contained in each gadget, and then analyze it like any other sequence of assembly instructions. However, in this paper we show that this procedure is in fact very complex and requires a number of dedicated tools and techniques.

ROPMEMU leverages techniques from the fields of memory forensics, emulation, multi-path execution, and compiler transformations to analyze complex ROP chains and recover their precise control flow graphs. Moreover, by using a novel multi-path emulation approach, our system is also able to reconstruct chains which are dynamically-generated at runtime, allowing an analyst to capture the behavior of the most complex attacks that can be encountered in the wild.

To summarize, this paper makes the following contributions:

- We propose the first memory analysis framework to analyze, dissect, reconstruct and simplify malicious code based on *code reuse* techniques.

- We discuss a number of practical challenges that need to be addressed to reverse engineer code implemented using return oriented programming. This goes far beyond what was observed in the past in simple exploits and what was discussed in previous papers.

- We tested our tool with the most complex example of this kind: a ROP rootkit containing chains with a total of 215,913 gadgets.

2. BACKGROUND

In this section, we provide the technical background required to understand the remaining part of the paper. We first introduce the return oriented programming paradigm and we summarize the available analysis techniques. We then provide an overview of the current trends and evolution of rootkit technologies as well as the recently-proposed concept of an ROP rootkit. Finally, we introduce in more detail the ROP rootkit proposed by Vogl et al. [58], that we will use as a case study throughout the rest of the paper.

2.1 Return Oriented Programming

Security countermeasures introduced in the last decade in modern operating systems forced attackers to adapt and find new ways to exploit programs. To overcome hardware defenses – such as the *no-execute* bit (NX) in PAE and IA-32e modes on Intel processors, software protections trying to emulate the NX bit behavior [53, 56, 57], and code signing [2, 3, 1] techniques – offensive researchers proposed several forms of *code reuse* attacks [39, 51, 50]. Over the years, *code reuse* attacks have been ported to different architectures [34, 12] and have evolved in a multitude of different techniques, such as return oriented programming without returns [13], jump return oriented programming [10], blind ROP [9], and sigreturn oriented programming [11].

ROP is now one of the most prevalent and widespread techniques adopted in the majority of the exploits observed in the wild. It is a particular instance of *code reuse* attack in which the attacker uses instructions already present in memory and chains them together to perform arbitrary computation. A single block of assembly instructions terminated by a `ret` (in its most traditional form) is called a *gadget*. A sequence of gadgets is then connected to form a *ROP chain* by putting their addresses on the stack and leveraging the `ret` instruction to return from one gadget to the next one. Please refer to the original formulation from Shacham [51] for a more detailed presentation of the internal details of return oriented programming.

ROP Analysis – In the past, ROP payloads were mainly used by exploits to disable the protection enforced by the NX bit and then execute a normal shellcode. In these cases, the ROP chain is generally very short, as its only goal is to invoke functions (e.g., `VirtualProtect` or `mprotect`) to change the page permissions of the memory containing the shellcode. As a consequence, the vast majority of ROP chains were in fact composed of a straight sequence of instructions without any branch or complex control flow.

There exists a countless number of offensive tools to simplify the creation of ROP chains – ranging from simple techniques [16, 30, 49, 7] to disassemble binaries, find gadgets and group them together – to more advanced tools [44, 40, 6] that use constraint solvers, intermediate languages and even emulators [42] to automate the chain creation as much as possible. Unfortunately, because of the simple form of the existing chains, to date no existing tool has been proposed to analyze ROP payloads.

More recently, researchers and malware writers discovered that return oriented programming is not only a useful technique to run exploits, but it also provides a very effective way to *hide* the malicious code. In fact, since ROP allows the implementation of a new functionality by reusing existing sequences of instructions, it makes the malicious code much more complex to identify, isolate, and analyze. As part of this emerging phenomenon, chains have started to contain complex application logic, therefore becoming much longer and much more complex. As a first example, malware samples have adopted ROP payloads to implement a dropper/downloader that fetches and runs the second stage [24]. Even more worrying, in 2014 Vogl et al. [58] presented the first complete example of a rootkit implemented in ROP.

In this paper we show that the complexity of these chain is well beyond what can be manually investigated by a human analyst. Therefore, we believe that this opens a new era for malicious code execution, and calls for a new set of tools and techniques to perform its analysis.

2.2 Rootkits

Rootkits are malicious software designed to gain persistent, *stealth* access to a compromised machine. In the last few years, rootkit technology has been rapidly evolving and increasing in sophistication. In order to conceal their presence and information, modern rootkits typically run at ring 0. This places the attacker at the same level as the OS kernel, so that the rootkit can undermine the security of the operating system and, potentially, remain undetectable for a long time. Several defensive mechanisms have been proposed to address this issue, but, unfortunately, ring 0 rootkits are still a severe threat. Offensive researchers have also investigated further possible ways to subvert the operating system security model moving deeper in the execution stack: prototypes exist for virtualization rootkits (ring -1) [32, 27, 19], SMM rootkits (ring -2) [20] and Intel ME rootkits (ring -3) [46]. The idea is to place the rootkit always a level lower than the defensive monitor in order to stay hidden on the compromised system.

Fortunately, all rootkits share a common weakness: they need to load their code into the running system. Modern countermeasures, such as secure boot and code signing, significantly hinder this process and make traditional attacking techniques no more effective against recent systems. To bypass these protection mechanisms, malware authors can use the same code reuse techniques adopted by exploit writers. A completely ROP rootkit was first theorized by Hund et al [23] in 2009. The authors proposed a proof of concept with several limitations. First of all, the malware had to repeatedly exploit a kernel vulnerability to execute arbitrary ROP payloads (e.g., to hide system processes). Second, this initial rootkit was not persistent – making it of little use in practice. In 2014, Vogl et al. [58] succeeded in making a ROP rootkit persistent and presented an open-source POC of their creation. In particular, the authors have shown how it is possible to perform hooking without injecting a single line of code in the kernel. In this case, the malware has to exploit the vulnerability only once to escalate privileges and trigger the persistent ROP payload.

Chuck – Chuck is the name of the persistent ROP rootkit proposed by Vogl et al [58], the only public example of this kind to date.

It includes four separate ROP chains: one persistent in memory and the other three dynamically generated at runtime. The first chain is the *initialization* chain and it is executed only once, the first time the kernel vulnerability is exploited (in this particular case CVE-2013-2094 [28]). This chain sets the hooks in the system, sets up the switching mechanism based on the `sysenter` instruction using the MSR registers 0x175 (`IA32_SYSENTER_ESP`) and 0x176 (`IA32_SYSENTER_EIP`), initializes the global state of the rootkit, prepares a memory region to deal with multiple invocations, and finally generates the second chain – the so-called *copy chain*. The *copy chain* is the only persistent ROP chain, and it is invoked every time a hook is triggered. It is also important to note that rootkit itself guarantees that the first chain is always and entirely stored in memory, even if its size is large [58]. First, it saves all the general purpose registers in the global state. Second, it creates and copies in memory a dynamic chain for each invocation of the hook. This third chain is called the *dispatcher chain*. The *dispatcher chain* is necessary to deal with hook invocations by multiple threads. The goal of this chain is to create a final, ad-hoc *payload* chain, which contains the core functionality of the rootkit and, at the end of its execution, it restores the original registers to resume the normal kernel execution.

The complexity of these four chains is considerably high. First of all, the size of a single chain is huge compared to the chains used by ordinary exploits. For instance the *copy chain* contains over 180k gadgets. Second, these chains have a non-linear control flow logic – making their analysis very complex. Third, the presence of dynamically generated chains make the analysis of this rootkit similar to a multi-stage packed malware, limiting the applicability of static analysis. Finally, the four chains compose a real kernel rootkit and thus the analyst has to deal with kernel issues such as privileged instructions and interrupts.

3. ROP ANALYSIS

To date, the complexity of analyzing ROP code has been completely underestimated. Few studies have focused on this problem, mainly taking simplistic approaches applied only to small examples. The first issue that an analyst may encounter when dealing with code reuse attacks is the fact that they are hard to locate in the first place. Since no code is injected in a ROP-based attack, finding the entry point of the chain can be difficult – especially when the input is the entire system memory. Three previous studies have proposed solutions for this problem [55, 47, 33] and therefore we will build on top of them for the rest of our paper.

However, locating the entry point is only the tip of the iceberg. The real analysis (i.e., what needs to be done *after* a chain has been located) is much more complex, and present a number of novel challenges:

[C1] Verbosity – the majority of ROP gadgets contain spurious instructions. For example, a gadget intended to increment `eax` may also pop a value from the stack before hitting the `ret` instruction that triggers the next gadget in the chain. Moreover, the code of a ROP chain contains a large percentage of return or other indirect control flow instructions, whose only goal is to connect the gadgets together. These are only few examples of why ROP code is very verbose and contains a large fraction of dead code that makes it harder for analysts to understand it. However, this is probably the simplest problem to solve as many transformations proposed in the compiler literature already exist to simplify assembly code.

[C2] Stack-Based Instruction Chaining – the most obvious difference between a ROP chain and a normal program is that in a chain the instructions are not consecutive in memory, but they are instead grouped in small gadgets connected together by indirect control flow instructions. So, what in a normal program could be a single block of 50 instructions, in a ROP chain can be split into more than 40 blocks chained by `ret` instructions.

At a first glance, this problem may seem trivial to solve. Since the addresses of each gadget in the chain are saved in

memory, one may erroneously think that it would be easy to automatically retrieve them, collect the corresponding pieces of code, and replace the entire chain with a single sequence of instructions. However, the stack-based instruction chaining can introduce subtle side effects that are hard to identify with a simple static analysis approach. For instance, since the sequence of gadgets is saved on the stack, but the code of each gadget also interacts with the stack (to retrieve parameters or just because of spurious instructions), in order to correctly identify the address of each gadget it is necessary to emulate every single instruction in the code.

[C3] Lack of Immediate Values – another difference between normal code and ROP chains is the fact that chains are typically constructed with "generic" gadgets (such as "store an arbitrary value in the `rax` register") that operate on parameters which are also stored on the stack. As a result, the vast majority of immediate values that are assigned to registers are interleaved on the stack with the gadget addresses. Again, code emulation is required to locate them and restore them back to their original position in the code.

[C4] Conditional Branches – in a ROP chain, a branch condition implies a change in the stack pointer instead of a more traditional change in the instruction pointer. This means that a simple conditional jump may be encoded with dozens of different instructions spanning multiple gadgets (e.g., to set the flag register according to the required condition, test its value, and conditionally increment the `esp` register). To translate the chain into more readable code, it is therefore necessary to identify these patterns based on their semantics and replace them with single branch instructions.

[C5] Return to Functions – function calls are typically implemented in ROP as simple return to the functions entry point. However, since normal gadgets are also often extracted from code located inside libraries, it is hard to distinguish a function call from another gadget. As it is the case for statically linked binaries, the lack of information on external library calls can make the reverse engineering process much more tedious and complicated.

[C6] Dynamically Generated Chains – the instructions of normal programs are typically located in a read-only section of the executable. Dynamically modified code is common in malware (e.g., as a result of packing) and, in fact, this severely limits the ability to perform static analysis on malicious code and considerably slows down the reverse engineering process. On the contrary, ROP chains are located on the stack, and it is therefore simple to use gadgets to prepare the execution of other gadgets in the future. This dynamicity means that it is not necessary for the entire chain to reside in memory at the same time – but it is instead common to build chains (or part thereof) on the fly.

[C7] Stop Condition – in this paper we assume that the analyst is able to locate the beginning of a ROP chain in memory. However, since an emulator is needed to analyze its content, it is important to also have a *termination condition* to decide when all the gadgets have been extracted and the emulation process can stop. The fact that complex ROP chains can invoke functions (which in turn may invoke other functions) interleaved with normal gadgets, and the fact that a chain can dynamically generate another chain in a different part of the memory, make this problem very hard to solve in the general case. For example, when does a ROP rootkit (that reuses instructions from the existing code in the kernel) terminate and the normal kernel tasks resume?

3.1 Implications

The previous seven challenges have several important implications for the analysis of ROP chains and previous works only proposed partial solutions. For instance, Lu et al. [36] and Yadegari et al. [60] identified a number of code transformations to handle **[C1]**. Moreover, Stancill et al. [55] and Lu et al. [36] used simple heuristics to follow the value of the stack pointer, thus partially addressing **[C2]** and **[C3]**. However, previous heuristics only applied to `ret`-based ROP chains, and were unable to follow indirect calls and jump instructions. Sadly, **[C4-7]** have never been mentioned before, probably because only in the past two years ROP chains have become complex enough to raise these points.

As it is better explained in Section 4, to fully address **[C2]**,**[C3]**, and **[C6]** it is necessary to emulate all the instructions in the chains and keep a shadow copy of the memory content. Moreover, a solution based on *multi-path emulation* is required to explore each path in the chain and retrieve its entire code. In turn, this approach requires the system to implement heuristics to detect the presence of branch instructions (**C4**). Finally, while recognizing functions (**C5**) can be addressed by using symbols information extracted from libraries and kernel functions, these functions often invoke system calls and this is a major obstacle for an emulator because their return values cannot be predicted with static analysis. Functions are not the only issue when using an emulator: precise heuristics for the stopping condition (**C7**) are also required and (as better explained in the next section) hard to implement.

This short discussion emphasizes how ROP analysis is in fact a multi-faced problem whose solution requires a combination of sophisticated techniques.

3.2 Adversarial Model

In this paper we assume that a motivated and well funded attacker was able to compromise a machine and successfully install malicious code in the form of a ROP chain. This can effectively bypass all the operating systems protections. For example, in the particular case of a ROP kernel rootkit, these protections include the kernel code integrity (e.g., PatchGuard), the driver signature enforcement, and the NX regions both at user-land and kernel-land (e.g., NX pools in Windows kernel from Windows 8).

We also assume a fully protected machine equipped with an Intel Ivy Bridge/Haswell CPU with the Supervisor Mode Execution Prevention (SMEP) and the Supervisor Mode Access Prevention (SMAP), and running an operating system (either Windows or Linux) that implements address space layout randomization (ASLR) in userspace, kernel memory, and for all the modules.

The infected machine can be either a bare-metal com-

puter or a guest virtual machine. In the second case, the memory forensic analysis may require an introspection system as the one provided by [22]. We also assume that the attacker can try to minimize the footprint of the malicious code by generating new gadgets at runtime and by overwriting them when they are not anymore needed.

Finally, we assume that the analyst was able to acquire a physical memory dump (e.g., by using pmem, lime, fmem, or by performing a DMA attack) after the malicious code became resident in memory. The technique proposed in this paper does not make any assumption on the ROP payload. The gadget and the chain have no constraint on the length as well as on the type of instructions. The case in which the chain is hostile and implement anti-analysis tricks is discussed in Section 6.

4. SYSTEM DESIGN

The ROPMEMU framework adopts a set of different techniques to analyze ROP chains and reconstruct their equivalent code in a form that can be analyzed by traditional reverse engineering tools. In particular, it is based on *memory forensics* (as its input is a physical memory dump), *code emulation* (to faithfully rebuild the original ROP chain), *multi-path execution* (to extract the ROP chain payload), *CFG recovery* (to rebuild the original control flow), and a number of *compiler transformations* (to simplify the final instructions of the ROP chain).

The framework is divided in different components that interact as shown in Figure 1 in five main analysis phases:

- **Multipath Emulation** - This step emulates the assembly instructions that compose the ROP chain. This is the only way to rebuild the exact instance of the running chain at the time of the dump. All the possible branches are explored and an independent trace (annotated with the values of registers and memory) is generated for each execution path (**C2** and **C6**). The emulator is also designed to recognize a number of returns-to-library functions, skip over their body, and simulate their execution by generating dummy data and return values (**C4**).

- **Trace Splitting** - In this phase the system analyzes all the traces generated by the emulator, removes the repetitions, and extracts the unique blocks of code.

- **Unchaining** - This phase applies a number of assembly transformations to simplify each ROP trace by removing the connections between gadgets and merging the content of consecutive gadgets in a single basic block. This step is also responsible to remove immediate values from the stack and assign them to the corresponding registers (**C2** and **C3**).

- **CFG recovery** - This pass merges all the code blocks in a single program, recovering the original control flow graph of the ROP chain. This phase comprises two steps. In the first one, the traces are merged in a single graph-based representation. The second step translates the graph into a real `x86` program by identifying the instructions associated to the branch conditions and by replacing them with more traditional EIP-based conditional jumps (**C4**). At the end of this phase, the program is saved in an ELF file, to allow traditional reverse engineering tools (e.g., IDA Pro) to operate on it.

- **Binary optimization** - In the final step, we apply known compiler transformations to further simplify the assembly code in the ELF file. For instance, this phase removes dead instructions in the gadgets and generates a clean and optimized version of the payload (**C1**).

In the rest of the section, we introduce each phase in detail and we describe how each of them have been implemented in our system.

4.1 Chain Discovery

Finding ROP chains in a physical memory dump is not a trivial task. However, three solutions have already been proposed in the literature [55, 47, 33] for this problem. Therefore, for the sake of simplicity, in this paper we assume that the analyst is provided with an image of the memory and an entry point of the first ROP chain.

Our case study was complicated by the fact that only one of the chains is persistent (the *Copy Chain*), while the other ones are generated on the fly depending on the system's state and therefore their content is only available in memory for few milliseconds. As a consequence, it is unrealistic to require an analyst to collect a snapshot of the memory containing all ROP chains – and therefore their content needs to be reconstructed by our system. The starting point of these dynamically generated chains is automatically derived from the emulation of the previous chain.

4.2 Emulation

The emulation phase is the core of our analysis framework. Its role is to "follow" the execution of each gadget to keep an updated position of the stack pointer and of the content of the memory.

A Turing complete ROP chain can be obtained by reusing a limited number of gadgets [51]. Therefore, also very complex ROP programs often include a very small number of *unique* assembly instructions. For this reason, we were able to implement a small custom emulator that supports the required `x86-32` and `x86-64` instructions and updates the state of the CPU (registers and flags) and of the memory after each instruction. In order to supporting the entire instruction set, we are now adapting our platform to use the recently released Unicorn [41] emulator. Unicorn provides the ideal solution for this task, by exposing Qemu's CPU emulator component and by proving a set of flexible bindings.

For a still more comprehensive approach, S2E [15] can be used to provide a full system emulation on top of Qemu [8]. However, this would considerably complicate the setup and deployment required by the system. Therefore, for our prototype we opted for a custom solution that gave us more flexibility and a smaller footprint.

At the beginning of the emulation phase, the initial state of the virtual CPU is set to zero by resetting the content of all registers except for the instruction pointer and the stack pointer (whose initial values need to be provided as input for our analysis). The emulator is then implemented

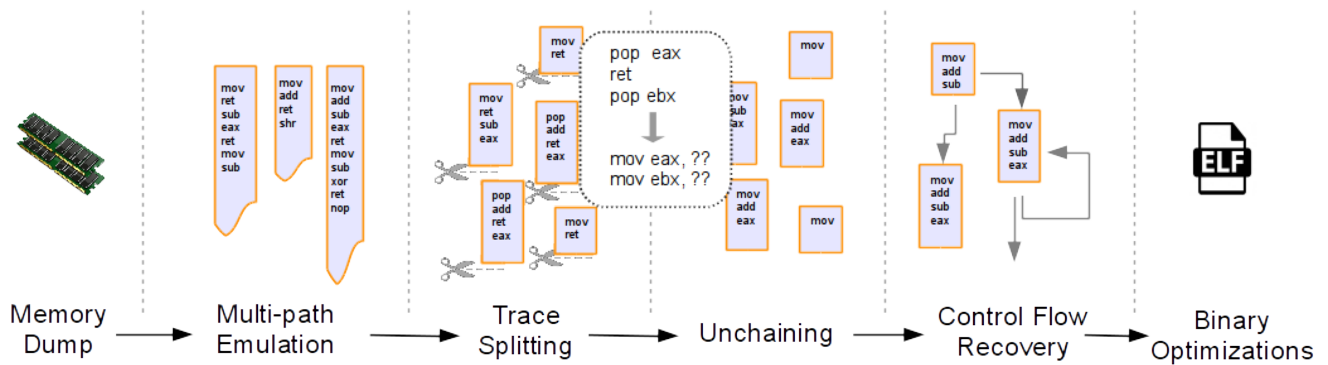

Figure 1: ROPMEMU Framework Architecture

as a Volatility [4] plugin to simplify the interaction with the memory dump by leveraging the Volatility APIs.

Execution Modes – At the end of the emulation a JSON trace is generated containing the CPU state for each assembly instruction. Depending on the complexity of the ROP chain, the size and the time required to generated this trace can be considerable.

For these reasons, we designed our emulator to support three execution modes: *full*, *incremental* and *replay*. In *full* emulation mode, the emulator executes the chain from scratch, starting from the provided entry point. The *replay* mode is completely based on an existing JSON trace and therefore it does not require any memory dump. This makes the rest of the analysis repeatable, and allows researchers to share traces without the need to transfer the content of the system memory (which may contain sensitive information and may be difficult to share for privacy reasons). Finally, the *incremental* mode is a combination of the previous two: it uses an input JSON trace (previously generated during a *full* emulation) and, once the last gadget in the trace is reached, it switches to *full* mode. This mode makes incremental analysis possible – a considerable advantage when dealing with very complex ROP chains.

Shadow Memory – The emulator initially reads the content of the memory from the memory dump. However, all write operations are redirected to a shadow memory area kept internally by the emulator. Subsequent read operations fetch data from the shadow memory (if the address has been previously written) or from the original memory image otherwise.

Chain Boundary – Although the analyst knows the starting address of the first ROP chain, it is unclear where it ends. This problem is very important because we do not want to keep emulating instructions beyond the end of a chain, thus polluting the analysis with unrelated code.

Our framework solves the problem by using a number of heuristics. To start with, the emulator detects large increments or decrements of the *stack pointer*. Typically, during the execution of a single ROP chain, these deltas are small. Based on this locality principle, it is possible to find the exact moment in which the chain under analysis is terminated. This simple rule needed to be refined to take into account long jumps that may occur inside a single, very long chain (see, for instance, the case described in Section 5). By including heuristics based on the length of a gadget, and excluding the detected function invocations, our prototype was able to correctly stop the emulation process at the last gadget in all our experiments. In case our heuristics fail, the analyst only needs to restart the emulator in *incremental mode* to continue the analysis of the chain from the point in which it was suspended.

Once the termination condition is triggered, the emulator stops and both the content of the shadow memory and the execution trace are saved to disk and are inspected to detect the presence of new ROP chains. If new stages are found, the emulator is re-started to analyze the next chain, and the process is repeated multiple times until all dynamically generated chains have been discovered and analyzed.

Syscalls and APIs – Complex ROP chains can invoke several system calls and library APIs, whose emulation is very complex (impossible in many cases) and goes beyond the scope of this paper. Our emulator recognizes when the execution is transferred to a system or API function, it saves its name in the trace, and then steps over its body to resume the emulation from the next gadget in the ROP chain.

This approach requires two types of information. First, the emulator needs to know the location and name of each API functions and system call routines. Luckily, this information can be easily retrieved by Volatility. Second, the emulator needs to know a valid output for each function. For instance, if the ROP chain allocates memory by calling `kmalloc`, the emulator needs to assign a valid (and not used) memory address to the function return value. Section 5 explains how we handled, on a case-by-case basis, more complex examples that require complex buffers or data structures.

Multi-Path Exploration – In the presence of long ROP chains with a complex control flow, a simple approach based on emulation is not enough to retrieve the entire ROP payload. The coverage is limited and takes into account only the executed branches – which often depend on the dummy return values generated by the emulator when

the chain invokes system functions. This point is crucial for the analysis, as researchers need the entire chain to understand all the features and components of the ROP code. A simple emulation approach would likely miss important parts and thus some core functionalities of the malicious code may remain hidden.

We address this problem by introducing a multi-path emulation. Although this approach has its roots in the well-known multi-path execution work proposed by Moser et al. [38], the original algorithm has been adapted to deal with ROP gadgets. In particular, our emulator is designed to recognize when the stack pointer is conditionally modified based on the content of the flag register. This pattern, however it is implemented, corresponds to a branch in the ROP chain. At the end of the emulation process, the JSON trace is analyzed to list all the branch points together with the value of the flags that was used in each of them by the emulator. The emulator is then re-started, this time providing an additional command-line parameter that specifies to complement the flag register at the required branch point, so that the execution can follow a different path. The exploration is terminated when all the branches have been explored. At the end, the analyst obtains several JSON traces containing different parts of the control flow graph.

However, in presence of loops in the ROP chain, the emulator could get trapped inside an endless execution path. The solution in this case is to keep track of the number of occurrences of the stack pointer during the execution of branch-related instructions. If this number is above a certain threshold (set to 10 in our experiments) the emulator automatically flips the flag bits to force the loop to end and explore the rest of the control flow graph. Similar heuristics are commonly applied to explore the behavior of malicious binaries.

4.3 Chain Splitting

The multi-path emulator generates a separate JSON trace file for each path in the ROP chain. The next step of our approach is in charge of splitting those traces, and removing duplicates parts that are in common between different traces. This part is divided in two steps. In the first, every trace is cut at each branch point, and a new block is generated and saved in a separate JSON trace. During this operation the framework also records additional information describing the relationships among the different blocks.

Since conditional branch instructions are based on the value of the flag register, our system uses tests on the flags content or `pushf` operations as indicators of a branch point. In particular, Chuck always pushes the flags on the stack to later retrieve them and isolate the `ZF` flag, whose values indicates which side of the branch needs to be taken. In the second pass, the chain splitter compares the individual blocks to detect overlapping footer instructions (i.e., gadgets in common at the end of different blocks) and isolate them in separate files.

The output of this phase is again a set of JSON trace file, this time not anymore associated to each individual path, but instead associated to each "basic block" in the chain. The chain splitter is implemented as a standalone Python script.

4.4 Unchaining Phase

This phase transforms each JSON file into a sequence of instructions in the target architecture. This is obtained by applying a number of simple transformations. First, all the `ret`, `call`, and unconditional `jmp` instructions are removed from the trace. Then, `mov` instructions are simplified by computing their operands. In fact, due to the fact that immediate values are stored on the stack, ROP chains often contain expressions involving several registers (e.g., `mov rax, [rsp+0x30]`) that at this stage are replaced with their actual value. Similarly, we transform `pop` into `mov` instructions, by fetching the required values from the corresponding location on the stack.

4.5 Control Flow Recovery

The input of the control flow recovery is the set of `x86` binary blobs generated by the unchaining phase, plus some additional information specifying the way these blocks were connected in the traces generated by the emulator. The goal of this phase is to replace all the code that belongs to the gadgets used to implement ROP branches with more traditional and more compact conditional jumps.

This step is not trivial because it is necessary to switch from the stack pointer domain to the instruction pointer one. At every branch point, we need to recreate from scratch the instruction pointer logic required to properly connect the two targets of a branch condition. In our case study, a simple conditional jump is implemented by 19 gadgets and 41 instructions. Our framework is able to recognize the condition and generate an equivalent assembly code in the instruction pointer domain. The 19 gadgets are translated into two assembly instructions: a conditional jump (in our case represented by either `jz` or `jnz`) and an unconditional jump (`jmp`).

The second task of the CFG recovery component is to detect and re-roll loops. ROP chains can contain both *return oriented loops* and *unrolled loops* that are programmatically generated when the chain is constructed. In the first case, the ROP instructions are used to conditionally repeat the same block of stack pointers, the same way a normal loop repeats the same sequence of EIP values. Unrolled loops repeat instead the same hard-coded sequence of gadgets over and over (typically as a result of a `for` loop in the C code that generated the ROP code), for a pre-determined number of times.

For instance, Chuck uses unrolled loops to copy the dynamically generated chains to their final memory location. In fact, in the original source code of the rootkit (written in C), this is implemented as a short `FOR` loop that generates the appropriate gadgets. In the rootkit itself, it becomes a long sequence containing five gadgets repeated thousands of times. A simplified version of the gadgets is presented in Figure 2.

The value of the `rdx` register is taken from the stack, and then copied to a memory location pointed by the register `rax`. Finally `rax` is incremented by eight (the value of `rdi` taken from the stack).

Our tool is able to automatically identify these recurrent patterns and replace the entire sequence of instructions with a more compact snippet of assembly code representing a real loop with the same semantics. The resulting code

```
pop rdx
mov [rax], rdx
pop rdi
add rdi, rax
mov rax, rdi

pop rdx
mov [rax], rdx
pop rdi
add rdi, rax
mov rax, rdi

...
```

Figure 2: Unrolled loop in the dispatcher Chain

is then wrapped withing a valid function prologue and epilogue and then embedded in a self-contained ELF file. It is important to note that it is not guaranteed that the file can actually be executed. If the original chain was part of a userspace shellcode, the ELF would probably contain all the instructions required to run the code. However, if the ROP chain is part of a kernel rootkit (as it is in our example), its code was initially designed to run in a very specific context in the kernel memory and therefore cannot be executed in a user-space program. However, our goal is just to generate a file that can be opened and analyzed by traditional reverse engineering tools such as IDA Pro.

4.6 Binary Optimization

The final step of our analysis consists of applying standard compiler transformations to optimize and simplify the generated code. For examples, dead code removal, simplifications of redundant mathematical operations, and global value numbering can greatly simplify the binary and makes it easier to understand for an analyst. However, these transformations have already been discussed in previous works [36, 60] and they are not the focus of our paper.

5. EVALUATION

In this section we describe the experiments we conducted to evaluate ROPMEMU on the most complex ROP-based payload publicly available. All the experiments have been performed on an Ubuntu 14.04 `x86-64` running Python 2.7 and Volatility 2.4. The virtual machine containing the rootkit has been provided kindly by the authors of Chuck and runs Ubuntu Server 13.04 64-bit with UEFI BIOS.

5.1 Chains Extraction

In the first experiment we tested the ability of the multipath emulator of `ropmemu` to correctly extract the persistent chain (the *copy chain*), and the two dynamically generated chains (the *dispatcher chain* and the *payload chain*). The last two are volatile and they are only created in memory when the right conditions are triggered. The results are summarized in Table 1.

ROPMEMU emulator was able to automatically retrieve the entire code of the three chains. The *copy chain* is the longest with 414,275 instructions, but it is composed of only a single basic block. The lack of a control flow logic makes this chain similar to a classic ROP shellcode, with the only difference of being composed of over 180K gadgets. This is a consequence of its main task: the creation and the copy in memory of the first dynamic component (*dispatcher chain*).

On the contrary, the *dispatcher chain* and *payload chain* have a lower number of gadgets but they have a more complex control flow graph. In particular, the *dispatcher chain* contains three branches and seven blocks of code. To recover the entire code, the emulator generated seven distinct JSON traces. The *payload chain* comprises instead 34 unique blocks and 26 branch points. This means the control flow graph has a more complex logic. Moreover, this chain invokes nine unique kernel functions (`find_get_pid`, `kstrtou16`, `kfree`, `__memcpy`, `printk`, `strncmp`, `strrchr`, `sys_getdents`, and `sys_read` – the last two hooked by the rootkit) for a total of 17 function calls over the different execution paths.

This experiment proves that `ropemu` can explore and dump complex ROP chains, which would be impossible to analyze manually. We believe these chains show the limits of the current malware analysis to cope with return oriented programming payloads and the effectiveness of the proposed framework.

5.2 Transformations

In this experiment we show the effect of the other phases of our analysis on the extracted ROP chains. In particular, since it is impossible to show the entire code, we present the effect of the transformations on the payload size. The results are summarized on table 2. As shown in the third column, the *unchain* pass reduces considerably the ROP chain size (on average 39%). The *CFG recovery* pass filters out the instructions implementing the conditional statements, translates the chain from the stack pointer domain to the instruction pointer one, and finally applies the loop compression step. These transformations reduce the *copy chain* to only 75 instructions (starting from over 414K). The *payload chain* is less affected by these transformations because it contained ROP loops instead of unrolled loop.

5.3 CFG Recovery

In the final experiment, we tested the ROPMEMU capability to retrieve and refine the control flow graph of a ROP chain as explained in section 4. Figure 3 and 4 illustrate the first phase on the *dispatcher* chain. In particular, Figure 3 represents the first version of the CFG, without any transformation. On Figure 4 we can observe the effects of the refinement steps. In these two figures every node represents a long stream of assembly instructions while the edges show the branch conditions.

The second step works on the binaries blobs and generates an ELF file. This ELF file connects all the blocks by leveraging the metadata information as explained in section 4 and the result can be inspected by ordinary reverse engineering tools. To test this functionality we opened the resulting file with IDA Pro. In Figure 5 we can observe the ELF representing the *copy chain* completely converted into the classic "EIP-based" programming paradigm. The graph is simple, there are no branches and the core func-

Chain	Instructions	Gadgets	Blocks	Branches	Functions	Calls
Copy	414,275	184,126	1	-	-	-
Dispatcher	63,515	28,874	7	3	1	5
Payload	6320	2913	34	26	9	17

Table 1: Statistics on the emulated ROP chains in terms of number of instructions, gadgets, basic blocks, branches, unique functions, and total number of invoked functions.

Chain	Initial State	Unchain Phase	CFG Recovery Phase
Copy	414,275	276,178	75
Dispatcher	63,515	40,499	16,332
Payload	6320	3331	2677

Table 2: Number of instructions in each chain after each analysis phase

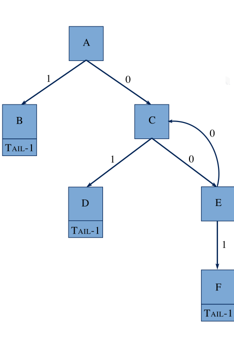

Figure 3: Dispatcher - Raw CFG

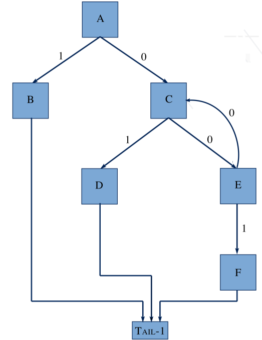

Figure 4: Dispatcher - Final CFG

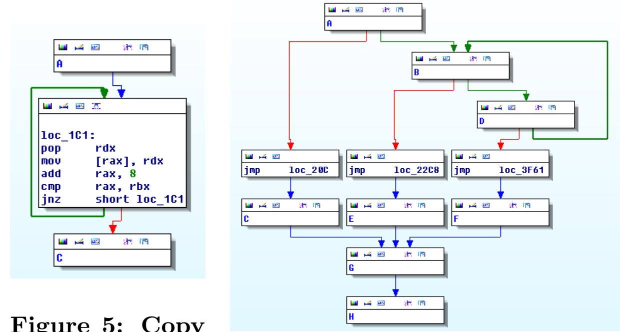

Figure 5: Copy - IDA Pro

Figure 6: Dispatcher - IDA Pro

tionalities are represented by the main loop highlighted in the picture. Figure 6 illustrates instead the *dispatcher chain* view on IDA Pro (for the sake of clarity every node is collapsed to generate a smaller picture). The graph is similar to Figure 4, with just few additional nodes due to how the basic blocks are connected together. As expected, the shape of the graph is the same.

The control flow graph of the *payload chain* comprises 34 blocks and is reported in Appendix. Overall, the graph has two main blocks: the `sys_read` block and `sys_getdents` one. In addition, the graph shows also the exit points (Q and C) and the loops (easy to identify from the backward edges). Even if the control flow graph contains few additional edges (due to the spurious optimizations introduced by the loop simplification), the information depicted in Figure **??** provides a quick but detailed overview about the behavior of the payload.

5.4 Results Assessment

An assessment system is fundamental to verify the results of the experiments. However, since it is not possible to run the final ELF to compare its behavior with the original rootkit, we decided to develop a number of individual verification tools.

First of all, we attached a debugger to the KVM virtual machine running the rootkit. For this task, we used GDB, extended with a set of Python plugins to extract information about the running rootkit and compare them with the results of our framework. The assessment framework is working on the live virtual machine while ROPMEMU is working on a memory dump. The GDB plugins collect the state of the guest VM (memory and CPU) and the trace of all the executed instructions. These information are then compared with the JSON traces generated by ROPMEMU to verify their accuracy.

We relied on this testing setup during development (to detect and patch bugs in our code) and at the end of the experiments to verify that both the emulation of individual instructions and the entire lists of instructions in each ROP chain was correctly reconstructed by ROPMEMU. It is important to note that the assessment framework is used only for debugging and cannot replace ROPMEMU as an analysis system. In fact, in the general case, the analyst does not have access to a virtual machine to emulate the compromised system, but only to its physical memory dump.

Using this setup we verified that all the results presented in this paper match those found using the live GDB analysis. Finally, we manually verified the control flow graph of the extracted chains by inspecting the source code of the ROP rootkit.

5.5 Performance

The performance of our system largely depends on the emulation phase. The emulator is built on top of Volatility and the time required to perform the multipath emulation is linear in the number of instructions and the number of paths to emulate. Our framework was able to emulate the entire *copy chain* in 52 minutes, while the *dispatcher* chain required 32 minutes to generate the three traces containing all the possible paths.

The performance of the *unchain* component depends instead on the size of the blocks to analyze. In our experiments, it ranged from the worst case of 61 minutes for *copy* chain (where everything is in a single huge basic block), to 3 minutes per block for the *dispatcher* chain (that is instead composed of smaller blocks). The *payload chain* traces have been generated on average in eight minutes while the *unchain* phase parsed each block in one minute.

Overall, the entire analysis of the rootkit from the emulation to the final ELF binary took four hours. All measurements have been recorded on a 16-Core Intel E5-2630 (2.3GHz) with 24GB RAM.

6. LIMITATIONS

As any other binary analysis tool, ROPMEMU has a number of intrinsic limitations. In particular, the proposed solutions combine two techniques: memory forensics and emulation. The first requires a physical memory dump acquired after the rootkit has been loaded in memory – and it is therefore prone to anti-acquisition techniques.

The ROP analysis relies instead on the emulator implementation. The main limitation of an emulation-based solutions is the accuracy of the emulator itself. In particular, this approach is prone to anti-emulation techniques specifically targeting instructions side effects. As a result, the current ROPMEMU prototype can be evaded by advanced ROP chains which implement ad-hoc anti-emulation techniques. However, 1) these techniques have never been observed so far in any ROP chain, and 2) similar limitations affect every existing binary analysis tool. For instance, IDA Pro can be easily evaded by malware implementing anti-disassembly tricks and malware sandboxes can be evaded by anti-emulation code. A path explosion problem is also common in approaches based on multi-path exploration or symbolic execution. However, these limitations do not make these tools useless – they just force the analyst to be more careful and manually disable evasion techniques before proceeding with the analysis. The same considerations apply to ROPMEMU.

Because of the intrinsic limitations of using an emulator on dynamically generated code, a perfect ROP reverse engineering system cannot exist. However, the solution presented in this paper is the first solution to allow the analysis of complex ROP code, and was designed and implemented to cope with the most sophisticated examples of this kind available today.

Finally, our current implementation is a research prototype and therefore lacks the robustness and completeness required to operate on arbitrary inputs.

7. RELATED WORK

Return Oriented Programming has been extensively studied in the scientific literature from several perspectives. However, very few works have presented novel techniques dedicated to the analysis of ROP chains and, due to space limitations, in this section we will focus only on this research.

Return oriented programming has been formalized by Shacham et al. [51], but the core ideas were already known in the underground community for years [39, 50]. The massive ROP adoption observed over the years has its roots in the protections introduced by the operating systems (namely *NX*). These protections significantly hindered the exploitation process and forced offensive researchers to devise *code reuse* attacks. ROP is without doubt the most common instance of these attacks and is widely adopted in modern exploits and, recently also in malware. The security community proposed many techniques to detect, prevent, or stop ROP, each with its own limitations and shortcomings. For example, threshold based defenses [45, 14] have been bypassed by using unexpected long sequences of gadgets [21]. Similarly, also control flow integrity (CFI) approaches have shown their limitations to combat ROP as described by Davi et al. [18]. Along the same line, ROP protections proposed by the industry – like the ones in the Microsoft's Enhanced Mitigation Experience Toolkit (EMET) [37] – have been bypassed by motivated researchers [5, 25, 48]. The most robust defense mechanisms are probably the one that apply at compile-time (e.g., [43]), but unfortunately it is hard to measure their effectiveness because the tools proposed so far are not available and, to the best of our knowledge, there are no public bypass for these techniques.

Moreover, offensive researchers have also proposed new *code reuse* attacks. Specifically, they showed it is possible to have ROP chains without any return [13]. Moreover, they proved the feasibility of jump oriented programming (JOP) [10] and sigreturn oriented programming (SROP) [11]. Finally, JIT-ROP [52] showed that it is possible to craft a ROP payload when fine-grained ASLR is in place. Although there are so many variations of *code reuse* attacks, ROP is still the most popular one. For this reason, in this paper, we created a comprehensive framework to cope with ROP chains.

In this direction, the first study has been conducted by Lu et al. [36]. The authors proposed DeRop, a tool to convert ROP payloads into normal shellcodes, so that their analysis can be performed by common malware analysis tools. However, the authors tested the effectiveness of their system only against standard exploits containing really simple ROP chains. In this paper, we adopt some of the transformations proposed by DeRop – which we complement by a number of novel techniques required to deal with the large and complex chains of a ROP rootkits. Our main goal is also more ambitious, as we want to achieve a full code coverage of the ROP payload, also in the presence of dynamically generated chains.

In another paper similar to our work, Yadegari et al. [60] proposes a generic approach to deobfuscate code, in which the authors considers ROP as a form of obfuscation. Their system is based on bit-level taint analysis that is applied to existing execution traces and can be used to deobfuscate the CFG. In addition, the paper also adopts transformations similar to the ones proposed by DeRop to handle

ROP payloads. Even though *Chuck* had already been released at the time, the authors claimed that no complex example of ROP chains was available, and they tested the system against small examples with a simple control flow logic. Moreover, the proposed system does not emulate the ROP chain and does not perform any code coverage. Instead, it focuses only on the simplification of existing execution traces.

Another interesting research direction focused on the problem of locating ROP chains in memory and potentially profile their behavior [55, 47, 33]. The first two solutions have been designed to analyze exploits targeting 32-bit user-land applications. Specifically, they both scan the program address space to identify the gadget and payload space and extract the entire chain. Recently, Kittel et al. [33] proposed a code pointer examination technique to isolate the main chains used by data-only malware and they tested their technique on a 64-bit system against *Chuck* [58].

ROPMEMU can adopt these techniques to identify persistent ROP chains in user- and kernel-space. In addition, the profiling phase proposed in these papers were quite simple, and they detect only persistent ROP chains. To overcome these limitations, we adopted an approach based on CPU and memory emulation. Finally, previous techniques do not work in presence of packed ROP chains [35] or chains which are dynamically generated at runtime [58].

8. CONCLUSION

In this paper we presented the first attempt to automate the analysis of complex code implemented entirely using ROP. In particular, we discussed the challenges to reverse engineer programs implemented using return oriented programming and we proposed a comprehensive framework to dissect, recostruct and simplify ROP chains. Finally, we tested the framework with the most complex case proposed so far: a persistent ROP rootkit. The proposed framework is motivated by the lack of methodologies and tools to analyze in depth ROP payloads of increasing complexity.

The solution we described is ROPMEMU, and comprises a combination of Volatility plugins and additional standalone scripts. Our framework can extract the entire code of both persistent and dynamically generated ROP chains through a novel multipath emulation approach, simplify the output traces, extract the control flow graph and generate a final binary representing a cleaner version of the original ROP chain. The analysts can then operate on this binary with traditional reversing engineering tools like IDA Pro. Overall, the results of our experiments confirm the accuracy and effectiveness of ROPMEMU to analyze advanced ROP chains.

9. REFERENCES

[1] Apple code signing. https://developer.apple.com/library/mac/documentation/Security/Conceptual/CodeSigningGuide/Introduction/Introduction.html.

[2] Microsoft Code Signing. https://msdn.microsoft.com/en-us/library/ms537361.aspx.

[3] Microsoft Driver Signing. https://msdn.microsoft.com/en-us/library/windows/hardware/ff544865%28v=vs.85%29.aspx.

[4] Volatility framework: Volatile memory artifact extraction utility framework. http://www.volatilityfoundation.org/.

[5] Aaron Portnoy. Bypassing All Of The Things. https://www.exodusintel.com/files/Aaron_Portnoy-Bypassing_All_Of_The_Things.pdf.

[6] Aurelien Wailly. nROP. http://aurelien.wail.ly/nrop.

[7] Axel Souchet. rp. https://github.com/0vercl0k/rp.

[8] F. Bellard. Qemu a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*. USENIX Association, 2005.

[9] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazières, and D. Boneh. Hacking blind. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, 2014.

[10] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang. Jump-oriented programming: A new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '11, 2011.

[11] E. Bosman and H. Bos. We got signal. a return to portable exploits. In *Security & Privacy*, 2014.

[12] E. Buchanan, R. Roemer, H. Shacham, and S. Savage. When good instructions go bad: Generalizing return-oriented programming to RISC. In P. Syverson and S. Jha, editors, *Proceedings of CCS 2008*, Oct. 2008.

[13] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy. Return-oriented programming without returns. In A. Keromytis and V. Shmatikov, editors, *Proceedings of CCS 2010*, pages 559–72. ACM Press, Oct. 2010.

[14] Y. Cheng, Z. Zhou, M. Yu, X. Ding, and R. H. Deng. Ropecker: A generic and practical approach for defending against ROP attacks. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*, 2014.

[15] V. Chipounov, V. Georgescu, C. Zamfir, and G. C. Selective symbolic execution. In *In Workshop on Hot Topics in Dependable Systems*, 2009.

[16] Corelan. Mona. https://github.com/corelan/mona.

[17] Dan Rosenberg. SMEP: What is It, and How to Beat It on Linux. http://vulnfactory.org/blog/2011/06/05/smep-what-is-it-and-how-to-beat-it-on-linux/.

[18] L. Davi, A.-R. Sadeghi, D. Lehmann, and F. Monrose. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 401–416, San Diego, CA, Aug. 2014. USENIX Association.

[19] Dino Dai Zovi. Hardware Virtualization Rootkits. https://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Zovi.pdf.

[20] S. Embleton, S. Sparks, and C. Zou. Smm rootkits: A new breed of os independent malware. In *Proceedings of the 4th International Conference on Security and Privacy in Communication Netowrks*, SecureComm '08, 2008.

[21] E. Göktaş, E. Athanasopoulos, M. Polychronakis, H. Bos, and G. Portokalidis. Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard. In *USENIX Security 2014*.

[22] M. Graziano, A. Lanzi, and D. Balzarotti. Hypervisor Memory Forensics. In *Symposium on Research in Attacks, Intrusion, and Defenses (RAID)*, RAID 13. Springer, October 2013.

[23] R. Hund, T. Holz, and F. C. Freiling. Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms. In *Proceedings of the 18th Conference on USENIX Security*

Symposium, 2009.

[24] James T. Bennett - FireEye. The Number of the Beast. https://www.fireeye.com/blog/threat-research/2013/02/the-number-of-the-beast.html.

[25] Jared DeMott - Bromium Labs. Bypassing EMET 4.1. https://bromiumlabs.files.wordpress.com/2014/02/bypassing-emet-4-1.pdf.

[26] Jean - Sogeti ESEC Lab. Analysis of the jailbreakme v3 font exploit. http://esec-lab.sogeti.com/posts/2011/07/16/analysis-of-the-jailbreakme-v3-font-exploit.html.

[27] Joanna Rutkowska. Bluepill. http://web.archive.org/web/20080418123748/http://www.bluepillproject.org/.

[28] Joe Damato. A closer look at a recent privilege escalation bug in Linux (CVE-2013-2094). http://timetobleed.com/a-closer-look-at-a-recent-privilege-escalation-bug-in-linux-cve-2013-2094/.

[29] Jonathan Corbet. Supervisor mode access prevention. http://lwn.net/Articles/517475/.

[30] Jonathan Salwan. ROPgadget - Gadgets finder and auto-roper. http://shell-storm.org/project/ROPgadget/.

[31] Kees Cook. Kernel address space layout randomization. http://selinuxproject.org/~jmorris/lss2013_slides/cook_kaslr.pdf.

[32] S. T. King, P. M. Chen, Y.-M. Wang, C. Verbowski, H. J.Wang, and J. R. Lorch. Subvirt: Implementing malware with virtual machines. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, Oakland, CA, May 2006.

[33] T. Kittel, S. Vogl, J. Kisch, and C. Eckert. Counteracting data-only malware with code pointer examination. In *18th International Symposium on Research in Attacks, Intrusions and Defenses*, Nov. 2015.

[34] T. Kornau. Return oriented programming for the arm architecture. In *Master's Thesis - Ruhr-Universitat Bochum*, 2009.

[35] K. Lu, D. Zou, W. Wen, and D. Gao. Packed, printable, and polymorphic return-oriented programming. In *Recent Advances in Intrusion Detection - 14th International Symposium, RAID 2011*.

[36] K. Lu, D. Zou, W. Wen, and D. Gao. derop: Removing return-oriented programming from malware. In *Proceedings of the 27th Annual Computer Security Applications Conference*, 2011.

[37] Microsoft. Enhanced Mitigation Experience Toolkit. https://technet.microsoft.com/en-us/security/jj653751.

[38] A. Moser, C. Kruegel, and E. Kirda. Exploring multiple execution paths for malware analysis. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, SP '07, 2007.

[39] Nergal. Advanced return-into-lib(c) exploits. http://phrack.org/issues/58/4.html.

[40] Nguyen Anh Quynh. OptiROP: the art of hunting ROP gadgets. In *BlackHat 2013*.

[41] Nguyen Anh Quynh and Dang Hoang Vu. Unicorn - The ultimate CPU emulator. http://www.unicorn-engine.org/.

[42] Nicolas Economou - CoreSecurity. Agafi (Advanced Gadget Finder). http://www.coresecurity.com/corelabs-research/publications/agafi-advanced-gadget-finder.

[43] K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda. G-free: defeating return-oriented programming through gadget-less binaries. In *Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC)*, ACSAC '10, pages 49–58. ACM, December 2010.

[44] pakt. ropc. https://gdtr.wordpress.com/2013/12/13/ropc-turing-complete-rop-compiler-part-1/.

[45] V. Pappas, M. Polychronakis, and A. D. Keromytis. Transparent rop exploit mitigation using indirect branch tracing. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, Washington, D.C., 2013.

[46] Patrick Stewin and Iurii Bystrov. Understanding DMA Malware. In *Proceedings of the 9th Conference on Detection of Intrusions and Malware & Vulnerability Assessment*.

[47] M. Polychronakis and A. D. Keromytis. Rop payload detection using speculative code execution. *2013 8th International Conference on Malicious and Unwanted Software: "The Americas" (MALWARE)*, 2011.

[48] Rene Freingruber. EMET 5.1 - Armor or Curtain? https://prezi.com/tnqeqis3vhum/zeronights-2014-emet-51-armor-or-curtain/.

[49] sashs. ropper. https://scoding.de/ropper/.

[50] Sebastian Krahmer. x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique. http://users.suse.com/~krahmer/no-nx.pdf.

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

[52] 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*, May 2013.

[53] Solar Designer. Openwall. http://www.openwall.com/linux/README.shtml.

[54] spender. UDEREF. https://grsecurity.net/~spender/uderef.txt.

[55] B. Stancill, K. Z. Snow, N. Otterness, F. Monrose, L. Davi, and A.-R. Sadeghi. Check my profile: Leveraging static analysis for fast and accurate detection of rop gadgets. In *16th Research in Attacks, Intrusions and Defenses (RAID) Symposium*, Oct. 2013.

[56] The PaX Team. Pageexec. https://pax.grsecurity.net/docs/pageexec.txt.

[57] The PaX Team. Segmexec. https://pax.grsecurity.net/docs/segmexec.txt.

[58] S. Vogl, J. Pfoh, T. Kittel, and C. Eckert. Persistent data-only malware: Function hooks without code. In *Proceedings of the 21th Annual Network & Distributed System Security Symposium (NDSS)*, Feb. 2014.

[59] Websense Security Labs. Technical Analysis on iPhone Jailbreaking. http://community.websense.com/blogs/securitylabs/archive/2010/08/06/technical-analysis-on-iphone-jailbreaking.aspx.

[60] B. Yadegari, B. Johannesmeyer, B. Whitely, and S. Debray. A Generic Approach to Automatic Deobfuscation of Executable Code. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2015.

A. Payload Chain Control Flow Graph