deRop: Removing Return-Oriented Programming from Malware

Kangjie Lu
Peking University, China
Singapore Management University, Singapore
kangjielu@pku.edu.cn

Dabi Zou
Singapore Management University, Singapore
zoudabi@gmail.com

Weiping Wen
Peking University, China
weipingwen@ss.pku.edu.cn

Debin Gao
Singapore Management University, Singapore
dbgao@smu.edu.sg

ABSTRACT

Over the last few years, malware analysis has been one of the hottest areas in security research. Many techniques and tools have been developed to assist in automatic analysis of malware. This ranges from basic tools like disassemblers and decompilers, to static and dynamic tools that analyze malware behaviors, to automatic malware clustering and classification techniques, to virtualization technologies to assist malware analysis, to signature- and anomaly-based malware detection, and many others. However, most of these techniques and tools would not work on new attacking techniques, e.g., attacks that use return-oriented programming (ROP).

In this paper, we look into the possibility of enabling existing defense technologies designed for normal malware to cope with malware using return-oriented programming. We discuss difficulties in removing ROP from malware, and design and implement an automatic converter, called deRop, that converts an ROP exploit into shellcode that is semantically equivalent with the original ROP exploit but does not use ROP, which could then be analyzed by existing malware defense technologies. We apply deRop on four real ROP malwares and demonstrate success in using deRop for the automatic conversion. We further discuss applicability and limitations of deRop.

Keywords: return-oriented programming, malware analysis

1. INTRODUCTION

Malware analysis has been one of the hottest areas in security research in the last 10 to 20 years. Many techniques and tools have been introduced and built to automatically analyze and defend against malware. This ranges from basic tools like disassemblers (e.g., IDAPro[1], OllyDbg[2]) and decompilers [11], to static [10, 16] and dynamic tools [4, 31] to analyze malware behaviors, to automatic malware clustering [3, 18] and classification techniques [22, 1], to virtualization techniques for analyzing malware [13], to signature- and anomaly-base malware detection [20, 27, 17, 30], and many others.

Although some of these techniques are designed to be able to deal with zero-day malware, e.g., anomaly-based detection, the vast majority of them are based on understanding of the existing malware techniques. Even anomaly-based detection might not work when the technology used by malware changes (instead of new malware exploiting an unknown vulnerability). Return-oriented programming is an example of such new malware technology.

Return-oriented programming (ROP) [26, 28] and its variations [7, 6, 14, 19, 21, 8, 5] have been shown to be able to perform arbitrary computation without executing injected code. They execute machine instructions immediately prior to return (or return-like [7]) instructions within the existing program or library code. Since ROP does not execute any injected code, it circumvents most measures that try to prevent the execution of instructions from user-controlled memory, e.g., the W $\oplus$ X protection mechanism [25].

One solution to this problem is to design patches to all existing malware defense technologies so that they can cope with return-oriented programming. Although not entirely impossible, it is definitely not scalable due to the huge amount of existing defense work. Even if it can be done, it will become a nightmare when yet another new malware technology emerges.

Instead, we propose to automatically remove return-oriented programming from a piece of malware before it is sent for further analysis by existing malware defense tools. In this paper, we design and implement deRop, an automatic tool to convert shellcode[3] using ROP into one that does not use

[1]http://www.hex-rays.com/idapro
[2]http://www.ollydbg.de
[3]The payload of an ROP attack usually contains many gadgets of addresses, constants, and junks. There is usually no instruction or code in an ROP payload, and strictly speaking we should not call it shellcode. Here we still use the word shellcode to refer to payload of an attack in general, should it use ROP or not.

ACSAC '11 Dec. 5-9, 2011, Orlando, Florida USA

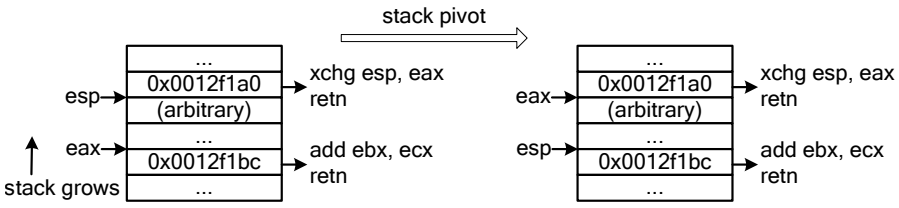

Figure 1: Special use of esp

ROP, while preserving the semantics of the original malware shellcode. Note that here we focus on enabling existing malware analysis tools to work on the output of deRop. deRop does not perform this analysis on the malware directly.

Removing return-oriented programming from malware is non-trivial. Among the many difficulties (discussed in Section 3), the use of register `esp` in ROP shellcode is one of the most important ones. Figure 1 shows an example in which `esp` is used as a stack pivot [2] to chain the execution of gadgets (address words pointing to "hidden" instructions and data) in ROP. The stack layouts on the left and right show the content of `esp` and `eax` when instructions pointed to by the two gadgets start getting executed, respectively. It is relatively easy to find out all the "hidden" instructions to be executed pointed to by the gadgets, however, deRop cannot simply take these instructions as the output of the conversion. For example, the instruction `<xchg esp, eax>` pointed to by one of the gadgets is used to update the value of `esp` so that it points to the next gadget to be executed. In this sense, `esp` in ROP serves the functionality of `eip`. `<xchg esp, eax>`, therefore, is not really part of the instructions this attack tries to execute, and should not be part of the output of deRop.

Another difficulty is to be able to find out the next gadget (and the corresponding "hidden" instructions). Intuitively, dynamic analysis could help solve this relatively easily. However, it comes with a price of having to execute the malware in the analysis, which is a big disadvantage considering the security of the analysis platform. Therefore, one of the criteria we have in designing deRop is to use static analysis to simulate the updates to `esp` as much as possible, and only resolve to dynamic analysis when needed.

We design and implement deRop to automatically convert shellcode using ROP into one that does not use ROP. deRop relies heavily on static analysis without executing the malware, with a minimum of dynamic analysis only to find the location of the first gadget in the malware payload and to find the initial value of `esp`, which are hard to obtain reliably with static analysis alone. We apply deRop on four real ROP malwares and manually verify that the output of deRop does not make use of ROP, and is semantically equivalent to the original ROP malware. We also discuss applicability of deRop and its limitations.

2. BACKGROUND AND RELATED WORK

2.1 ROP and its variations

Shacham et al. propose Return-Oriented Programming (ROP) in 2007 [28]. ROP uses a large number of instruction sequences from the original program and the libraries, and chains these instruction sequences ending with `ret` together to perform arbitrary computation. ROP is also ported to other platforms such as SPARC [6], ARM [21], Harvard [14], and voting machines [8]. Besides that, Hund et al. [19] make use of ROP to propose Return-Oriented Rootkits, which can bypass kernel code integrity protection mechanisms. Li et al. [23] propose the corresponding "return-less" kernels to defeat Return-Oriented Rootkits. However, ROP is recently extended to use return-less gadgets to achieve the same effect [7, 5].

2.2 Defense of ROP

With the development of ROP, some researches are seeking ways to detect and prevent ROP attacks. One direction is to make use of the characteristics of ROP, e.g., short pieces of instructions ended by a `ret`. Davi et al. [12] and Chen et al. [9] make use of dynamic binary instrumentation frameworks to instrument program code. When the number of consecutive sequences of five or fewer instructions ending in a return reaches a threshold, it will trigger an alarm. Another direction is to look for violations of last-in, first-out invariants of the stack data structure that call and return instructions usually maintain in normal benign programs. Buchanan et al. [6] suggest to maintain a shadow return-address stack, which can be used to defend against ROP. Francillon et al. [15] implement a shadow return-address stack in hardware for an Atmel AVR microcontroller such that only call and return instructions can modify the return-address stack.

Davi et al. [12] claim that it is possible to extend their ROP defender with a frequency measurement unit to detect attacks with return-less ROP. The idea is that pop-jump sequences are uncommon in ordinary programs, while return-less ROP [7] invokes such a sequence after each instruction sequence.

Most recently, Onarlioglu et al. [24] propose G-Free, which is a compiler-based approach to defeat against any possible form of ROP. Their solution is to eliminate all unaligned free-branch instructions inside a binary executable, and to prevent aligned free-branch instructions from being misused.

Unlike these defense mechanisms, deRop does not try to detect or stop ROP. Instead, it converts any (attack) program that uses ROP into one that does not. This has huge implication on the applicability of existing malware analysis tools and they could now be used to analyze ROP attacks.

2.3 Existing malware analysis tools

There have been many malware analysis tools proposed. Some examples include disassemblers (e.g. IDAPro, OllyDbg) and decompilers (e.g., HexRays decompiler[4]), static [10, 16] and dynamic tools [4, 31] to analyze malware behav-

[4]http://hex-rays.com/decompiler.shtml

iors, automatic malware clustering [3, 18] and classification techniques [22, 1], virtualization techniques for analyzing malware [13], signature- and anomaly-base malware detection [20, 27, 17, 30], and many others.

Most, if not all, of these techniques were proposed to analyze traditional malware without considering ROP. In fact, most of them were proposed before ROP was introduced. Traditional malware code (usually in form of shellcode) is very different from ROP in that ROP shellcode consists of addresses, constants and junks but not instructions. Therefore, direct applications of such tools on ROP code will likely fail.

In this paper, we are not trying to propose a new design of these malware analysis tools so that they could work on ROP. There are too many such tools proposed and it is not practical or scalable to patch all of them. Instead, we propose an automatic converter, called deRop, to convert ROP shellcode into its semantically equivalent non-ROP shellcode so that any of the existing malware analysis tools can analyze it.

3. DIFFICULTIES

In Section 1, we briefly discuss why existing malware analysis tools are not able to analyze ROP shellcode effectively. In this section, we detail the difficulties involved in designing deRop, which takes input some shellcode that uses ROP and outputs non-ROP shellcode to be analyzed by existing malware analysis tools.

3.1 Locations of gadgets

The payload of an ROP attack would usually leave some junk before the first gadget, e.g., if the exploit is via a buffer overflow. Therefore, deRop needs to find a way of locating the first gadget in the ROP shellcode. In some SEH exploits, there are even junks between the first and the second gadget since where `esp` points to might not follow the location of the return address. Although deRop tries to use static analysis as much as possible and tries to avoid using dynamic analysis (see discussion in Section 1), locating the first two gadgets in an ROP attack using static analysis turns out to be unreliable, and therefore we customize a debugger to do it instead. Note that the debugger never runs any malicious code.

3.2 Keeping track of esp

As pointed out in Section 1, `esp` in ROP shellcode has a special usage as a global state pointer whose function is to get the address of the next group of instructions, just like `eip` in normal programs. We need to keep track on the value of `esp`, e.g., to locate the next gadget to be used in the execution. This is non-trivial especially in sophisticated ROP shellcode that has conditional branches.

3.3 Stack layout and constant relocation

ROP and non-ROP code load constants into registers in very different ways. Traditional shellcode usually uses `<mov reg, imm>`, while ROP shellcode usually pre-arranges the constant on the stack, and then uses `<pop reg>` to load the constant into a register. Therefore, deRop needs to relocate the constants in its transformation, and the input and output of deRop has very different stack layout. A mapping between these constants in the input ROP shellcode and the output non-ROP shellcode is developed to keep track on them.

3.4 Function calls

Some gadgets in ROP are used to call functions. This is usually achieved by stack pivot instructions or `<pushad>` followed by `ret`. One may argue that no special treatment is needed for function calls as we can simply inline their body to remove ROP and preserve semantics at the same time. However, the objective of deRop is to enable other malware analyzers to be able to analyze the resulting non-ROP shellcode, therefore being able to recognize these function calls and to conform to normal function call conventions is important. Difficulties here include identifying gadgets that perform function calls and parameter usage (e.g., parameters being constants or pointers) for the call.

3.5 Loops

Loops in ROP are usually implemented using stack pivot to perform conditional jumps. The difficulty here inlcudes identifying loops as well as finding out the actual condition of the loop. deRop uses some heuristics to handle these difficulties, which turn out to be effective in our experiments.

4. DESIGN AND IMPLEMENTATION OF deRop

In this section, we detail the design and implementation of deRop. We first give an overview of the design in Section 4.1, and then present a running example to aid the explanation (see Section 4.2). Section 4.3, Section 4.4, and Section 4.5 describe the three main steps in the design of deRop. Finally, we briefly describe how deRop is implemented in Section 4.6.

4.1 Overview

Although converting ROP shellcode to non-ROP shellcode to be analyzed by existing malware analysis tools is non-trivial (see difficulties discussed in Section 3), we design and implement deRop, an automatic tool to remove return-oriented programming from a piece of code. Figure 2 shows an overview of the design of deRop.

deRop first uses a customized debugger to find out the locations of of the first two gadgets in the input ROP shellcode. This is the only component in deRop that uses dynamic analysis, and our special design of the debugger makes sure that potentially harmful instructions in the attack code do not get executed (we don't even take the ROP shellcode as input to the vulnerable program).

After finding out the locations of the first two gadgets in the ROP shellcode, deRop uses a loop to analyze each individual gadget. This step of the analysis employs only static analysis techniques to simulate the execution of each gadget, and then output instructions that do not use ROP. During the simulation, deRop keeps track of many important information including the register values, mapping between addresses of instructions and data in the ROP input shellcode and non-ROP output shellcode. Finally, deRop performs a post-processing to improve the output shellcode so that it can be readily analyzed by existing malware analyzers.

4.2 A running example

Due to the complexity of certain parts of deRop, we find it easier to understand if we explain it with a simple example.

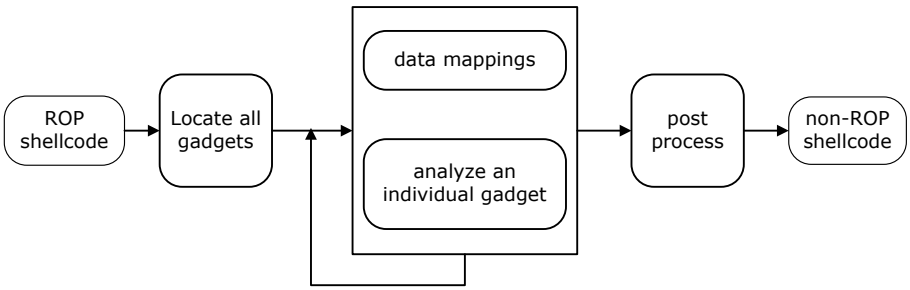

Figure 2: Overview of deRop

We use a real ROP exploit from `exploit-db`[5] which exploits CoolPlayer 2.18[6] with DEP (Data Execution Prevention) . The transferred results are shown in Table 1.

We choose this ROP exploit because it is simple and exhibits a typical structure of existing ROP exploits in that they first use ROP to perform a crucial step in the attack, and then trigger a piece of traditional (non-ROP) shellcode to perform whatever the attacker wants to achieve. In this particular example, the ROP part uses the function `SetProcessDEPPolicy()` to disable DEP for the process, so that the traditional shellcode in the data segment can be executed.

The first three columns of Table 1 show the structure of this exploit, where the first part contains junks to perform buffer overflow, the second part uses 7 gadgets (the first gadget doesn't play a role in this attack) to perform ROP, and the last part contains the traditional non-ROP shellcode to be executed. Gadget 2 to 6 are used to set up register values to be pushed on the stack by gadget 7 using `pushad`. The subsequent `ret` instruction causes the process to look up a return address on the stack (which is located where `esi` is pushed) and to transfer control to that location.

The stack layout right after `pushad` is executed is shown in Figure 3. Gadget 6 and 5 set `esi` and `edi` as pointers to `ret` instructions, which means that when control is transferred to them, they do nothing but moves on. `ebp` is set to be the address of `SetProcessDEPPolicy()` by gadget 4, which, when called, uses the value of `ebx` as its parameter. Gadget 2 and 3 set `ebx` to be 0, so that when `SetProcessDEPPolicy(0)` is called, data execution prevention is disabled.

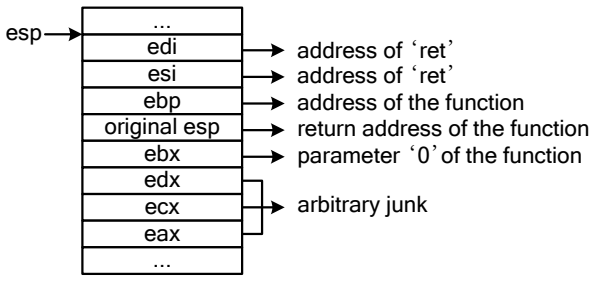

Figure 3: Stack layout right after pushad is executed in our example

[5] `http://www.exploit-db.com/`

[6] `http://www.exploit-db.com/exploits/15895/`

4.3 Locating gadgets

The first challenge we face is to locate the gadgets in an ROP exploit code. As discussed in Section 3 and shown in Table 1, this is non-trivial since the ROP exploit may contain junks before the first gadget. In some cases, there are even junks between the first and the second gadgets. To reliably find out the locations of the first and the second gadgets, we design a debugger to dynamically monitor the execution of the vulnerable program.

We stress that this is the only component of deRop that does dynamic analysis, and deRop does it in such a way that potentially harmful instructions in the attack code are never executed.

We first prepare a buffer that is of the same length of the ROP exploit as shown in Figure 4 to be used to exploit the vulnerable program. There are two index numbers in this buffer — a byte-index number and a word-index number denoted $\mathsf{idx_b}(x)$ and $\mathsf{idx_w}(x)$, respectively, where x is a 4-byte word in the buffer. $\mathsf{idx_b}(x)$ (2 bits long) appears in each byte of x, and is used to find the alignment offset in case the gadgets in the exploit are not aligned at multiples of 4-byte words. $\mathsf{idx_w}(x)$ (24 bits long) is used to tell which 4-byte word the first gadget is located[7]. We execute the vulnerable program, use this buffer to exploit it, and observe the values of `eip` and `esp` when the exploit succeeds. The value of `eip` tells us the size of the junk before the first gadget, while the value of `esp` tells us the size of the junk between the first two gadgets (indirectly).

When testing with the running example shown in Table 1, our debugger finds that $\mathsf{idx_w}(\mathsf{eip}) = 55$, $\mathsf{idx_w}(\mathsf{esp}) = 56$, and $\mathsf{idx_b}(\mathsf{eip}) = \mathsf{idx_b}(\mathsf{esp}) = 0$. This means that location of the first gadget is at an offset of $55 \times 4 + 0 = 220$ bytes, and the value of `esp` is at an offset of $56 \times 4 + 0 = 224$ bytes before the first gadget executes.

Now we have managed to find out the location of the first gadget without executing any potentially harmful instructions in the ROP exploit. The next is to find out the location of the second gadget. As explained in Section 1, `esp` plays the role of `eip` in ROP. Therefore, all we need is to find out the value of `esp` *after* the execution of (instructions pointed to by) the first gadget. This can be achieved by analyzing how such instructions change the value of `esp`, since we already know the value of `esp` *before* the execution of the first gadget.

In our running example, the first gadget points to the instruction `pop ecx` (see Table 1), which added 4 to the

[7] Note that the 8th bit is 1 instead of 0 in order to make sure that none of the bytes in the buffer has a value of 0.

Table 1: A running example

ROP shellcode			Resulting non-ROP shellcode	
Gadget #	Payload	Instructions	Initial results	Post-process
	"0x41" ×220	junk		
1	0x7c9fb028 0x42424242	pop ecx retn	mov ecx,[0x12f1ab]	mov ecx,0x42424242
2	0x7c9eaddd 0xffffffff	pop ebx retn	mov ebx,[0x12f1af]	mov ebx,0xffffffff
3	0x77c127e1	inc ebx retn	inc ebx	inc ebx
4	0x7c9ea67b 0x7c8922a4	pop ebp retn	mov ebp, [0x12f1b3]	mov ebp, 0x7c8922a4
5	0x7c9eeb47 0x7c9c1508	pop edi retn	mov edi, [0x12f1b7]	mov edi, 0x7c9c1508
6	0x7c9c204c 0x7c9c2051	pop esi retn	mov esi, [0x12f1bb]	mov esi, 0x7c9c2051
7	0x7ca11073	pushad retn	mov esp, 0x12f1bf pushad pop edi jmp edi	mov esp, 0x12f1ab pushad pop edi jmp edi
	"0x90" ×10 240 bytes	nop shellcode	0x42424242 0xffffffff 0x7c8922a4 0x7c9c1508 0x7c9c2051	nil

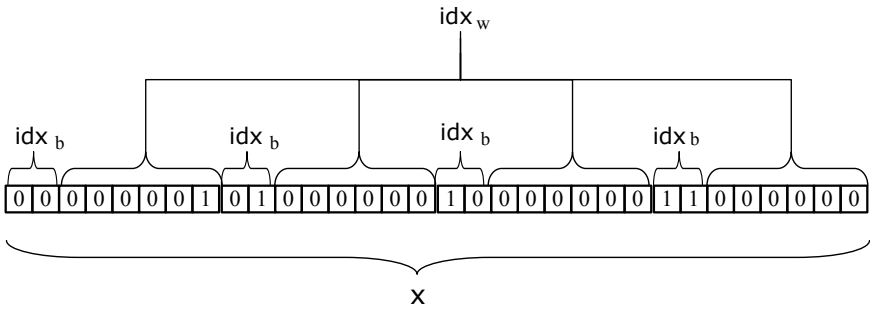

Figure 4: Debugger payload format

value of `esp`. Therefore, we know that `esp` is at an offset of 224 + 4 = 228 bytes, which will also be the location of the second gadget. Note that this means that there is no junk between the first two gadgets, since the size of the first gadget is 8, which is exactly the difference between the offsets of the first two gadgets.

Locating subsequent gadgets follows the same idea by monitoring the value of `esp`. Note that dynamic analysis is not needed in locating subsequent gadgets, since all we need is the change to `esp` as the result of executing (instructions pointed to by) the gadgets, which can be obtained by static analysis.

4.4 Removing ROP

As discussed in Section 3, removing ROP from an exploit is not as simple as chaining all instructions pointed to by the gadgets. We need to analyze the instructions pointed to by each gadget one by one, and monitor 1) address of each instruction, 2) values of registers, and 3) addresses of memory accesses. This is very similar to simulating the execution of the ROP shellcode, although we perform all this statically while preserving the semantics of the original ROP shellcode. Next, we detail how deRop deals with various types of instructions pointed to by each gadget.

4.4.1 *Push and pop instructions*

`push` and `pop` instructions are used very often in ROP. In particular, they are used together to transfer the value of one register to another, the latter typically being `esp`. ROP also usually prepares constants in its payload, which are on the stack during the initial exploit, and pops these constants from the stack to various registers. A few examples are shown in Table 1 in gadgets 1, 2, 4, 5, and 6.

Table 2 shows how deRop deals with `push` and `pop` instructions. In cases where `push` and `pop` are used together (see the second row of Table 2), we use a `mov` instruction to replace them, where `addr1` is the location of the data that is used to initialize `eax`. Note that here we do not replace them with `<mov ecx, eax>`. The reason is that `pop` does not clear the data (of `eax`) on the stack, and therefore such data might be used again later in the ROP exploit. When `pop` is used alone to assign a constant prepared on the stack to a register, we replace `pop` with the corresponding `mov` instruction as shown in Table 2.

An important requirement to enable this transformation from ROP instructions to non-ROP instructions is that we keep track on the location of data in memory. For example, `addr1` is the address of the data that is stored in `eax`, and `addr2` is the address of the data the ROP originally prepares

Table 2: Push and pop transferring example

ROP instructions	non-ROP instructions
`push eax; pop ecx`	`mov ecx, [addr1]`
`pop ecx`	`mov ecx, [addr2]`

on the stack. deRop keeps track on the addresses of them in the process of analyzing the instructions.

Another challenge is that the memory layout of the ROP exploit and the resulting non-ROP exploit might not be the same. In our example shown in Table 1, the gadgets in the payload contains constants, which is unique in ROP. Non-ROP shellcode does not usually mix code and data together in such a special way (it may do so with immediates as operands of instructions, which will be discussed in Section 4.5). Therefore, deRop relocates such data to another location in memory, e.g., see the last 20 bytes of the initial results presented in Table 1.

4.4.2 Memory related instructions

deRop deals with memory related instructions by setting aside a memory region in the output of deRop to store constants, and keeping a mapping between the corresponding constants in ROP and non-ROP shellcode. As shown in Table 1, the non-ROP shellcode reserves the last few bytes to store 5 constants, which were originally located separately in 5 different gadgets.

Table 3 shows two examples of instructions that involve memory access. In the first example, the address `0x1234` is outside the range of the ROP exploit code. For these type of addresses, deRop keeps them unchanged. While in the second example, the address `0x12345678` falls in the range of the original ROP shellcode, and deRop replaces it with a new memory location at the end of the resulting non-ROP shellcode (`addr3`). deRop updates its mapping of the two locations of this data so that future access of the same data can be handled properly.

Table 3: Memory related instruction transferring example

ROP instructions	non-ROP instructions
`add ecx,[0x1234]`	`add ecx,[0x1234]`
`add [0x12345678], ecx`	`add [addr3], ecx`

4.4.3 Stack pivot instructions

ROP shellcode typically uses stack pivot instructions to set the value of `esp`. This is critical in ROP since `esp` has a special usage as a global state pointer (just like `eip`) to get the address of the next gadget.

deRop monitors all the stack pivot instructions (e.g., `<add esp, 8>`; `<xchg esp, eax>`) to monitor the value of `esp`. Note that deRop simply needs to monitor it so that it knows where the next gadget is, while there does not need to have any corresponding instructions in the output of deRop, see the first example in Table 4.

Table 4: Stack pivot transferring example

ROP instructions	non-ROP instructions
`add esp, 20`	nil
`xchg eax, esp`	`cmp eax, addr4; jz offset1`

However, there is a special case we need to be careful of, which is when `esp` points to a gadget that has been analyzed by deRop. This happens in some advanced ROP shellcode that does looping. deRop recognizes this by checking the value of `esp` against addresses of all gadgets, and handles it with a conditional jump instruction in the output.

In the second example shown in Table 4, deRop finds out that `esp = addr4` points to a gadget previously analyzed that is located at an offset of `offset1`. In this case, the condition of the loop in the original ROP shellcode is represented by the value of `eax`, which equals to `addr4` except in the last round of the loop. Therefore, we compare the value of `eax` and `addr4` in the non-ROP shellcode, and continues the loop by jumping to offset `offset1`. Note that we apply the same strategy in other cases where it is suspected that a conditional jump is intended.

4.4.4 Function calls

Function calls are common in ROP. Special handling of function calls is not a necessity because as long as all instructions in the function are converted with semantics preserved, deRop is sound. In the example shown in Table 1, we could have continued our analysis after gadget 7, after which control is transferred to the function `SetProcessDEPPolicy()`. In that case we will be using deRop to analyze the body of `SetProcessDEPPolicy()`, which is unnecessary (since this function does not use ROP) and degrades readability (the output of deRop will not show a function call but with the body inlined). Due to these disadvantages, we decide to make an effort to recognize function calls and follow some function call conventions in the output of deRop.

Recognizing function calls in ROP can be done by identifying some function characteristics of the epilogue and prologue. For example, the beginning of a function usually contains instructions to change the value of `ebp`, `esp`, etc. However, such characteristics are not reliable especially in release versions of libraries, and we choose not to rely on them.

Instead, we use a simple but reliable heuristic that the number of instructions in a function is usually much larger than that in instructions pointed by gadgets in ROP. In particular, the number of instructions pointed to by a gadget is usually smaller than 5, while there are usually more than 50 instructions in a function. We therefore set a threshold of 50 in recognizing functions. Even if we fail to identify some very small functions using this heuristic, the drawback is minimal as not following function call convention for such a small function would not have seriously affected the malware analyzer's performance. Another heuristic deRop uses is to match the destination address with those in the export table (Windows) or GOT/PLT (Linux). If the address exists in the tables, it obviously corresponds to a library function call.

After identifying a function call, deRop chains the parameters, variables, and the return address on the stack in the right order and then updates `esp` to point to the first parameter. In cases where the parameters or variables are pointers, deRop also prepares the corresponding data and structure pointed to by these parameters and variables.

In the example shown in Table 1, there are two things to do to make sure that the function call works well in the resulting non-ROP shellcode. One is to set up `esp` as the return address after the function returns. This is simply the current value of `esp` in the non-ROP shellcode, which will be the value of `eip` when the non-ROP shellcode is executed.

The other is to set up `edi` as the target of the jump, which is the address of the function to be called. We use `jmp` instead of `call` instruction because the execution of `call` pushes the address of the next instruction on the stack, which diverges the ROP execution.

4.4.5 Unconditional jump instructions

An ROP exploit might use unconditional jumps to execute some instruction sequences indirectly. For example, the gadget may point to `<jmp eax>` while `eax` points to the instructions to be executed. In this case, deRop simply recursively applies the analyzing process on the instructions at the jump target.

4.4.6 Other instructions

These instructions can be processed easily since they do not involve reading/writing of memory locations, changing of stack layout, or changing of control flow. We simply copy these instructions to the output of deRop. Examples include `<xor eax, ebx>`, `<add eax, ebx>`, and many others.

4.5 Post-processing

There are a few steps that we can perform after all gadgets are analyzed. The purpose of this post-processing is to make the output of deRop look more similar to traditional non-ROP shellcode.

4.5.1 Data in memory

As discussed in Section 4.4, we relocate some data/constant in the original ROP shellcode to a specific location in the output of deRop, e.g., the last 20 bytes of the initial results presented in Table 1. This output is correct in the sense that it is semantically equivalent to the original ROP, but traditional non-ROP shellcode might not do this, e.g., the constants are usually directly inserted into code as operands instead of being stored at a different location on the stack. Moreover, using a specific location in memory makes the output of deRop dependent on a specific execution instance. For example, such non-ROP shellcode might not execute well in different running instances of an ASLR [29, 32] system.

deRop performs its post-processing to find out the use of such data throughout the program/malware. If the data is not directly used as a parameter of a function to be called, and is not being accessed more than once, then deRop replaces the memory access with the immediate value in the operand of the instruction, and also deletes the data from its allocated memory location in the initial result[8]. In our example shown in Table 1, this applies to gadget 1, 2, 4, 5, and 6. We can see that all the 20 bytes at the end of the non-ROP shellcode are gone after post-processing, and the memory accesses are replaced by immediate values in the instructions. Under ASLR, we set it as an option for deRop to change the absolute addresses into relative addresses (i.e., offsets from the base of the resulting non-ROP shellcode).

4.5.2 Null-bytes

Exploit payloads usually require that null-bytes do not exist, since it truncates the payload in various operations. deRop first encodes all null-bytes into non-zero values, and then adds a decoder to the final shellcode to be executed at the beginning of the exploit to restore the original value.

4.5.3 Return address

In cases where the exploit payload is to overflow a buffer to overwrite the return address, deRop replaces the word that overwrites the return address with the address of the start of the resulting shellcode.

Note that the processing of null-bytes and return address are needed only if the output of deRop is used directly to exploit the vulnerable program.

4.6 Implementation

We have implemented deRop in C++ with Visual Studio 6.0 with less then 2,000 lines of code. deRop now is implemented as a prototype, but using deRop as a product (e.g., running deRop on end user machines) is just a engineering problem. We believe deRop can be fitted into large system for malware detection and analysis. It consists of two components, a debugger and an analyzer. The debugger takes as input the length of the original ROP shellcode and the vulnerable application, and outputs the offsets of the first gadget and the value of `esp` when control is first transferred to the exploit code. It uses some Windows API (e.g., `WaitForDebugEvent()`, `GetThreadContext()` and `ContinueDebugEvent()`) to get debug information and constructs a special buffer which is shown in Figure 4.

The analyzer takes input the information from the debugger as well as the original ROP shellcode, removes ROP from it and outputs the non-ROP shellcode. The implementation of it uses a third-party tool `ndisasm`[9] to disassemble the binary code pointed to by addresses in the gadgets.

With the implementation of deRop, we run some ROP exploit examples for evaluation, and show our results in Section 5.

5. EVALUATION

In this section, we report success in applying deRop to four real-world ROP exploits. In all four cases, deRop manages to remove ROP and output the semantically equivalent non-ROP shellcode. We use the resulting non-ROP shellcode to exploit the vulnerable programs, and confirm that the same behaviors are observed in the original ROP exploit and the non-ROP exploit.

The first two columns of Table 5 summarize the four vulnerable applications and the corresponding exploits we use in our evaluation. The exploits are all published at Exploit Database[10] and tested on Windows XP SP3. Note that a common feature in these ROP exploits is that they use ROP to call `SetProcessDEPPolicy()` to make the appended non-ROP shellcode executable. Some of them do this by calling `VirtualProtect()`. In this sense, these four examples are relatively easy as the ROP portion performs limited functionality. We stress that we choose these four examples not because of their simplicity. It is simply because the fact that most current real-world ROP exploits use this strategy.

The last three columns of Table 5 show the result of our debugger in locating the gadgets (see Section 4.3). Note that in two of the exploits the first gadget is located at a large offset, and deRop manages to find it.

[8] In cases where pushad is used to prepare data (parameters, variables, and return address) of a function call, deRop also replaces them with the corresponding immediate values.

[9] http://www.nasm.us/doc/nasmdoca.html

[10] `http://www.exploit-db.com`

Table 5: Locating gadgets in ROP exploits

Vulnerable application	Exploit	Offset of 1st gadget	Offset of `esp`	Value of `esp`
CoolPlayer 2.18	DEP bypass	212 bytes	216 bytes	0x12F18C
WM Downloader 3.1.2.2	Buffer overflow & DEP bypass	17,432 bytes	17,440 bytes	0x0DC5C0
MP3-Nator	Buffer overflow & SEH-DEP bypass	28 bytes	32 bytes	0x12FBB8
SnackAmp	Buffer overflow & SEH-DEP bypass	10,564 bytes	10,570 bytes	0x12FA80

5.1 Results of removing ROP from four real-world exploits

Table 6 shows the results of removing ROP from these four exploits. An interesting observation is that the output non-ROP shellcode and the original ROP shellcode are of about the same size. This is a bit counter-intuitive since each gadget in ROP points to a sequence of instructions, which is supposed to be longer than the gadget itself. Our investigation into the details reveal that this is because

- the instruction sequence pointed to by each gadget usually contains only less than 5 instructions;
- there is a lot of junk among gadgets in ROP;
- most of the `push`, `pop`, and `esp`-related instructions are removed in the output non-ROP shellcode.

5.2 Other observations and discussions

We also encounter some special situations when applying deRop on these real-world ROP exploits.

5.2.1 *Non-ROP shellcode in ROP exploits*

This is the case for most real-world ROP exploits. ROP first calls a function which makes the non-ROP shellcode executable, and then transfers control to the non-ROP shellcode. This shows the importance of deRop being able to recognize function calls in ROP so that the output of deRop can be easily analyzed by existing malware analysis tools.

5.2.2 *System call*

ROP can be used to make system calls, which is common on Linux. deRop is able to address this in the same way of handling functions as discussed in Section 4.4. deRop recognizes system call making by looking for `<call *%gs:0x10>`, which is the new system call instruction on Linux.

5.2.3 *ASLR*

The address of the stack is different in different running instances of the same program under address space randomization. deRop is capable of removing ROP from exploits that execute on ASLR systems. Note that due to the post-processing of data in memory as discussed in Section 4.5, the output of deRop is also independent of any specific running instance. That is, although the output of the dynamic analysis of deRop (value of `esp`) differs in each running instance, the output of deRop can be always the same, and is semantically equivalent to the original ROP and executes on the vulnerable application in all running instances.

5.2.4 *Return-less ROP*

While transferring ROP into non-ROP, `ret` instruction has two special functions: 1) indicating the end of the instruction sequences pointed by gadget; 2) changing the value of `esp`. These two functions are used to extract instruction sequences pointed by gadget and trace the value of `esp`. When come to return-less ROP, through it does not use `ret` instruction sequences ending with `ret`, there are also similar instructions act like `ret`, e.g., `jmp` instruction and `pop-jmp` instructions are used to chain next gadget. Thus, we can simply monitor the ret-like instructions rather than `ret` instruction to apply deRop in return-less ROP.

5.2.5 *Semantically equivalent instructions*

As discussed in Section 4.4, for different kinds of special instructions, we use corresponding instructions to replace them. Based on the characteristics of ROP and stack , we select the most direct and uniform instructions which are certain. Actually, for different ROP shellcode, the transferring processes are a little different and there are multiple semantically equivalent instructions, so deRop provides the post-processing to optimize the resulting non-ROP shellcode. The post-processing described in Section 4.5 is just to optimize the memory related instructions. There may be more direct and simpler but equivalent instructions to optimize other types of instructions, which related to equivalent semantic technique and we leave it for future work.

6. CONCLUSION AND LIMITATIONS

In this paper, we design and implement deRop to remove return-oriented programming from malware instances. deRop enables malware analyzers to use many existing malware analysis tools to analyze ROP-based malware, which has not been taken into consideration when the existing analysis tools were designed and built. deRop is a fully automated tool that preserves the semantics of the original malware. We evaluate deRop by applying it to four real-world ROP exploits and demonstrate its success in removing ROP and preserving semantics.

We have discussed some of the limitations of deRop in previous sections, e.g., its output is one running instance specific in ASLR. Besides that, deRop needs dynamically executing the vulnerable application in order to locate the gadgets in the ROP exploit. However, we stress that this dynamic analysis does not involve running any potentially harmful instructions in the original ROP exploit code. One last limitation of deRop is that its output might still be slightly different from traditional shellcode even with the post-processing. For example, the output of deRop calls a function using `jmp`.

7. REFERENCES

[1] M. Bailey, J. Oberheide, J. Andersen, Z. M. Mao, F. Jahanian, and J. Nazario. Automated classification and analysis of internet malware. In *Proceedings of the 10th International Conference on Recent Advances in Intrusion Detection (RAID)*, 2007.

[2] P. Bania. Security mitigations for return-oriented programming attacks. *Whitepaper, Kryptos Logic Research*, 2010.

Table 6: Results in removing ROP from four real-world ROP exploits

Exploit on	CoolPlayer		WM Downloader		MP3-Nator		SnackAmp	
Code	ROP	non-ROP	ROP	non-ROP	ROP	non-ROP	ROP	non-ROP
Size (bytes)	488	495	21,831	21,418	1,003	1,202	16,700	16,764
# of gadgets	7	N/A	44	N/A	34	N/A	53	N/A
# of instructions	N/A	10	N/A	80	N/A	50	N/A	86
# of instructions executed	16	10	134	80	91	50	133	86

[3] U. Bayer, P. M. Comparetti, C. Hlauschek, C. Krugel, and E. Kirda. Scalable, behavior-based malware clustering. In *Proceedings of the 16th Annual Network and Distributed System Security Symposium (NDSS)*, 2009.

[4] U. Bayer, A. Moser, C. Krugel, and E. Kirda. Dynamic analysis of malicious code. *Journal in Computer Virology, Volume 2(1), 67-77*, 2006.

[5] T. Bletsch, X. Jiang, and V. W. Freeh. Jump-oriented programming: A new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security (ASIA CCS)*, 2011.

[6] E. Buchanan, R. Roemer, H. Shacham, and S. Savage. When good instructions go bad: generalizing return-oriented programming to risc. In *Proceedings of the 15th ACM conference on Computer and communications security (CCS)*, 2008.

[7] 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.

[8] S. Checkoway, A. J. Feldman, B. Kantor, J. A. Halderman, E. W. Felten, and H. Shacham. Can dres provide long-lasting security? the case of return-oriented programming and the avc advantage. In *Proceedings of the 2009 Electronic Voting Technology Workshop/Workshop on Trustworthy Elections (EVT/WOTE)*, 2009.

[9] P. Chen, G. Xiao, X. Shen, X. Yin, B. Mao, and L. Xie. Drop: Detecting return-oriented programming malicious code. In *Proceedings of the 5th International Conference on Information Systems Security (ICISS)*, 2009.

[10] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant. Semantics-awaremalware detection. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy*, 2005.

[11] C. Cifuentes and K. J. Gough. Decompilation of binary programs. *Software Practice and Experience, Volume 25 (7): 811-829*, July 1995.

[12] L. Davi, A. Sadeghi, and M. Winandy. Ropdefender: A detection tool to defend against return-oriented programming attacks. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security (ASIA CCS)*, 2011.

[13] A. Dinaburg, P. Royal, M. Sharif, and W. Lee. Ether: Malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM conference on Computer and communications security (CCS)*, 2008.

[14] A. Francillon and C. Castelluccia. Code injection attacks on harvard-architecture devices. In *Proceedings of the 15th ACM conference on Computer and Communications Security (CCS)*, 2008.

[15] A. Francillon, D. Perito, and C. Castelluccia. Defending embedded systems against control flow attacks. In *Proceedings of the first ACM workshop on Secure execution of untrusted code (SecuCode)*, 2009.

[16] D. Gao, M. K. Reiter, and D. Song. Binhunt: Automatically finding semantic differences in binary programs. In *Proceedings of the 10th International Conference on Information and Communications Security (ICICS)*, 2008.

[17] S. A. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security, Volume 6(3)*, 1998.

[18] X. Hu, T. cker Chiueh, and K. G. Shin. Large-scale malware indexing using function-call graphs. In *Proceedings of the 16th ACM conference on Computer and communications security (CCS)*, 2009.

[19] R. Hund, T. Holz, and F. C. Freiling. Returnoriented rootkits: Bypassing kernel code integrity protection mechanisms. In *Proceedings of the 18th USENIX Security Symposium*, 2009.

[20] J. O. Kephart and W. C. Arnold. Automatic extraction of computer virus signatures. In *Proceedings of the 4th Virus Bulletin International Conference*, 1994.

[21] T. Kornau. Return oriented programming for the arm architecture. Master's thesis, Ruhr-University Bochum, Germany, 2009. Online: `http://zynamics.com/downloads/kornautim--diplomarbeit--rop.pdf`.

[22] T. Lee and J. Mody. Behavioral classification. In *Proceedings of the 15th Annual EICAR Conference (EICAR)*, 2006.

[23] J. Li, Z. Wang, X. Jiang, M. Grace, and S. Bahram. Defeating return-oriented rootkits with "return-less" kernels. In *Proceedings of the 5th European conference on Computer systems (EuroSys)*, 2010.

[24] K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarottie, and E. Kirda. G-free: Defeating return-oriented programming through gadget-less binaries. In *Proceedings of The 26th Annual Computer Security Applications Conference (ACSAC)*, 2010.

[25] OpenBSD. W xor X, the openbsd new features. `http://www.openbsd.org/33.html`.

[26] R. Roemer, E. Buchanan, H. Shacham, and S. Savagm. Return-oriented programming: Systems,languages, and applications. *Manuscript*, 2009. Online: `http://cseweb.ucsd.edu/hovav/dist/rop.pdf`.

[27] V. S. Sathyanarayan, P. Kohli, and B. Bruhadeshwar. Signature generation and detection of malware families. In *Proceedings of the 13th Australasian*

conference on Information Security and Privacy (ACISP), 2008.

[28] 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.

[29] H. Shacham, M. Page, B. Pfaff, E. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM conference on Computer and Communications Security (CCS)*, 2004.

[30] K. Wang and S. Stolfo. Anomalous payload-based network intrusion detection. In *Proceedings of the 7th International Symposium on (RAID)*, 2004.

[31] C. Willems, T. Holz, and F. Freiling. Toward automated dynamic malware analysis using CWSandbox. *IEEE Security and Privacy, Volume 5(2)*, 2007.

[32] J. Xu, Z. Kalbarczyk, and R. K. Iyer. Transparent runtime randomization for security. In *Proceedings of the 22nd Symposium on Reliable and Distributed Systems (SRDS)*, 2003.