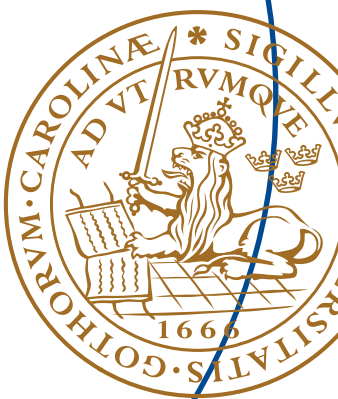
Master's Thesis

Analysis of Defenses against Return Oriented Programming

Patrik Billgren



Department of Electrical and Information Technology,
Faculty of Engineering, LTH, Lund University, June 2014.

Analysis of Defenses against Return Oriented Programming

Patrik Billgren
ada09pbi@student.lu.se

Department of Electrical and Information Technology
Lund University

Advisor: Martin Hell

June 25, 2014

Printed in Sweden
E-huset, Lund, 2014

Abstract

Return Oriented Programming is a security exploit technique which builds on code-reuse from program libraries.

Over seventeen different protections against this attack are available, which are in need of being analyzed, categorized and compared.

The protections are thoroughly compared, a couple of them are implemented and tested, and new design ideas are explored.

A new protection design ORPScan is presented, which combines the strengths of two different techniques, In Place Randomization and Input Scanning. ORPScan can be used to detect Return Oriented Programming attacks without any false positives.

Acknowledgements

I want to thank my supervisor Christopher Jämthagen for being available for discussions every week and for helping me out of hard situations. I want to thank my examiner Martin Hell for giving me the ideas to start this work, and for giving me great advice throughout the work. I also want to thank Erik Nylander for all interesting discussions about the `x86` instruction sets and running unaligned code.

Finally I want to thank Young Ik Eom who provided me with information about the Zero-sum Defender protection.

Contents

List of Figures

List of Tables

Chapter 1
Introduction

1.1 Problem Introduction

Return-Oriented Programming (ROP) is a computer security exploit technique which since 2007 [28] has been publicly known and used to attack vulnerable systems. Many different protections against ROP-based attacks have been developed and released. The protections all have different properties, different application areas and varying efficiency. Some protections work as compiler extensions, to be used once during compilation, and others work as program monitors, which monitors the execution of a program live.

The department of Electrical and Information Technology is working with research in the fields of computer and web security. Protections against ROP is one of the subjects of the research, and the department is working with evaluating their own protection.

From the department at Lund University's side, and from other departments in this scientific field's side, there is a need of classifying and analyzing the available protections. The protections need to be thoroughly analyzed and compared against each other. There is also a need of looking further into the ability to extend the functionality of the existing protections. By combining parts of different protections a more complete, or a more efficient protection may be developed.

1.2 Goals of the Thesis

The goals with the thesis are the following:

- To study and analyze different known protection techniques against ROP with focus on efficiency.
- To explore the possibility of combining parts of different techniques to find new approaches to protect against ROP.
- To implement, test and evaluate different known protection techniques.
- To implement, test and evaluate a new protection design.

1.3 Results of the Thesis

The following are the results of the thesis:

- A background description of ROP and its history.
- A summarizing description of every known protections.
- A comparison of all the known protections with focus on efficiency.
- An implementation and a description of a couple of known protections.
- An implementation and a description of a newly designed protection.
- Test results of the evaluation of the different implemented protections.

1.4 Report Outline

The report will start with a chapter with a background description, chapter 2. This chapter will describe the early techniques that led to the developing of the ROP technique. It will describe the ROP technique thoroughly, and will end with a classification of the different protection techniques.

Chapter 3 presents the summarizing description of the currently most known ROP protections. This chapter ends in a detailed comparison of the protections.

Chapter 4 presents a deeper study of two different known ROP protections. It describes the implementation in greater details, and also contains an experimental evaluation.

Chapter 5 contains a description of the newly designed protection ORPScan, together with a background, design description and an evaluation.

Chapter 2
Background

2.1 Introduction

In this chapter Return-Oriented Programming (ROP) and its history will be presented. First the buffer overflow vulnerability will be described, which is often used to initiate an ROP-attack. That will be followed by the Return-into `libc` (RILC), which is a predecessor to ROP. A thorough description of ROP and Jump-Oriented Programming (JOP) will be given, which will be succeeded by a presentation of early protections against those kind of attacks.

All the techniques will be described assuming a 32-bit `x86` architecture. The techniques described are not limited to this environment though, as they can also be used on e.g. a SPARC architecture.

2.2 Buffer Overflow

A buffer overflow means that an allocated data buffer in the memory is written with data that overflows the boundaries of the buffer. Buffer overflow is a programming error that can result out of bugs in the program or by adversaries that misuses the program in a way that was not intended. Buffer overflows can occur in dynamically or statically allocated data. Statically allocated data is placed in the stack memory region of the process, to which a buffer overflow would open up possibilites to change the program flow.

If a buffer on the stack is overflown, other variables on the stack following the buffer can be overwritten. After the local variables follows the return address for the current frame, which also can be overwritten with a sufficiently long overflow. If an adversary sends a sufficiently long data string in a buffer that is vulnerable to buffer overflow, he might be able to overwrite the return address to control the program flow. This kind of buffer overflow can be used to start the ROP-attack.

Figure 2.1 shows an illustration of the stack content before a buffer overflow attack is done. On top of the stack lies a buffer, which is a local variable allocated on the stack, with size of 4 bytes. The five top portions of the stack belongs to the current executing function's frame. If a memory sequence longer than 4 bytes is copied to the buffer, it will overwrite the other local variables. If the memory sequence is even longer it may also overwrite the saved frame pointer, the return address, the function arguments and finally the preceding function's stack frame.

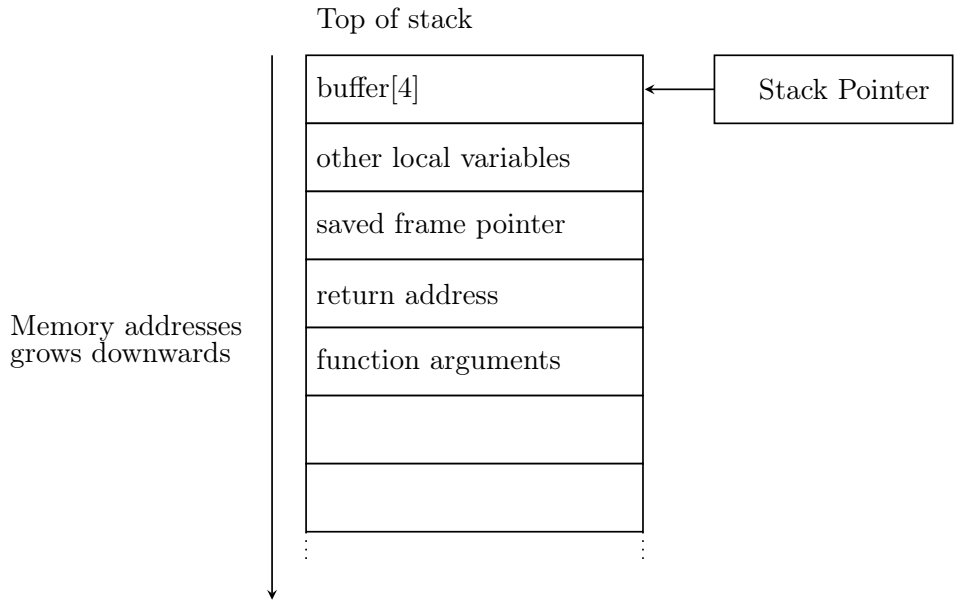


Figure 2.1: Illustration of the stack content before a buffer overflow attack.

2.3 Return-into-`libc`

Being able to subvert the control flow of a program, an adversary is able to run any code of his choice. By injecting new code into the memory space of a program, an adversary is able to run any code he wants. Injected code is often called *shellcode*.

To protect against executing injected code a new protection mechanism was invented, $W \oplus X$. $W \oplus X$ means that every memory location in a program can be either *Writable* or *Executable*, but not both. This means that if the adversary is able to inject code, he will not be able to execute it. This defense was implemented in hardware as well as in software.

Solar Designer came up with a new attack that did not involve running injected shellcode. He called it Return-into `libc` (RILC), which name alludes to its technique of returning into the library `libc`. `libc` is a library in Linux which is loaded in almost every program. By moving program flow into the library, he was able to run shellcode to start a new shell without injecting any new code. Solar Designer demonstrated the attack, which searched through the library to find an address of the function `system()`. He also searched the library to find an address to the string `"/bin/sh/"`. The intentions of the attack was to call `system()` function with that particular string, in order to start a new process with a new shell.

To perform the attack he exploited a buffer overflow vulnerability to overwrite the return address of the current function. By overwriting the return address with the address of the `system()` function, he was able to branch into the library instead of the original return point, when the current function returned. To imitate

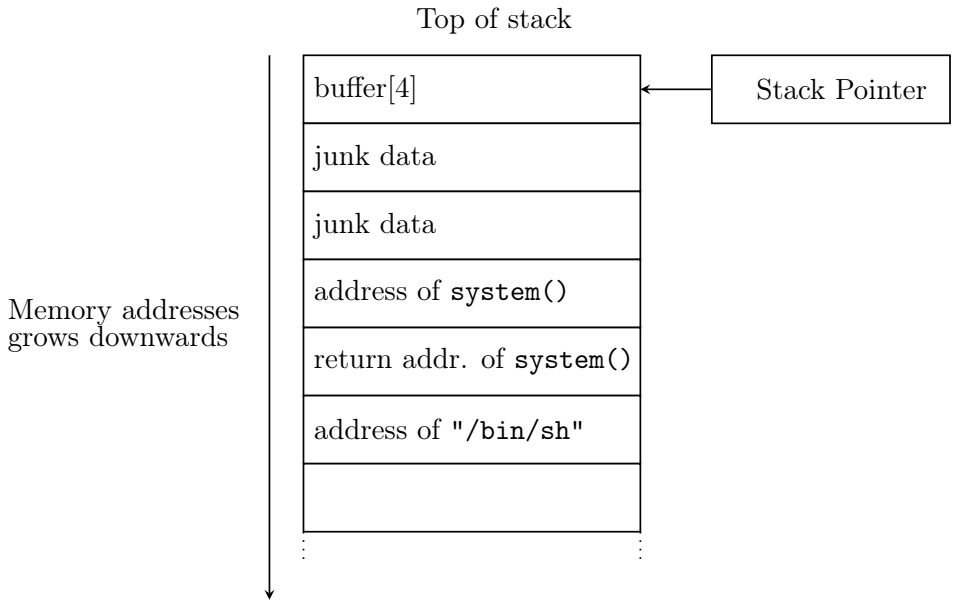


Figure 2.2: Illustration of the stack content after a buffer overflow has been exploited to initiate a RILC attack.

a normal function call, he also overwrote the words behind the return address to place the return address of the `system()` function and the argument to it. The argument was simply the address to the `"/bin/sh/"` string.

Figure 2.2 illustrates the stack right after the buffer overflow is exploited. Compare with figure 2.1 and notice how the return address has been overwritten with the address of the library function. The junk data is data that is not relevant for the attack, and can thus be overwritten with any bytes [1]. Also notice that the saved frame pointer, and parts of the preceding function's frame are overwritten. This can safely be done since the program will not return to normal control flow after the attack.

2.4 Return Oriented Programming

ROP is, similarly to RILC, a subversion of the execution flow into a library with known location. ROP is a generalization of RILC because it allows returning anywhere in a library.

The building blocks of RILC are functions in a library. In ROP the building blocks are instead snippets of code ending in a return instruction. These snippets of code are called gadgets and can consist of two or more instructions. By executing

[1] As long as it does not make the program to crash, throw an exception or terminate in any other way without returning from the current executing function.

the gadgets after each other arbitrary code can be executed. The idea behind ROP is that in a sufficiently large program or library there are enough gadgets to undertake arbitrary computation. In [28] it is proven that in the commonly used library `libc` in Linux there are enough gadgets to create Turing-complete ROP-attacks.

To understand how gadgets can be executed sequentially the return instruction has to be explained. The return instruction does two things:

```
%eip = %esp; // Move execution to %esp
%esp = %esp + 4; // Pop the stack
```

It moves execution to the address that the stack pointer points to, i.e. the top of the stack. It then pops the stack by adding 4 bytes to the stack pointer. This corresponds to an unconditional branch to the address at the top of the stack, and a stack pop. ROP abuses this principle by placing many gadget-addresses sequentially on the stack, and letting the program flow branch to them by executing return instructions.

To start the ROP-gadget execution a similar technique as in RILC is used. A buffer overflow vulnerability is exploited to overwrite the stack memory. The first gadget that is to be executed is placed on the return address of the currently executing function. The addresses of the following gadgets are then placed sequentially further down the stack. Since every gadget ends in a return instruction, the execution will jump to the start of the next gadget when a gadget ends. If a gadget needs arguments or data on the stack, that can be put between the gadget addresses.

Figure 2.3 illustrates the stack content after a buffer overflow has been exploited to start an ROP attack. Just as in the RILC attack in figure 2.2, the original return address has been overwritten with the address of the location of where the execution should branch to, i.e. the first gadget. After the first gadget address, the rest of the gadget addresses are put sequentially in the order they should be executed. The list of gadget addresses can be as long as the stack allows.

2.4.1 Return-Oriented Programming gadgets

ROP gadgets consists of a sequence of instructions ending in a return instruction.

Unaligned gadgets

The `x86` architecture is of Complex Instruction Set Computing (CISC) form and has variable instruction length. Since the instructions have variable length they are only aligned to whole words. This means that one combination of instructions can form a different combination of instructions if they are interpreted from a different starting byte. The new combination of instructions are unaligned compared to the original instructions. This property can be exploited in ROP by finding addresses of and jumping to unaligned gadgets.

The following byte sequence is disassembled as `x86` assembly instructions. The left side shows the byte sequence and the right side shows the disassembled instructions.

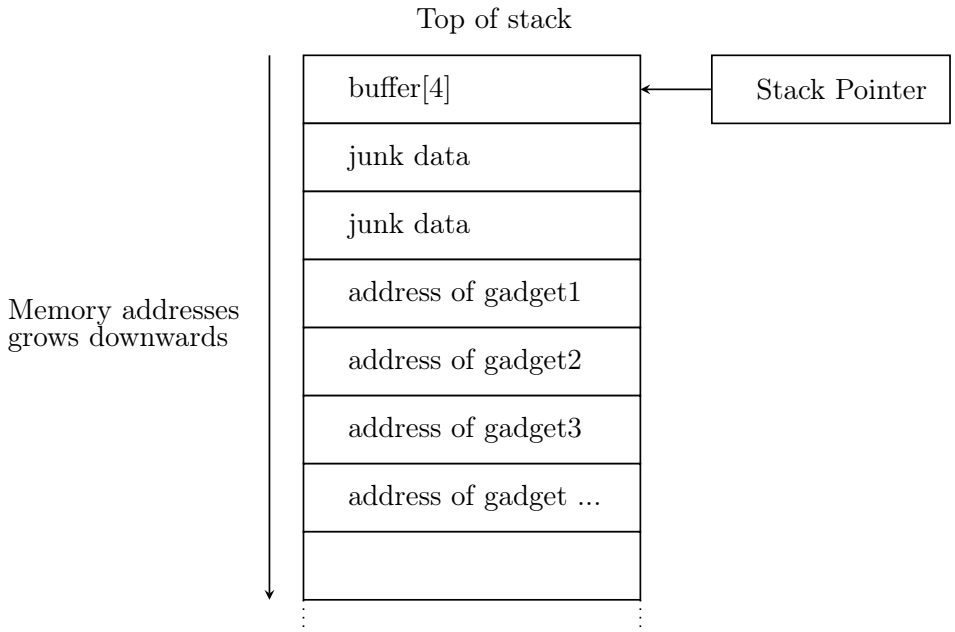


Figure 2.3: Illustration of the stack content after a buffer overflow has been exploited to initiate an ROP attack.

```
3c 24             cmp al, 0x24
24 c3             and al, 0xc3
```

On the right side we can see that the byte sequence is interpreted as a `cmp` instruction, followed by an `and` instruction. If we instead start the disassembling at the second byte in the sequence, we will get the following.

```
24 24             and al, 0x24
c3                ret
```

On the right side we can see a typical ROP gadget that ends in a return instruction. This shows how unexpected instruction sequences, that can be used as gadgets, can reside unaligned in other instructions.

Load- /Store gadgets

Load- and Store gadgets are gadgets that loads data to or stores data from registers. They can load a constant to a register, memory content to a register or store a register to a memory location.

Constants Constants can be attached to an attack by putting the constant between the gadget addresses in the ROP payload. To load the constant, the address of a gadget for loading the constant from the stack will be put in front of the constant. A gadget of that kind could have the following instructions `pop %edx; ret`. When the gadget starts executing, the stack pointer will point to the next word on the stack, i.e. the constant. `pop %edx` will put the word pointed to by the stack pointer in the register `%edx` and then move the stack pointer to the next word, which will be the address of the next gadget.

Memory content To load from and store to memory, gadgets with move-instructions can be used. `movl (%eax), %edx ; ret` is a gadget which will move the word that is located at the location pointed to by `%eax` into `%edx`. To load data from a constant address a constant-loading gadget can be used to first load a constant into `%eax`.

Similarly a store-gadget could consist of `movl %eax, (%edx); ret` to store the content of `%eax` into the memory location pointed to by `%edx`.

Arithmetic and logical gadgets

Arithmetic and logical gadgets are gadgets that computes addition, subtraction, exclusive-or, and, or, not, shifts and rotates. All binary operations have simple instruction sequences as gadgets. They take two registers as parameters, where both registers contains the values or alternatively one of the registers point to a memory location with the value. Immediate binary instructions can do operations with constant values, without loading them into a register. E.g. `addl %eax, %(edx); ret` adds the value of `%eax` with the value pointed to by `%edx` and puts the result in the second operand.

Branching gadgets

Branches can be divided into unconditional branches and conditional branches, where conditional branches are the hardest to implement in ROP.

Unconditional branches Branching in ROP is very different from normal execution. In normal execution the instruction pointer `%eip` is manipulated to do branching e.g. in loops. To implement a loop of several gadgets in ROP, the `%esp` has to be manipulated instead, since it always points to the next gadget that is to be executed. To do an absolute branch the absolute address can be put on the stack in the same way as a constant. To load it into `%esp` a gadget that looks like `pop %esp; ret` could be used.

Relative branches can be implemented as a subtraction or addition on the `%esp`, for doing a backward jump or a forward jump. When doing a subtraction caution has to be taken if the values above the stack have been changed.

Conditional branches There are many different ways to implement conditional branches. [28] and [26] both proposes solutions that uses an instruction that sets the carry flag and uses the carry flag to perturb the `%esp`. The carry flag can be used to set a word to either `0xFF FF FF FF` or `0x00 00 00 00` depending on if it is true or false. This word can be `and`-ed with the the relative value that is to be added to `%esp`.

E.g. to test if a value is zero, the `neg` instruction can be used. The `neg` instruction sets the carry flag if the value is non-zero, since it uses two's-complement to negate the value. To test whether two values are equal, they can be subtracted and then negated. If the subtraction results in non-zero, the values are not equal and the carry flag will be set. To test if a value is smaller than another one can be subtracted from the other, which will set the carry flag if the right-hand side is larger than the left-hand side.

The carry flag can be extended into a word of all ones using a combination of different gadgets. The instruction `sbb %esi, %esi` can be used if `%esi` is set to zero. The instruction subtracts `%esi` plus the carry flag from `%esi` and saves the result in `%esi`. This means that if the carry flag is true, `%esi` will become `0xFF FF FF FF` and else it will become `0x00 00 00 00`. A constant can now be loaded and `and`-ed with `%esi` and then added or subtracted to `%esp` in order to implement a conditional branch.

Call gadgets

ROP-attacks are often used to execute injected code, or to start a new process such as a shell. In order to do that gadgets that call functions are used. Since calling a function and returning from it during a ROP-attack does not disturb the attack directly, it simply could contain a call instruction, e.g. `call %esi; ret`.

There are however a couple of things to consider.

- If old gadget addresses above the stack pointer needs to be kept, care has to be taken when calling a function. Because the return address and the

local variables of the function will be allocated on the stack, the old gadget addresses need to be saved elsewhere.

- Caller-saved registers are register that the calling function must save if they cannot be changed during the function call. In Linux these registers are `%eax`, `%ecx` and `%edx`.

Adapting gadgets

When looking for gadgets in a library it can be hard to find gadgets that have the exact desired properties. Memory references could have different offsets, arithmetic operations can have different operands or gadgets can contain unwanted instructions. Often these differences can be accounted for when crafting a ROP-attack, and the unwanted side-effects can be removed.

If a memory reference contains an unwanted immediate offset such as `movl 32(%eax), %edx; ret`, the value in `%eax` has to be the desired address minus 32. To accomplish this, a constant could be adjusted in the ROP-payload or an arithmetic gadget could be used to subtract 32 of the register.

Null-bytes

There are limitations on the addresses of the gadgets, that together with the constants, will most often be inserted in the exploited application as strings or streams of data. Since string and stream manipulation will consider a `null`-byte (`0x00`) as an end of string or end of stream, there cannot be any `null`-bytes in the payload. This puts limitations on gadget addresses, to not choose gadgets with addresses containing `null`-bytes. Constants will need to be manipulated before using them in the program. E.g. if the constant `0x10 00 00 00` is needed, a bitwise inversion gadget can be used on the loaded constant `0xEF FF FF FF` which does not contain any `null`-bytes.

Gadgets with side-effects

If a gadget contains instructions that creates unwanted side-effects, other gadgets may have to be executed to restore the desired state.

E.g. if an addition-gadget contains the following instruction `addl (%edx); %eax, push %edi; ret`, the `push %edi` instruction is not part of the addition and will change the state of the stack. To be able to use this gadget as an addition gadget this has to be accounted for in the following ways. Since the content of `%edi` will be put on the top of the stack, which is where the execution will branch on the return of this gadget, the content of `%edi` has to be an executable memory address. This can be achieved by using a load constant technique, with a valid memory address as a constant. In order to not execute code that will have more unwanted side-effects, the address to a return instruction can be passed. A gadget that only consists of a return instruction is the ROP version of a No Operation Instruction (NOP).

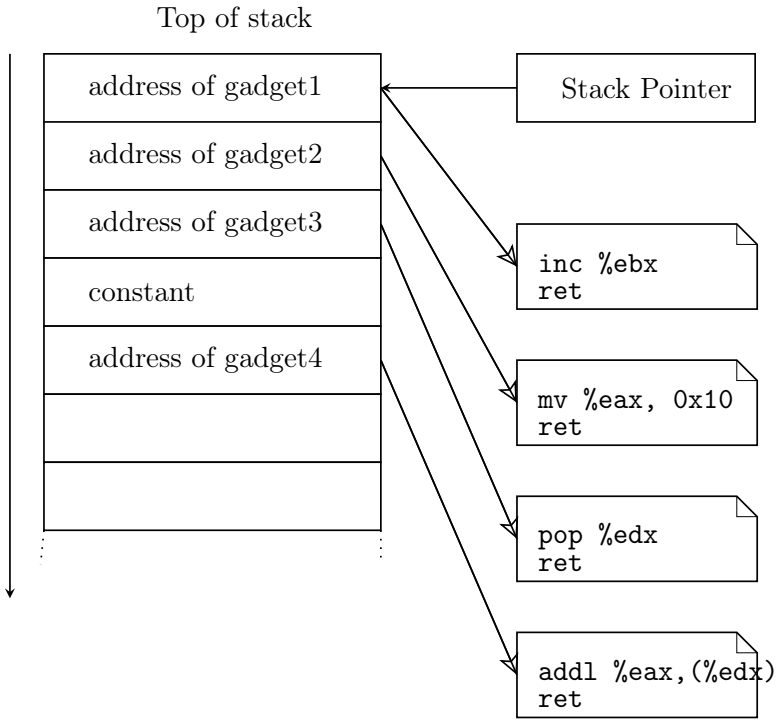


Figure 2.4: Illustration of the stack during a real ROP attack. The gadgets are illustrated to the right in the figure.

2.4.2 Example

Figure 2.4 shows an illustration of a real ROP attack in action. The stack pointer points to the top of the stack, and the gadget next to be executed. For every gadget that is executed the stack pointer will be increased and point to the next gadget. Gadget number three is a constant loading gadget, that loads the constant succeeding the gadget address on the stack to the `%edx` register. After the `pop` instruction the stack pointer will be increased to point to the fourth gadget.

2.5 Jump-Oriented Programming

In this section the Jump-Oriented Programming (JOP) will be described. Similarly to ROP, JOP uses a series of gadgets that are chosen to run after each other. The gadgets are put in a dispatch table which is then called by the dispatcher.

Two main differences between JOP and ROP is that JOP does not rely on the `%esp`-register to point to gadgets, and the control flow is not controlled by return instructions. In JOP any register can be used to point to the next gadget, and the control flow is controlled by a dispatcher gadget, which moves control flow from one gadget to the next within a dispatch table.

2.5.1 Dispatch table

The dispatch table is the equivalent of the stack in a ROP-attack where the gadget addresses and additional data is loaded. The dispatch table can be located anywhere in the memory and does not need to be executable. The purpose of the dispatch table is to locate all the gadgets that are to be executed in the attack, and to provide them with data.

2.5.2 Gadgets

In JOP there are two different types of gadgets, dispatcher gadgets and functional gadgets. The dispatcher gadget is used to govern the control flow. The functional gadgets are the gadgets that perform the arithmetic that is the purpose of the attack.

Dispatcher Gadget

The dispatcher gadget is responsible for controlling the control flow. It is the equivalent to the return instruction in ROP. To do this it has a virtual program counter which points to memory addresses in the dispatcher table. The dispatcher gadget increments the program counter with a constant and then jumps to it. By using this same dispatcher gadget, the attack will be able to run all functional gadgets in the dispatcher table sequentially.

The way the dispatcher gadget modifies the program counter is not limited to a specific constant. The dispatch table can be laid out in memory with larger gaps between the addresses. It can also work like a linked list where the dispatcher gadget dereferences the program counter each time.

Here is an example of a dispatcher gadget which uses the `ecx` register as program counter:

```
ecx = ecx + 8;
jump [ecx];
```

Functional Gadgets

The functional gadgets are similar to the gadgets in ROP. They contain a sequence of useful instructions and ends in a branch instruction. Instead of ending in a return instruction, they should end in a jump- or call instruction that branches to the dispatcher gadget, or another gadget that does. The different branch instructions available are indirect jumps or calls. Calls can be used because the side effect of pushing a return address on the stack will not affect the attack.

Functional gadgets can perform the same as ROP-gadgets with a couple exceptions. Since the virtual program counter is in a register at all times, it can be altered to simulate branching gadgets. Unconditional branching can be achieved by changing the program counter. Conditional branching can be achieved by changing the program counter's value with a conditionally calculated value, or by using the conditional move instruction that is available on the x86.

The other exception is that data cannot be loaded on the stack when starting the JOP-attack. Functional gadgets will then have to use a pointer to where the data is loaded at. To load data from a pointer and increment it, there are a few different string loading and loop sequence instructions to use. The stack can although be used to push and pop data to and from, since the stack is not used to control the control flow.

2.5.3 Initializing

To initialize the attack, just as in ROP, the stack has to be overwritten or a functional pointer has to be overwritten. Another approach is to overwrite the `setjmp` buffers, which is a description of a Central Processing Unit (CPU) state, that can be restored by executing the instruction `longjmp`. The dispatch table has to be injected in the memory, and then an initializing gadget is called. The initializing gadget sets all registers to a specific start state, and then redirects control flow to the dispatcher gadget.

2.6 Early Protections

This section presents the early protections which were not directly focused on prevention of ROP-attacks. The next section presents the different techniques preventing ROP-attacks.

The first protections against buffer overflow attacks, and the attacks based on them were compiler- and operating system based. The first vulnerabilities that was protected was the buffer overflow and code injection attacks. Later were protections against code-reuse attacks developed which were also operating system dependent.

2.6.1 Buffer overflow and code injection

Protections against buffer-overflow attacks on the stack have been present since 1998 in StackGuard [8]. StackGuard is a compiler extension that offers protection against buffer-overflow attacks. StackGuard places a canary word on the stack, before the return address of the active frame. This effectively forces an adversary who wants to overwrite the return address to also overwrite the canary word. The canary word is compared with the original to see whether that is the case. The canary word is randomly chosen, to make guesses impossible. Other similar protections are ProPolice [11] and StackGhost [12] for the Sun Microsystem's Sparc processor architecture.

As mentioned earlier, the $W \oplus X$-protection was introduced in the OpenBSD operating system. The $W \oplus X$ enforces every memory page in the process' address space to be either writable or executable. This enforcement made simple code injection techniques impossible. If arbitrary code is written by an adversary on a memory page, it will never be able to execute. Also, if a memory page is executable, the adversary will never be able to write code to that page.

Similar techniques were implemented in hardware in processors. Intel and AMD implemented the feature as one bit per memory area, that represents if the memory area is executable or not. The solutions were called eXecute Disable (XD) and eXecute Never (XN) respectively. In software, the $W \oplus X$ is called Data Execution Prevention (DEP) and was implemented in Linux in kernel 2.6.8 (2004), Windows in XP SP 2 (2004) and in MacOSX during their transition to x86 (2006).

2.6.2 Code-reuse attacks

Address Space Layout Randomization (ASLR) is a protection against code-reuse attacks and was introduced by the Linux PaX project in 2001. The idea of ASLR is to randomize the layout of the memory image each time an application is loaded into memory. The base address of the stack, heap and loaded libraries will be randomized, which makes it hard to guess the location of where injected code is located and where library code is located. ASLR makes it very hard to perform code-reuse attacks, and in combination with DEP it creates a very solid protection against most buffer-overflow based attacks.

In Linux kernel version 2.6.12 (2005) ASLR was enabled as default. Linux also implemented Position Independent Code (PIC), which randomizes the base address of the main executable, in 2003. Microsoft introduced ASLR in 2007 with Windows Vista, and have included it in all newer version. It is only enforced as a default option for core operating system binaries. Applications require a special linker flag to enforce ASLR. MacOSX introduced ASLR in 2007 for system libraries. In 2012 the entire MacOSX system including applications is protected by ASLR.

2.6.3 Circumventions

Despite all these protections, there are still buffer-overflow and other similar vulnerabilities in modern software. Bulba and Kil3r presented in 2000 [5], different

methods of circumventing the StackGuard and StackShield protections. StackShield is similar to StackGuard a stack protection, by saving the original return address in a special table, making it impossible to overwrite. Bulba and Kil3r's solution originates in alterations of pointers that are not the return address, e.g. function pointers and `longjmp` buffers.

Shacham et.al. presented the weaknesses of ASLR in [29]. They showed that 32-bit architectures has too narrow limits in entropy and used a derandomizing technique to find the location of ASLR-protected libraries. Their solutions also worked on systems with DEP enabled. Other ASLR-circumvention techniques are partial address overwrites [10] and information disclosure [3].

2.7 ROP-Protections

For a system using both $W \oplus X$ and ASLR there is still a need for protections against ROP-attacks. There are many different kinds of ROP-protection, from which almost every known is present in this analysis.

The different protections presented in this analysis have been categorized into five different main categories; Control Flow Integrity, Instruction Monitoring, Input Scanning, Instruction Rewriting and Memory Randomization. This categorization is a part of the results of the thesis work, and is done to generalize the protections to better understand them. This categorization divides the protections in fairly even groups, except for Input Scanning, which only has one member.

The categories will be described here.

2.7.1 Control Flow Integrity

Control Flow Integrity (CFI) is a low-level protection against code-reuse attacks. In a code-reuse attack, the adversary manipulates the program flow into non-intended execution paths. The idea of CFI is to force the control flow of a program to the execution paths that are intended by the programmer. This can be done at different levels of granularity.

In [37], a computable metric is proposed, that can be used to compare the efficiency of different CFI techniques. Average Indirect target Recordeduction (AIR) is a value that describes the Control Flow protection a technique can provide. The value is a fraction of the amount of indirect branch-targets that are eliminated by the technique. In an ideal CFI-technique the AIR value would be 1. There are a few different kinds of CFI-techniques.

Instruction-CFI This technique removes all unaligned branch-targets by restricting CFI to instruction boundaries.

Bundle-CFI Puts instructions into 16- or 32 byte bundles and forces CFI to these bundles.

Reloc-CFI Relies on relocation information in binaries and forces all indirect branches to target only these locations. Return branches are forced to locations directly after call-instructions.

Strict-CFI Has the same enforcements as reloc-CFI but does not need the relocation information.

Bin-CFI This technique can handle branches in more complex binaries that contain e.g. returns used as jumps, returns to caller function without return address (C++ exceptions), jumps to return addresses (`longjump`), runtime generation of new Indirect Control Flow (ICF) targets and indirect jumps using arithmetic operations.

CFI-based techniques covered in the analysis:

CFI Section 3.1

Control Flow Locking Section 3.6

G-Free Section 3.3

Control Flow Integrity and Randomization for Binary Executables Section 3.14

Control Flow Integrity for COTS libraries Section 3.15

2.7.2 Instruction Monitoring

Instruction Monitoring is a technique where certain instructions are monitored by connecting them to certain events. Instruction Monitoring is based on assumptions about the state of an application during the execution of different instructions. By monitoring the application, and checking the state of it during the connected events, possible ROP-attacks can be detected.

A simple example of an instruction monitoring based technique is the ROPDefender. ROPDefender assumes that every function in an application will always return to the address that succeeds the address that it was called from. By monitoring each call- and return instruction in an application, this state can be validated during execution.

Monitoring instructions can be done in different ways. It can be done statically during compilation by inserting calls to functions after the specific instructions. It can also be done dynamically by running the application in a Virtual Machine (VM) or by instrumenting it with a framework like PIN.

Instruction monitoring techniques covered in the analysis:

DROP Section 3.2

ROPDefender Section 3.5

kBouncer Section 3.11

ROPGuard Section 3.12

ROPecker Section 3.16

Zero-sum Defender Section 3.17

2.7.3 Input Scanning

Input Scanners, as the name suggests, scan input data in order to detect ROP-attacks. A ROP-attack is similar to a code injection attack in the way that it injects data that manipulates the control flow of the program. There are many input scanners that detects code injection attacks, by looking for instructions in the data. The data, or the payload, in a ROP-attack differs from the payload in a code-injection attack by consisting of addresses to gadgets instead of instructions. A more detailed description of input scanners is given in section 5.2.

There is only one known input scanner in this analysis, ROPScan. ROPScan scans the input data for potential gadget addresses, and emulates them in an emulation environment to determine if they are actual gadgets.

Input Scanning techniques covered in this analysis:

ROPScan 3.7

2.7.4 Instruction Rewriting

Instruction Rewriting is a technique that removes the possibility of performing an ROP-attack by rewriting the instructions of the protected application. The instructions can be rewritten to completely remove usable gadgets. Instructions can also be rewritten during the loading of an application, making existing ROP-gadgets useless. Instructions can be rewritten as assembly instructions before assembling the application, or directly in the binary file as assembled instructions.

A simple example of instruction rewriting is the Return-less Kernel. The Return-less Kernel rewrites the assembly instructions of a kernel to remove all opcodes that represent the return instruction. By removing every return opcode available in the memory image this technique removes the possibility to perform any ROP-attack.

Other types of instruction rewriting techniques can e.g. change the order of certain instructions and change the use of registers and memory addresses.

Instruction Rewriting techniques covered in this analysis:

Return-less Kernels Section 3.4

G-Free Section 3.3. This is combination of Instruction rewriting and CFI.

In Place Randomization Section 3.10

2.7.5 Memory Randomization

Memory Randomization is a technique where the location of executable code is randomized in the memory space for every running instance of the application. Memory randomization is a similar protection to ASLR. In ASLR the location of whole libraries are randomized, while in these techniques the granularity is smaller. An application that is protected with memory randomization is very hard to attack with ROP, since the addresses of the gadgets are not known before running the application. The addresses of the gadgets are also new every time the application is run.

In Binary Stirring the location of basic blocks are randomized, Marlin randomizes locations of functions and Instruction Location Randomization randomizes locations of instructions.

Memory Randomization techniques covered in the analysis:

Binary Stirring Section 3.8

Instruction Location Randomization Section 3.9

Marlin Section 3.13

Chapter 3

ROP-Protections

A result of this thesis work is an analysis of the most known ROP-defenses. In this chapter the analysis will be presented. The different ROP-defenses will be presented in chronological order, in the same manner. Each presentation consists of a short introduction, a description of the general design, a description of the implementation details and finally of an analysis.

3.1 Control Flow Integrity

Control Flow Integrity [2] is a simple CFI-technique that is based on binary rewriting to enforce all control flow transfers to follow the original Control Flow Graph (CFG). To do this, all ICF targets are checked dynamically.

3.1.1 Main Features

First the binary is analyzed to determine its CFG. The CFG is the base of the program flow that CFI enforces. The computed CFG is conservative in the call flows, as it allows any call to target any function entry. To restrict the program flow of the CFG an analysis of the relocation entries in the binary is done.

To enforce the computed CFG the binary is instrumented to perform ICF validations. The instrumentation is based on giving each procedure a 32 bit identification. Before each ICF the branching function assumes the target has an identification and verifies that it is a valid identification for this branch.

Identification Insertion Identifications can be inserted with the side-effect free instruction `prefetchnta` with the identification as an immediate, which prefetches data into caches. This instruction is inserted at the address of each ICF target.

Identification Validation Identifications are verified with a compare-instruction, that compares the inserted identification at the target address, and the identification which is a hard-coded immediate operand. The inserted identification is positioned 4 bytes into the target, because of the opcode of `prefetchnta`. If the compare is not equal it is assumed that the branch is invalid and a jump is done to an error procedure. Otherwise a jump is

performed into the destination as normal. Return branches will be changed to jump-instructions.

There are a few properties of the identifications that must hold in order for this technique to be effective against ROP-attacks.

- The identification bit sequences have to be unique and not present anywhere in the memory except for in the identification and the identification validation.
- The identification bit sequences must reside in non-writable code, to make it impossible for the adversary to change the identifications. This holds true for most systems, that prevent a program to write to its code segment.
- The identification bit sequences must not be executable, to make it impossible to craft new instructions from the middle of the sequence.

  An alternative to this property is to make the sequences random for each loading of the program. It is then possible to use the sequences as immediate operands in the code, but the adversary cannot craft gadgets in beforehand.

3.1.2 Implementation details

The CFG is computed with `Vulcan` [31], which is an instrumentation system for x86 binaries that only requires the binary. When instructions are added to the binary, most address references in the code must be changed.

3.1.3 Analysis

Strengths

- Has a simple, easy and verifiable implementation.

Weaknesses

- Has not been tested against ROP-attacks.
- Has high run-time overhead compared to other protections.

Other thoughts

All aligned return instructions are removed. The rest of the branch instructions are all protected with validation code. However, all unaligned branch instructions are still left in the program, which means that they could still be used. Also some gadgets could be constructed from the identification bits in combination with the jump succeeding it, if not care is taken to not choose identifications that have opcodes in it.

3.2 DROP: Detecting Return-Oriented Programming Malicious Code

DROP [6] is one of the first known protections against ROP. It is based on dynamic runtime instruction instrumentation to recognize ROP-attack patterns.

3.2.1 Main features

DROP specifies two important criterions, G_{size} and S_{length}. G_{size} is the number of instructions in a gadget and S_{length} is the number of gadgets in a row within the same library/binary memory space. The measured G_{size} and S_{length} can be used to decide if a sequence of executed instructions is an ROP-attack or not.

DROP specifies thresholds T_0 and T_1 respectively for both these criterions in order to detect ROP-attacks. If G_{size} for an instruction sequence is below T_0 the sequence is considered to be a gadget and if it is above it is not. If S_{length} is above T_1 the gadget-sequence is considered to be an ROP-gadget sequence and if it is below it is not.

At every return instruction that is executed an ROP-detection check is done. The exact detection algorithm is as follows: DROP recognizes the return instruction and records the target address. If the address is within the library that is checked against and the number of instructions is below T_0 it then records the gadget as a candidate gadget. The number of contiguous candidate gadgets (gadgets that are run after each other in the same memory space) is then calculated and if it is bigger than T_1, an ROP-attack is detected.

Chen et. al. have performed experimentation to determine the most optimal values of T_0 and T_1. They have tested DROP on hundreds of applications with sizes between $10KB$ and $100MB$. According to their analysis the most optimal values for T_0 and T_1 is 5 and 3 respectively. With these values there are no false positives or false negatives encountered.

3.2.2 Implementation details

DROP would need to be implemented as part of a dynamic binary instrumentation tool such as `Valgrind` [20]. The algorithm is fairly easy to implement. Alternatively it could be implemented in `PIN`, which would be faster than `Valgrind`. `Valgrind` translates the binary code into `VEX` intermediate language.

3.2.3 Analysis

Strengths

- A simple algorithm that has no false positives.

Weaknesses

- Slows down the program execution significantly (on average 5.3x).
- Only detects return-gadgets, but not gadget that ends in jump instructions.

- Cannot detect ROP-attacks with gadgets in different libraries.
- Cannot detect ROP-attacks with gadgets that consist of less than three instructions.

3.3 G-Free - Gadget Free Binaries

G-Free [21] is a combination of CFI and Instruction rewriting. It removes all possible gadgets in the binary by forcing execution of functions from start to finish and rewriting instructions.

3.3.1 Main features

Eliminates all possible sources of reusable instructions. By combining a few different techniques it is possible to eliminate almost all gadgets in the binary. It forces execution to be done from start to finish in each function which means it de-generalizes any potential ROP-attack into a RILC attack. Running arbitrary code is therefore impossible and Turing-completeness is removed.

The following techniques are used in G-free:

Alignment Sleds

Creating a NOP-sled before critical code to avoid it being run unaligned. Even if execution has jumped into an instruction unaligned it will be forced to align when it reaches the NOP-sled. A maximum of 9 NOP's is needed in all cases. The NOP-sled can be prepended with a jump to the end of the sled, to skip it under normal execution.

Return Address Protection

Adding a header and a footer to each function that ends in a return instruction. The header encrypts the saved return address and the footer decrypts it. If the function would get jumped to in the middle, the return address will become invalid before returning. The encryption consists of a simple x-or with a random key.

Frame Cookies

All functions that contain a jump or a call get a header that computes and pushes a function-specific and run-time random cookie that is validated before doing the jump/call. If the validation goes wrong execution is stopped. A footer is inserted that removes the cookie from the frame. This modification changes the stack layout which will require further changes on memory offsets and references.

Code Rewriting

This is done to eliminate unintended free-branch instructions. This can be done instead of using the alignment sled. There are a few different code rewriting techniques.

Register Reallocation Unintended return instructions can appear if certain combinations of registers are used as operands in different instructions. To avoid this, the register allocation performed during compilation is manipulated. Either the registers can be swapped with other registers, or the allocation algorithm can be run again.

Unintended return instructions can also appear in immediate floating point instructions. They cannot be removed easily so an alignment sled has to be put in front of it.

Instruction Transformation There are a couple of instructions that contain the opcode for return. These can be removed by simply switching them with one or a couple of instructions that have the same effect. E.g. `movnti (0x0f 0xc3)` can be replaced by a regular `mov`.

Jump Offset Adjustments If a relative jump offset contains the opcode for return it has to be changed. If the opcode is present in the least significant bytes there can simply be some NOP's inserted to move the jump destination. If the opcode is present in the more significant bytes it needs 256, $64K$ and lastly $12M$ nops.

According to the authors even situations with the opcode in the second least significant byte is not very common, because it would correspond to a jump of $12MB$. If it was discovered anyway it would be possible to relocate the functions or code chunks addressed by the jump.

Immediate and Displacement Reconstruction Immediate values of instructions can contain the opcode for free-branch instructions. To remove statements like that they can be switched with several other statements that constructs the same value. For example if an immediate value is loaded, that value minus one can be loaded instead, and then increased. Memory accesses can also have unintended opcode in it, and have to be rewritten.

Inter-Instruction Barriers Some instructions after each other will create new unaligned jump or call instructions. To avoid this a barrier of NOP's can be placed between them. In some cases NOP's are not enough, because they can also create a new such instruction. If the last byte of an instruction, e.g. is `0xFF`, this will become a `jmp` together with `0x90` (`nop`). An instruction that has no effect, e.g. `mov %eax,%eax`, can then be chosen instead.

3.3.2 Implementation details

G-free is implemented as a pre-processor for the GNU Assembler, which is the backend of GCC.

The assembly code that is generated by the GCC assembler `cc1` is intercepted. The modifications are then done to remove all possible gadgets. The new assembly code is then handed over to the `gas` assembler. The compiler or assembler are neither modified in this implementation.

One problem is that real numeric values of immediate values and memory displacements at this point in compilation cannot be seen, which means this has to be done as a two-step process. In the first step the different locations where immediate values resides will be tagged. In the post-processing step the binary file is checked whether the tagged locations need to be modified. This produces a log file, which will be fed to the pre-processor again for a second time compilation. This time it can modify the values that need to be modified.

If NOP's are inserted many memory references may get their value changed, which means that new unintended code might show up. The compilation may therefore be run again until all such opcodes have disappeared.

3.3.3 Analysis

Strengths

- G-free has low overhead. Tested on different applications with 3.1% overhead and in the Phoronix Test Suite with 1% overhead.

Weaknesses

- It creates large binaries (+30%).
- There are still a few gadgets left.
- It is hard to implement since there is much handwritten code in `libc`.

3.4 Return-less Kernels

Return-less Kernels [19] is a compiler-level instruction rewriting protection. It uses three different techniques to completely remove all four different types of intended and unintended return instructions from code.

3.4.1 Main features

Three different techniques are implemented during the compilation of applications; Return Indirection, Register Allocation and Peephole Optimization.

Return indirection

This technique removes all return opcodes that are within normal return instructions. Normal execution flow consists of call - return pairs. When exploited by ROP however, the execution flow will consist of many returns without any calls.

In Return Indirection the call and return instructions are modified. The call instruction will push a return index on the stack instead of a return address. The return index corresponds to a return address which lies in a return address table. When the return instruction is executed it pops the return index and looks up the corresponding return address in the return address table.

The new call is implemented as `push $index, jmp dst`. The new return is implemented as `pop %reg, jmp *RetAddrBase(%reg)`. The return address table will be protected from modification with the help of DEP. The table can be generated offline, since all the return addresses of a call will be the instruction after the call.

Register Allocator

This technique removes return opcodes that are part of an instruction operand as a register. For example the instruction `mov %rax, %rbx` has the machine code `48 89 c3` which contains a return instruction (`c3`). There are only two sets of registers that need to be replaced; `%rbx` and `%rdx` combined with certain instructions (mostly `mov`).

In the Return-less Kernel prototype this is implemented as a linear scan register allocation with additional support to remove return opcodes that are considered unsafe due to return opcodes. During the algorithm, if an unsafe register is used the algorithm is run again without the unsafe register.

Peephole Optimization

This technique removes return opcodes that are contained in the opcode of non-return instructions or in the immediate part of different instructions.

In the first case, all unsafe instruction can be searched for offline and replaced by other instructions with the same purpose. The analysis in the paper shows that only one such instruction is present in the FreeBSD kernel: `movnti mem32/64, reg32/64`. It can be replaced with a simple `mov`.

For immediate operands there are two general scenarios: If the operand is an immediate constant or if the operand is a relative offset in a jump instruction. In the former the instruction can be replaced with some multiple steps to acquire the same results. For example if a register is to be compared with a constant, it can be replaced with code for loading the constant minus one into a register. The register can then be increased before comparing the two registers.

In the latter case, NOP-instructions can be inserted to make the offset larger. This works if the return opcode is in the first or second byte positions ($1 - 255$ NOP's). If the return opcode is in a higher-order byte than that we will need at least $64K$ of NOP's, and the link script will instead be changed to relocate the target function.

3.4.2 Implementation details

`LLVM` is a compiler framework which provides program analysis and transformations at different phases such as compiler-, link- and runtime. All features in the Return-less Kernel are implemented in the back-end of `LLVM` which provides a high-quality code generator.

The return indirection is implemented at the end of the prologue/epilogue code insertion phase, the register allocation is merged into the current register allocation phase and the peephole optimization is implemented in the late machine code optimization phase.

3.4.3 Analysis

Strengths

- Removes all return instructions in the code which simply removes any possible ROP-attack.

Weaknesses

- Return-less Kernels will not remove the possibility of RILC attacks, since calls of `libc`-functions are still in the return-address table.
- Since it is a compiler-based approach the source code of the kernel is needed. This makes it impossible to support e.g. Windows.
- Does not protect against JOP-attacks

Other thoughts

This technique does not handle `iret`-instructions, which there are only three of in the FreeBSD kernel. If implemented in Linux this instruction has to be analyzed as well.

The peephole optimization for instructions that unintentionally contains return-instruction will have to be extended when used for other libraries and applications than the FreeBSD kernel.

3.5 ROPDefender

ROPDefender [9] is a Just In Time (JIT)-monitoring of code at runtime that detects ROP-attacks by examining call-return pairs.

3.5.1 Main features

The main idea is to detect ROP-characteristics regarding call and return pairs. In a normal program flow every call will be followed by a return that returns to the same address that was pushed in the call. This general rule is broken during an ROP-attack and that is what this defense detects. There are also some other exceptions to this general rule that has to be taken into account.

Shadow stack

To control the call-return pairs a shadow stack is used. Every time a call is executed the return address is pushed onto the shadow stack. Every time a return is executed the top of the stack is compared with the top of the shadow stack. If they do not match an ROP-attack is detected.

ROPDefender consists of two components: a detection unit and a binary instrumentation engine. The detection unit inspects the current instruction, calls appropriate routines and handles the shadow stack. ROPDefender can successfully intercept unintended code since it is always evaluates the same instruction that is being interpreted by the instruction pointer.

All components of ROPDefender are parts of the Trusted Computing Base (TCB), which means the adversary cannot attack it. Since every instruction executed is under the control of the VM it should be completely protected.

Exceptions to the call-return rule

The system calls `setjmp` and `longjmp` allow functions to bypass multiple stack frames. These system calls can be used for non-local jumps and are usually used in exception handling to unroll a stack. The first call saves the current execution state of the process and the second restores the saved execution state.

To address this ROPDefender simply pops the shadow stack until a match is found or until it is empty. If a match is not found during the popping, an ROP-attack is detected.

3.5.2 Implementation details

ROPDefender is built upon the binary instrumentation framework `PIN`. The detection unit is written in C++ in about 80 Lines of Code (LOC).

PIN, that consists of a VM with a JIT compiler and an emulation unit, can be configured via Pintools. ROPDefender is written as a Pintool in which the instrumentation code is specified. PIN loads both the Pintool and the target process, and then starts to compile the instructions into instrumentation code. There are specific functions in the PIN Application Programming Interface (API) that can determine if an instruction is a call or a return. If these functions return

true, the analysis routines are called. The analysis routines will then be run before the actual instruction.

A more detailed description of the implementation is given in section 4.2.

3.5.3 Analysis

Strengths

- Simple implementation, written in only 80 lines of code.

Weaknesses

- Does not protect against gadgets ending in jump instructions.
- High run-time overhead
- The security depends on the security of the TCB. If the adversary can attack through the TCB, this system is not secure.

3.6 Control-Flow Locking

Control-Flow Locking [4] is an assembly and binary rewriting technique based on CFI. It uses locks, that are placed before all indirect calls.

3.6.1 Main features

The authors describe three kinds of control-flow operations that need to be protected: instructions that unintendedly happen to implement call, jump or return; return instructions; and indirect call and jump instructions. The first category is protected by aligning code as in Instruction-CFI (see section 2.7.1).

The other two categories are protected as follows: Control-Flow Locking uses the Control Flow Graph (CFG) to determine all indirect calls to functions. When an indirect call is made a lock is set in the memory - which will be unlocked when the function's return is executed. This extra code is inserted at compile- and link time. A key k is located somewhere in the memory which is used to check what state the lock is in. k can have four different values:

- $k = 0$ means unlocked.
- $k = 1$ means indirect call or jump.
- $k > 1$ means return from non-indirectable function.
- $k < 0$ means return from indirectable function.

The value of k will be computed at link time as follows: The list of direct call instructions which refer to this function will be hashed to a value `d`. Then it is determined whether this function may be called indirectly. To do this the function's symbol can be looked for in data declarations or as operands to a non-control flow instruction. If this is true the boolean `indir` is `true`, otherwise `false`. The following formula is used to compute the value of k:

`(indir ? : 0x8000 0000 : 0) | (0x7FFF FFFF & d)`. If `indir` is true this will result in a negative value and otherwise a positive value. This will allow the comparison code to be written in the fewest `x86` instructions possible.

Lock/Unlock operations

Here comes a description of the different lock and unlock operations inserted in the protected application. At direct calls the key will remain unlocked `(k = $0)`. After the call has returned the key will be compared to the specific k value for that function. This will ensure that it was the same function that returned as was called.

At indirect calls the key will be locked `(k = $1)`. After the call has returned the key will be checked if it is less than zero, otherwise it violates CFG. This means that the function it was returned from must be an indirectly callable function.

At indirectly callable function targets there are unlock operations directly after the label. The unlock code will do an unsigned compare with the value `$1`. If k is larger than `$1` it violates the CFG. This ensures that only indirect calls (`$1`) or direct calls (`$0`) will be able to execute further.

The following is the code inserted after labels to indirectly callable functions:

```
indir_callable:
cmpl $1, k
ja violation
movl $0, k
```

Before return instructions a lock-code will be inserted that first checks that k is unlocked (`$0`) and then locks k with the function's specific k value.

To protect against system calls, a verification code can be inserted before each such call. The verification code will verify that `k = $0`, which means the lock is unlocked.

The value of k has to be protected from an adversary. It is important that it can only be changed in the lock- and unlock operations. This can be achieved by using registers that are originally meant for memory segmentation on the `x86` architecture.

An application can have multiple memory segmentations with mappings between them. Bletsch et. al. claims that memory segmentations are not used in modern applications, which means that the segment registers (`%es`, `%fs`, `%gs`) can instead be used for storing k. For this to be secure there has to be no intended or unintended instructions in the applications that reads from or writes to the segment registers.

3.6.2 Implementation details

Control Flow Locking is directly implemented in `libc` via `diet libc`. `diet libc` is a smaller version of `libc` which was simpler for this implementation. A Control-Flow Locking enabled variant of `libgcc` was also produced wich is included as a static library by `gcc`.

The Control-Flow Locking system is implemented in two phases during compilation and linking. The first one rewrites the assembly code. It will align instructions and function entry points on 32-byte boundaries and restrict control flow instructions to 32-byte boundaries according to Instruction-CFI. It inserts lock and unlock operations asserting that no symbols will be called indirectly.

Information about the code symbols, symbol references, lock and unlock operations are noted in the Executable and Linkable Format (Linux) (ELF)-binary under the `.lockinfo` section. This information is then used in the post-link phase to determine all indirect calls and jumps where it will insert additional unlock operations. The second phase will use the information in `.lockinfo` to construct a call graph and identify all the lock and unlock operations. It will use this information to patch the binary with the calculated k-values for each function.

Strengths

- Low run-time overhead

Weaknesses

- There is a lot of manually written code in `libc` which does not have to comply with the compiler-standards for assembly code. For example the

difference between functions and ordinary labels are usually differentiated with a `.L` in the beginning of the label. This convention is not always used by manual programmers.

- Can only be applied to applications that does not use memory segmentation.

Other Thoughts

The protection is only as good as the call-graph that is generated by the tool. This call-graph could be improved if the programmer at a higher level would precise how indirect calls would be made.

Bletsch et.al suggests [4] that this technique could be combined with G-free. G-free could be used to remove unintended gadgets, by rewriting the code, instead of using alignments as in this solution. This could potentially give performance benefits

3.7 ROPScan

ROPScan [25] is an input scanner with a CPU emulator that detects shellcode in input data from network or memory buffers.

3.7.1 Main features

ROPScan is monitoring a process at run-time and emulates code each time input data has entered the process as if it was an ROP-payload. The emulator has a snapshot of the process image and the virtual memory that belong to the process.

Scan input data

Input data is scanned for ROP-payloads by trying to treat each 4 byte as a pointer to the available gadget space. ROPScan needs to search from all possible starting points in the data, since it is not known where in the data a potential ROP-gadget sequence may start. This is done with a sliding window technique that slides one byte for each interpretation try.

If a valid address is found the emulator starts executing from that point to see whether it is the start of a gadget chain. The emulator has a complete snapshot of the process image including the registers and the stack, to be able to do the same computations as in the real execution. It will continue to execute until an invalid address is encountered, an invalid or privileged instruction is encountered or if it hits either the gadget length threshold or the total instruction threshold.

ROP-detection

In order to distinguish incidental gadget chains from intended ROP-gadget chains there is a need to do further checks on the emulated instructions. An ROP-gadget must end in an indirect branch that uses the input data to calculate the branch target. One example is if the gadget ends with the instruction `jmp %eax`, and the value in `%eax` has not been loaded from the input data, the gadget is most probably not a ROP-gadget. Another example is when the gadget ends with a return instruction and the value in the stack pointer is not provided by the input data.

Experiments

Polychronakis et. al. have performed experiments with ROPScan in order to decide upon good thresholds when an ROP-gadget chain should be considered to be found. They have tested in total about 1.86TB of data of varying kinds, including random binary, random ASCII, network streams and PDF memory buffers. They also tested real ROP-payloads in order to determine the minimum gadget-length in a real ROP-payload.

According to their experimentations the threshold for determining if a gadget chain is an ROP-gadget chain should be six valid gadgets.

3.7.2 Implementation details

ROPScan is implemented on top of `Nemu` [24], which is a scanner and a CPU emulator that detects exploiting shellcode in input buffers or network traffic. The authors suggests to implement ROPScan in a shellcode detection system like `ShellOS` [30] instead, to get higher throughputs.

3.7.3 Analysis

Strengths

- ROPScan has accurate detection of ROP attacks, due to its precise detection techniques.

Weaknesses

- The complexity of the algorithm is very high since it relies on emulation of the whole process.
- Cannot scan input data without access to the whole memory image of the protected application.

Other thoughts

Polychonakis et. al. suggests that instead of executing every valid address found in the gadget space, the addresses could be compared to a precomputed list of gadget addresses. The addresses could be precomputed with help of existing gadget discovery tools and save them in a hash table.

3.8 Binary Stirring

Binary Stirring [35] is a binary rewriting technique that randomizes the location of basic blocks.

3.8.1 Main features

Commercial Off The Shelf (COTS)-binaries are stirred with a program that produces a new binary where each basic block has a new random location. It requires no source code or other information than the binary. A wrapper randomizes the binary each time it is loaded. This means that gadgets that are computed on one instance of the binary will not be useful at another running instance of the binary.

The whole process is based on two phases: Static Rewriting Phase and Load-Time Stirring Phase.

Static Rewriting Phase

The Static Rewriting Phase copies the code and data into two copies where the original is treated as data and new as code. The old version is set as non-executable and all bytes are preserved. The old version can therefore be used as data source without the risk of having the code executed. The new version is disassembled into code blocks, which can be randomly stirred in the memory space every time the program starts.

The basic blocks are code-sections which has one entry point and ends with an unconditional jump. Basic blocks can also be divided into smaller blocks by inserting `jmp 0` which is a semantic no-op.

The new version keeps the data references to the old, and need thus not to be recomputed. The data in the new version becomes unreachable garbage code between the executable code and there is no need to detect or remove it.

Load-Time Stirring Phase

In this phase all the basic blocks that were disassembled in the last phase are randomly reordered in the new code section.

After reordering the basic blocks some of the code-pointers will have to be repointed (e.g. those that points to a method dispatch table). Since these repointings will be done very frequently they have to be efficient.

To do this the jump targets in the old version of the code will have their addresses updated to the new version during the first phase, which creates a lookup-table for the new version. Two instructions are then inserted before each jump address to patch the address at run-time with the help of the lookup-table in the old code. These instructions are `cmp byte ptr [%eax], F4h` and `cmovz %eax, [%eax+1]` where `%eax` is the register which holds the address. This code checks if the address starts with the tag byte `0xF4` which is inserted in the first phase to indicate that the pointer is old and needs to be updated.

Position Independent Code

PIC is instructions that computes their own location in the code at run-time and performs pointer arithmetic to compute locations of other instructions and data tables. These computations may be disturbed during stirring and has to be accounted for by the stirring module. The kind of PIC that the prototype handles are ones that pushes the Stack Pointer (SP) on the stack via a call instruction to the next instruction, and then pops it into a register to effectively get its own address. The PIC adds or subtracts a constant to the register to point it to a global offset table and then loads the target address from there.

This PIC is identified by the call instruction that calls the next address in memory. The computation instructions are then replaced by instructions that instead calculates the address after stirring. The new sequence may have to be padded with NOP's to get the same length.

There may be other forms of PIC but the authors have not encountered any in their testing of the prototype.

Statically Computed Returns

In this prototype all return addresses are assumed to be valid since they are pushed on the stacks by call instructions. This means there are no pointer checks before executing a return instruction.

There are a few special cases in the initializer code generated by GNU Compilers where three operands are pushed on the stack to be jumped to by return instructions. The prototype handles this as a special case and inserts jump target checks before performing those returns. The system could also be built with always having return-checks with a performance penalty.

Short functions

If two jump targets are nearby it is generally assumed that they are 5 bytes apart from each other, since 5 byte tagged pointers are used in the lookup-table. In some rare cases there exists targets that are less than 5 byte apart. The rewriter then has to strategically locate the two basic blocks in memory so that the tagged pointers can overlap with the `0xF4` tag being part of one of the target addresses.

For example if the new version section starts at address `0x4000 0000`, the encoding `F4 00 F4 00 04 00 04` can encode two overlapping tagged pointer at addresses `0x0400 F400` and `0x0400 0400` respectively. This strategy can effectively support at least 135 two-pointer collisions and 9 three-pointer collisions.

3.8.2 Implementation details

The disassembler is written as an `IDAPython` script together with `IDAPro`.

The Load-Time Stirring Phase is written as a static library to the target program. It takes place at load-time and needs to be executed before the target code starts executing. This is done by loading the static linked libraries in the right order.

On Windows this is achieved by the system load order which says that static linked libraries are loaded before modules that link them. On Linux the object is compiled as a shared object (`.so`) and injected using the `LD_PRELOAD` environment variable in the address space of the target process.

3.8.3 Analysis

Strengths

- Low run-time overhead (1.6%)
- Can randomize modules that cannot be randomized by ASLR because of missing relocation information.
- Removes up to 99.99% of all ROP and JOP gadgets.

Weaknesses

- Can only randomize code that is available at run-time and not e.g. obfuscated code that unpacks during execution or JIT-compiled code.
- Some of the debugging decisions cannot be made by the disassembler alone and requires an expert's guidance.
- File sizes become in average 73% larger.

3.9 Instruction Location Randomization

Instruction Location Randomization [16] randomizes the location of all instructions post-deployment, to make it impossible to craft gadgets. It requires a Per Process Virtual Machine (PVM) that runs the code via a fall-through table.

3.9.1 Main features

A fall-through table is generated offline which maps addresses to instructions or references. The address that the Instruction Pointer (IP) points to will be looked up in the table. If an address maps to an instruction, that instruction should be executed and the IP increased as normal. If it maps to a reference, the IP should jump to that address.

In the offline analysis all the instructions are placed randomly in the memory. The program is then run on a PVM which uses the fallthrough map to guide through the executable.

Offline Analysis

The offline analysis has three responsibilities: locating instructions, locating indirect branch targets and identifying call sites. This phase uses a few different tools.

At first a recursive descent disassembler (`IDA Pro`) and a linear scan disassembler (`objdump`) are used to locate all functions and instructions. Direct and fall-through are put into an instruction database which is validated by the Disassembly Validator. The functions are noted as sets of instructions and are put in the database.

The indirect branch analysis finds all places in the code that are possible targets for indirect branches. It is hard to find all the Indirect Branch Target (IBT)'s in the program and the prototype for this technique does not cover all of them. A linear scan of the data is done to find all pointer-sized constants that could possibly be an IBT. This works most of the time, but e.g. when C++ uses exception handling this can lead to faults. Sometimes the addresses of IBT's have to remain non-randomized.

Call Site Analysis

The Call Site Analysis looks at all call instructions to see whether the return address can be randomized. This is done because there may be non-randomized instructions left. If a function follows the normal semantics call-return, if it has only standard function exits using the return instruction, if it has only entrances via the function's entry point and if it is not modifying the return value then the return address can be randomized. This works for most of the times, but not for C++ exception handling routines.

After these processing steps the reassembly takes over. Its purpose is to create the rewrite rules to make the program randomized. It goes through all the instructions in the database and creates rewrite rules for them.

3.9.2 Implementation details

The PVM is a per-process virtual machine which uses the rewrite rules to instruct the PC where to go next. It has a cache memory to make it more efficient. It is written as an extension to an existing PVM (Strata) and contains around 1 Kilo Lines of Code (KLOC).

A Postgres database is used during the offline analysis.

3.9.3 Analysis

Strengths

- Impossible to guess where instructions are (it has high entropy). 31 bits of entropy on a 32-bit machine and 63 bits of entropy on a 64-bit machine.

Weaknesses

- Indirect branches cannot be randomized.
- The PVM has an average overhead of 8% and total overhead is at lowest 13%.

Other Thoughts

The prototype does not support randomization within shared libraries. It could easily be extended to do so, but if it is possible it would be better to provide the randomization within the library instead.

On Linux this could be done by using a randomized compiler since the source code is known.

On Windows Instruction Location Randomization (ILR) can be used where the PVM could watch for dynamic library loadings and randomize them at loading time.

3.10 In Place Randomization

In In Place Randomization [22], binaries are rewritten by changing instructions randomly to remove the usability of in advance computed gadgets.

3.10.1 Main features

The binary rewriting is done before running the binary, which makes the program random for every running instance. After rewriting, the program will have exactly the same length, same functions and same basic blocks - but it will not have the same instructions.

This protection is written for `x86` which is a very extensive instruction set which makes it possible to replace instructions with a high probability. There are three kinds of rewrite rules used in this implementation which will be described here.

Atomic Instruction Substitution

Gadgets can consist of both intended and unintended instructions. Unintended instructions occur on x86 assembly instructions since they have different lengths and are not aligned. When an unintended instruction-based gadget is found, one or many of the instructions that constitutes the gadget will be replaced. Replacements can be done with instructions that has the same semantic meaning in the program.

For example the instructions `add r/m32, r32` and `add r32, r/m32` have different op-codes, but will have the exact same meaning when both operands are registers. Another example is to change the order of operands in an instruction. `cmp a1, b1` and `cmp b1, a1` will have different op-codes but the same meaning.

Instruction Reordering

Reordering instructions can be done if the order of the instructions is not important or has no meaning semantically. This rewriting technique can break an adversary's assumption of both intended and unintended gadgets' impacts.

Dependencies The order of instructions in a program is decided by the compiler, and is often a combination of many factors like instruction cycle cost and optimization techniques. Instructions can often be re-ordered with the same semantic meaning, but at the cost of these factors. To reorder instructions in a basic block much information has to be calculated about the dependencies between the instructions.

A dependence graph is constructed for each basic block. Each instruction is derived two sets which contains the registers that are used respectively defined by the instruction. E.g. two instructions will have a dependency if one of them defines a register that the other uses, if one of them writes to memory and the other reads from memory or if one of them writes a flag which the other one reads. The memory dependency rule is very conservative because it is hard to statically analyze relative memory addresses. In this

prototype no effort is done to compute if they are actually writing to the same memory location.

Callee-saved Registers Callee-saved registers on the `x86`-architecture are registers that the function-callee has the responsibility to preserve after the function call. Normally these registers are needed by the function callee, and the normal course of action is to push the register-values on the stack in the beginning of the function and to pop them in the end of the function. Depending on the platform and the compiler there may be around four different callee-saved registers. The compiler chooses the order to push and pop them in an arbitrary way, which makes it an easy target for In Place Randomization (IPR) to reorder. As long as the registers are pushed and popped in the reverse orders IPR can reorder them freely.

There are some special cases where the push- and pop-sequences can occur multiple times in a function, which needs to be taken care of as well. If a function has many exit points there will be pop-sequences at every exit point. A function can have pop- and push-sequences in other places than at the start and end if the registers only need to be preserved for a certain execution path.

This technique will destroy the assumption of many gadgets that only contains one or a couple of register pops from the stack and end in a return which are fairly common.

Register Reassignment

Different registers can be switched randomly if it is possible to interchange them for a certain path.

Live Ranges To determine if registers can be switched IPR has to calculate every register's live range. A live range for a register is the set of program points in the CFG of the program where the register is live. The register r is live at a program point p if and only if there is a path from p to a use of r that does not go through a definition of r. To calculate the live ranges for each register a live-variable analysis algorithm is performed.

A register will have many disjoint live ranges during program, and the different registers will have overlapping live ranges. To determine if two can be interchanged for a certain execution path the live ranges must match or the shorter register's live range must be extendable to match the other register's live range. For a live range to be extendable it must not overlap another of its own live ranges. If a register can match its live range with two or more registers they can all be interchangeable during the live range.

Non-interchangable Registers In some cases the registers cannot be interchanged, e.g. the one-byte instruction `mov` always uses two specific registers, and there is no corresponding one-byte instruction that can replace it with other registers. If a compiler is not following the standard calling conventions for a private or static function it may use registers in other ways than expected. To detect this the `IDAPro` uses an algorithm to thoroughly go through every function in the program and the calls to the functions.

This technique will break the assumption gadgets have on the registers they are trying to manipulate. It will also destroy contingent unintended gadgets. The reason why this technique cannot randomize all gadgets can be the following: the gadget is part of data that is used by the program, the gadget is part of code that could not be disassembled or the gadget was not affected by the transformations used.

3.10.2 Implementation details

`IDAPro` is used to disassemble the Portable Executable (PE)-files to extract instructions that are used. Functions and basic blocks are extracted from the code which are accurately assembled. For extracted code with low accuracy no assumptions on functions or basic blocks are made in order to prevent destructive modifications on code that is misidentified.

Instructions are then converted to a new internal representation that holds more information about the instruction, such as which registers are used implicitly and which registers and flags are read by the instructions. The instruction sequences that could potentially be used as gadgets, both intentionally and unintentionally, are searched for - and rewritten according to the rewriting rules. If instructions that have to be moved from their original location includes an absolute address, the corresponding entry in the `.reloc-section` has to be changed accordingly.

Pappas et.al. suggest that the randomization phase is made offline, because it has high complexity (in the order of minutes). A pool of different randomized instances of the program can be shipped with the program together with a loader that loads a new one each time the program is loaded.

3.10.3 Analysis

Strengths

- Lightweight at run-time (almost no overhead).

Weaknesses

- Only about 80% of gadgets are removed with this technique.
- The randomization algorithm has high complexity.
- File size overheads become big when a big pool of program instances is attached.

Other Thoughts

The authors suggest that this technique could be combined with other defense-techniques since it is so lightweight. It could be combined with other techniques that rewrites the binary files before running other randomization techniques.

The authors of Binary Stirring criticises this technique because it is hard to deploy - each randomized copy has to be separately distributed. A distributed copy

could although come with a pool of different randomized version and a loader then picks different versions each time the program is loaded.

3.11 kBouncer

kBouncer [23] detects ROP-characteristics at run time by monitoring the hardware-featured Last Branch Record (LBR)-stack via a kernel module.

3.11.1 Main features

kBouncer does LBR-verifications each time the Windows API is called.

LBR-stack

The LBR-stack is a hardware feature that saves the branch and target addresses of the 16 latest executed branches[1]. Saving the branches in LBR gives only a small overhead since it is implemented in the hardware. The downside with LBR-registers is that monitoring them can be done only from kernel-level code.

LBR-verification

LBR-verification includes examining the LBR-stack for ROP- and JOP-characteristics. kBouncer does LBR-verification each time a Windows API function is called. Windows API functions are almost always used in ROP-attacks to in their turn do system calls to change the current system's properties. E.g. ROP-attacks can try to change the current memory page into executable memory. It is necessary to verify the LBR-stack at the time the Windows API function is called and not when the actual system call is done, because between those points almost the whole LBR-stack is overwritten with preparations for the system call.

To prevent ROP-attacks from doing the system call directly instead of going via the Windows API functions, kBouncer has implemented a scheme to force it to. When a Windows API function is called the kernel module is notified to do an LBR-verification. At the same time the kernel module also writes a checkpoint that confirms that the code has passed through the API function. When a system call is invoked the checkpoint is verified by the kernel module, and if it is valid, it is deleted from memory.

The kernel module then passes execution to the system call. If the checkpoint is not valid, a violation is reported. This enforcement scheme will always work on legitimate code because the native system call API is not exposed to user-level programs, and has to be done via the Windows API.

Keeping the LBR-verifications to the minimum needed to detect a ROP-attack is important for this defense, since the overhead would be too large otherwise.

Characteristics

The two characteristics that are verified in an LBR-verification are non-call Preceded Gadgets and Gadget Chaining.

Non-call preceded Gadgets kBouncer takes advantage of call-return pairs. The general rule is that for every call instruction there will be a return instruction

[1] Note that this number is CPU-dependent

that moves the execution to the address right after the corresponding call. If a return branches to an address that is not preceding a call instruction kBouncer detects a possible ROP-attack. The LBR-verification checks that each return branch in LBR is preceded with a call instruction. If kBouncer detects such a return-branch then a violation is reported and the program is stopped. kBouncer does not check that the pair of call and return are right, it only checks if there is any call-instruction.

Gadget Chaining kBouncer also looks for gadget chaining when doing an LBR-verification. A gadget chaining is a chain of ROP and/or JOP gadgets which each consists of a specified number of instructions or less. The gadgets are all chained via indirect branches. In this implementation gadgets can also be fragmented through branches, as long as they are shorter than 20 instructions in a sequence.

Pappas et. al. show in experiments that in this implementation there is almost no risk that normal execution will be detected as gadgets, even if the LBR-verifications would be done much more frequently. Their experiments show that after running seven different popular programs on Windows and doing over 97 million LBR-verifications - the maximum length of a gadget-like chain is nine, which is far from 16 which is their maximum limit.

3.11.2 Implementation details

kBouncer consists of three different components:

Offline Gadget Extraction and Analysis Toolkit

This component does a disassembling analysis of the binary to calculate the offsets of every call-preceded gadget and all other gadgets. These offsets are saved in two different hash tables.

A User-Space Thin Interposition Layer

This interposition layer is implemented on top of the `Detours` framework, which provides capabilities of intercepting library calls on the Windows platform. When the target program is loaded this component calls the kBouncer kernel module to enable the LBR feature on the CPU. It then registers hooks to the selected Windows API functions which will invoke the LBR-verification each time those functions are called.

When the LBR-verification is invoked this component notifies the kernel module in order to do the verification. This component is communicating with the Kernel Module via control messages over a pseudo-device that is exported by the kernel module (available via the `DeviceIoControl` API function).

Kernel Module

The Kernel Module is responsible for enabling and disabling the LBR functionality, the LBR-verification and writing and verifying checkpoints before letting a system

call executing. To enable and disable LBR the instructions `rdmsr` and `wrmsr` are available.

Identifying non-call preceded gadgets is done by looking at the branch target of the last branches to check whether they are preceding a call instruction. The number of consecutive targets that point to gadgets are counted backwards from the most recent branch to identify gadget chaining. Return-targets are looked up in the call-preceded hash table and any other branch-target is looked up in both hash tables.

The API-call verification has not been implemented in the prototype since it is not possible to do changes to certain critical data structures such as the System Service Descriptor Table in 64-bit Windows 7. It would be easy to implement in e.g. Linux.

Pappas et. al. proposes a list of 52 different Windows API functions that should be protected in the kBouncer implementation. Extending the list is not needed because of the extra overhead for no extra security.

3.11.3 Analysis

Strengths

- Low performance overhead (1.00%).

Weaknesses

- Does not capture call-preceded gadgets (intended or unintended).
- Requires a kernel module.
- For processors with a small LBR-stack an adversary could ensure that the last branches looks valid, or are call-preceded.

3.12 ROPGuard

ROPGuard [13] provides run-time detection against six different ROP-characteristics. These checks are done only before sensitive system calls are made, which makes it very lightweight and easy.

3.12.1 Main features

Fratric defines critical functions as functions that are used in ROP-attacks that can e.g. change the properties of memory or creating a new process. These functions are often used to open up possibilities to run arbitrary code in the target process and they are called via the ROP-gadgets. By doing ROP-verifications only at times when critical functions are invoked the protection will become very lightweight. The different checks implemented in ROPGuard include the following:

- Checking the stack pointer if it is inside the boundaries of the current thread.

  In ROP-attacks the adversary can control the stack and in particular the stack pointer. To check whether the stack pointer is manipulated the Thread Information Block (TIB) in Windows can be examined to get information about the current function's stack frame. The TIB is undocumented which makes it not a 100% reliable. If the stack pointer is not inside the boundaries of the stack frame there is suspicion of an ROP-attack.

- Looking for the address of critical function on the stack to see if it was entered via a return rather than a jump or call.

  Gadgets that ends in returns, will leave the address of the return target just above the stack pointer when entering the return target. To check whether a critical function was entered via a return-branch, the bytes above the stack pointer are saved from being modified, and then a check is run to see if they are the same. If a return N instruction is used in the gadget the address will be $(N+1)*4$ bytes above the stack pointer. If the critical function was entered via a return-branch an ROP-attack can be suspected.

- Checking the return address to see that it is executable and that it is preceded by a call.

  At the start of a critical function its return address can be checked for different properties.

  1. It has to return to an executable code section. If it does not return into an executable code section it can be suspected that the critical function's purpose is to change the property on the code section to become executable.
  2. It has to be a call-preceded return target. If it is not a call-preceded return target we can suspect that the critical function was targeted via a return branch.
  3. Further verification on the call can done. If the return is call-preceded we can check that the call is actually referring to this critical function.

E.g. if the call is of the type `call [eax]` - the address in the `%eax` register is compared to the address of the critical function.

This verification can be unreliable e.g. if the call was followed by a jump to the critical function. It is possible to detect most of these cases - but not all.

- Checking the stack frame pointer.

  The stack frame pointer can be used to check the return addresses of the functions that called the critical function, in a similar way as described in the last check. The stack frame pointer can also be checked to actually be on the stack and that it is below the stack pointer.

  This verification works only if the program is compiled to use the `%ebp`, or any other consistent register, as a stack frame pointer. This check is not enabled by default, but ROPGuard can be extended with a white-list of programs that are known to be compiled with the stack frame pointers.

- Simulating the execution flow.

  The code execution after a critical function can be simulated in ROPGuard. The instructions that changes the stack pointers are simulated and the rest are ignored. When a return instruction is encountered the same checks as before can be applied on this return statement; that the return target is executable and that it is call-preceded. The simulation proceeds until it has reached a specified limit or until it encounters a non-return branch instruction. This simulation makes it possible to detect if the critical function is part of a longer gadget chain.

  The simulation could be extended to also follow indirect call- and jump instructions to detect gadgets that are not only return-based.

- Function specific checks.

  For some critical functions specific checks can be done for calls that will potentially harm the system. Fratric proposes two different specific checks, and more can be added.

  VirtualProtect `VirtualProtect` can be used by an adversary to change the memory properties of the stack to be executable. This is a common way to do an ROP-attack and is detected by checking the parameters of the function. The parameter `lpAddress` tells what address the pages start at that are going to be changed, and `flNewProtect` tells what kind of property the memory will have after the call.

  LoadLibrary Loading a library over SMB can be prevented by checking the parameters of the function LoadLibrary. There are also similar functions that loads libraries.

3.12.2 Implementation details

ROPGuard is implemented as a library (`.dll`) and a starter process. It can be started with already running processes and with newly started processes. When

ROPGuard is started it starts to parse a configuration file which defines which checks should be enabled. It then injects the library into the process and inline-patches all critical functions with validation code. The validation checks uses the library to execute the validations.

The starter executable starts the process in a suspended state, injects the library and then resumes the process. On Windows XP and earlier versions the starter process instead patches the newly created processes with an infinite loop. During execution of the infinite loop the starter process injects the library and then removes the loop.

3.12.3 Analysis

Strengths

- Lightweight implementation with minimum amount of detection code as possible.
- Low performance overhead (0.48%).

Weaknesses

- Prototype only detects return-based, and not jump-based attacks.
- Does not detect attacks which are not using system calls.

Other thoughts

The prototype only detects return based attacks. A future version may include ability to detect jump based attacks, with gadgets ending with and containing jump instructions.

3.13 Marlin

Marlin [15] is a memory randomization technique. All functions in a program are randomly distributed in memory when the program is loaded.

3.13.1 Main features

The process of Marlin is divided in two phases; Swapping phase and Jump-patching phase. Marlin has a high entropy. If a program contains n different functions, the number of different permutations of the binary is $n!$. Every time an application is loaded by the program loader the memory space is randomized.

Swapping phase

A function is identified by its symbol. A simple swapping phase is done, where all symbols in the binary are iterated. For every symbol a random other symbol is chosen, and the functions succeeding are replaced with each other.

Jump-patching phase

After the swapping phase a jump-patching phase is done. This phase patches all relative and absolute branches that have been changed due to the swapping, to point to their new locations in memory.

3.13.2 Implementation details

Marlin is implemented as a customized program loader. The `mmap` system call is issued by the loader to map a binary to its process' memory space. Marlin permutes the order of which the system calls are made for different symbols in the binary.

3.13.3 Analysis

Strengths

- Marlin is based on a simple algorithm.
- The only performance overhead is during the loading of the program (4 seconds).
- Has high entropy to make it hard to brute-force.

Weaknesses

- If the symbols are stripped from the binary, randomization has to take place at another granularity (e.g. at basic block level, as in Binary Stirring).
- There are many difficulties with this approach that has not been mentioned in the paper: e.g. data references and Position Independent Code.

3.14 Control Flow Integrity and Randomization for Binary Executables

Control Flow Integrity and Randomization for Binary Executables [36], called CCFIR, is a CFI implementation that enforces integrity via springboard code section.

3.14.1 Main features

CCFIR analyzes and rewrites binary files via their relocation table. Each module in the program gets a springboard which encodes target restrictions via code alignment. It locates all indirect control transfer instructions, indirect jump, indirect calls and returns, and ensures that their targets are legal by rewriting them. A check is performed in the springboard before each such transfer to see that they are valid.

The relocation table in a binary is a list of pointers created by the compiler and assembler. Each entry in the list is a pointer to an address that needs to be changed when the loader relocates the program. The relocation table is not always present, but if the binary is compiled with ASLR it has to, because the program will always get run in a new random place in memory each time it is loaded.

Springboard

Springboard memory areas are enforced to have the address' 27th bit set to `0`. The memory is divided in pieces of 128MB, with data in the odd sections and their respective springboards in the even section. This policy makes it easy to check if an address belongs to a springboard.

The springboard contains stubs that the original code will jump to. For every valid indirect jump in the original code there is a stub in the springboard, which contains a direct jump to the original target. The indirect jump in the original code is replaced with a direct jump to the springboard-stub.

The springboard is divided into different regions which makes the valid targets distinguishable. There are three different regions which contains different kinds of stubs - indirect jumps to function pointers, return addresses into normal functions and return addresses into sensitive library functions (e.g. `system()` in `libc`). Depending on which region the stub lies in the type of indirect transfer target can be determined.

The springboard is aligned according to SFI, which is a technique to enforce control flow integrity. There are three main properties of the control flow integrity:

1. Indirect call and jump instructions are enforced to only jump to function pointer stubs in the springboard. They can only jump to targets that are 8-byte aligned but not 16-byte aligned. The alignment makes sure that no unintended instructions are executed, but only the ones intended. There are no function pointer stubs that directs to sensitive functions in the springboard, which makes it impossible to do an indirect call or jump to a sensitive function.

2. Return instructions in normal functions cannot jump to sensitive return address stubs. They must be 16-byte aligned with their 26th bit set to 0. This enforcement makes RILC and ROP very limited because they often rely on returning into system functions.
3. Return instructions in sensitive functions can jump into any return stub in the springboards, but must be 16-byte aligned. Since all return instructions must be 16-byte aligned many ROP-gadgets are removed that rely on jumping into the middle of a function. All these properties are ensured by placing code that checks this before every indirect jump/call and return instruction.
4. Lastly the entries in the springboard are also randomized at load-time to make it harder to guess the addresses of function pointers and return address stubs.

3.14.2 Implementation details

Implemented in three major modules; BitCover, BitRewrite and BitVerify.

BitCover Dissasembles the PE-file and identifies all indirect call, jump or return instructions and potential indirect control transfer targets.

BitRewrite Inserts springboards, redirects pointers to the new boards, infers checks of the validity of branches and randomizes the springboard.

BitVerify Verifies that the binary conforms to the security policies.

CCFIR is written in `C++`. BitCover is built upon `Udis86` which is an open source disassembler library.

3.14.3 Analysis

Strengths

- Low average overhead (3.6%).
- Most of the gadgets are removed and 100% of the gadgets are invalid due to the CFI.

Weaknesses

- File sizes increase about 30%.

3.15 Control Flow Integrity for COTS libraries

Control Flow Integrity for COTS libraries (CFICOTS) [37] is an implementation of CFI. CFICOTS disassembles binaries to rewrite them and to insert a new copy of the code segments that enforces the CFG.

Disassembly

CFICOTS uses two different disassembling techniques, linear and recursive. The linear disassembly starts at a code segment, looks at instructions one by one and decodes them. It does not take account for different blocks or gaps in the program and may therefore give erroneous results.

The recursive disassembling technique takes the program's CFG into account, and performs a depth-first search in it. Recursive disassembly will disassemble more code than linear since it will follow all control transfers, and disassemble from each possible entry point. It will not disassemble code that are only reachable via ICF targets, which is why more information is needed. The relocation information in a binary will provide some of the information, if present.

CFICOTS starts by doing a linear assembly on the whole binary. It then corrects all potential errors by doing a recursive disassembly followed by an error correction. To correct errors it has to find disassembled code that is actually a gap. To do that it will look for three different properties: invalid opcode, direct control transfers outside the current module and direct control transfer to the middle of an instruction.

When one of these properties are found it is sure to have found a gap, and starts looking for the start and the end of the gap. To find the start, it simply walks backwards to the first unconditional control flow transfer. To find the end it walks forward to find the first ICF target. All ICF targets are found during the static analysis.

Static analysis

A static analysis is done to find all possible ICF targets. Zhang et. al. defines five different categories that ICF targets may fall in.

1. Code pointer constants are code addresses that are computed at compile-time. It is hard to distinguish a code pointer constant and other types of constants in code. There are two constraints that can be applied, and if they are true the constant is considered a code pointer constant.

   If the pointer falls within the range of the code addresses in the current module and if it points to an instruction in the disassembled code (i.e. it does not fall in the middle of an instruction). If it is a shared library being disassembled, the constant will represent a memory offset. In that case the offset is checked to be valid from the base of the code segment.

2. Computed code addresses are code addresses that are computed at run-time. Computing code addresses by doing arbitrary arithmetic on the pointer is very uncommon in the code Zhang et. al. has studied. The only cases where

they have found such are in jump tables and C++ exception handling. To find jump tables a three step process is done:

The first step is to find function boundaries, since most jump tables are intra-procedural and all the ICF transfers are done within the same function. To find the function boundaries a conservative method is done, where exported function symbols are treated as function boundaries.

The second step is to find all data dependencies that computes the jump target in the jump table. To do that it walks backwards using the CFG from the indirect jumps.

The third step is to enumerate all possible values of the jump target. If an enumeration falls in the current region a computed code address is found.

3. Exception handling addresses are code addresses that are used to handle exceptions. These addresses are listed in ELF-binaries in the sections `.eh_frame` and `.gcc_except_table`.
4. Exported symbol addresses are exported function addresses. These addresses are listed in ELF-binaries in the section `.dynamic`, which is the dynamic symbol table.
5. Return addresses are code addresses following a call instruction. They are easily found during the disassembly.

Instrumentation and regeneration of binary

The CFI-instrumentation of the code is done in the disassembled assembly form. After instrumentation the code will be assembled into a new binary using the Gnu Assembler (`gas`).

The new binary's code will be extracted and injected into the old binary under a new section. This new section will be the code-section, and the old code-section will be set to non-executable. This ensures that the new code will be able to read data from the old code. The ELF header, relocation information and dynamic symbol section need to be updated as well.

The instrumentation consists of a few different code modifications which are described here.

Address Translation The new code will need to have its addresses updated to point to the new code. Since all code address constants cannot be distinguished from other constants, they cannot be changed directly. An address translation table, called Module Translation Table (MTT), is used which consists of pairs between original addresses and labels in the new code. E.g. the original address `0105` will be represented by the label `L_0105` in the new code. All direct branches in the new code can be translated to a direct call to the right label. All indirect branches will instead jump to a trampoline code, which will translate the destination to the right label.

All valid ICF targets that have been calculated in the disassembly will be put in a hash table which maps original addresses to code that transfers control to the right label. If the address is not found in the hash table the address is not a valid ICF target, and the process will stop executing.

To be able to translate ICF transfers between different modules two tables are needed. To locate the right module a Global Translation Table (GTT) is used, which maps ICF targets to the address translation routine in the right module. Since any two modules on a 32-bit Linux system must be apart by at least 4KB, only the 20 leading bits needs to be used. This means that the table can be implemented as an array with only 1M entries.

Linker and loader modification To keep the GTT up to date during dynamic linking, when modules are loaded and unloaded, the dynamic linker needs to be modified. Zhang et. al. have modified `ld.so`, which is the standard dynamic linker/loader in Linux systems. When a module is dynamically loaded that is referenced by the program, or when the application uses the system calls `dlopen` and `dlclose`, the loader will also insert the right information in the GTT.

When a binary is rewritten in CFICOTS the entry point of the binary may change. `ld.so` makes use of the entry point when it is invoked to load a program. The loader needs to be modified to compensate for the change in the entry point.

`ld.so` is also changed in a third way, because it uses a return instruction for lazy symbol resolving. That means it computes a target address of a symbol, pushes it on the stack and performs a return instruction to jump there. CFI does not allow return branches to exported symbols, and this will therefore need to be changed to a jump instruction.

Signals Signals need to be handled separately since they will still specify locations in the old code segment when doing a branch. To do this CFICOTS intercepts `sigaction` and signal system calls and stores the address of the signal handlers that are supposed to handle this signal. The signal handler argument to the system call is then changed so that the signal handler in the new code section will be invoked.

Optimizations Some optimizations are implemented in order to make the overhead lower. Branch predictions can be optimized by writing the address translation code for calls and returns so that the processor can predict return branches by changing the return address that lies on the stack.

In some cases the address translation can be avoided:

- Jump tables can be rewritten to use a jump table with the new addresses, instead of translating the addresses each time.
- PIC includes calls to `get_pc_thunk`, which has the responsibility to load the Program Counter (PC) into a general use register. This can be replaced with a call/pop-pair to remove the address translation for that function.
- Return targets can be profiled in order to see which are used most. To remove the address translation on those places a comparison can be done with the known return targets. If they are equal, the return can be replaced with a direct branch.

Normally the translation code needs to save the e-flags and registers to not disrupt the normal execution. If the instrumentation detects that there is no instruction that uses or defines the e-flags or some registers, this can be skipped.

3.15.1 Implementation details

CFICOTS is written for 32-bit Linux systems which use the ELF-binary type. The linear disassembly is written on top of `objdump`. `Gas` is used as the assembler to generate the new binary file. `Objcopy` is used to inject the new sections to the binary ELF-file.

Strengths

- Eliminates up to 95% of all gadgets.
- Low performance overhead (4.29%).

Weaknesses

- Large binary overhead (139%).
- Cannot reliably disassemble dynamically changing or obfuscated code due to its static disassemble method.

3.16 ROPecker

ROPecker [7] is a run-time ROP-detection which monitors instructions and analyzes the latest branches taken by examining the LBR-stack.

3.16.1 Main features

ROPecker is divided into two parts; an offline phase where the binary and its libraries are put in a pre-processor for statical analysis and a run-time phase which detects the ROP-attack at runtime.

The offline phase analyzes the binary and its libraries and creates a database with all the possible gadgets. It starts from the first byte and disassembles a given number of instructions, (for example six). If the sequence ends with an indirect branch instruction and does not contain a direct branch it can be seen as a possible gadget.

It then analyzes all the instructions prior to the branch instruction to determine what impact it can have on the stack and registers. This analysis is very alike the analysis done by programs that are creating gadgets from a given binary or library.

The run-time phase analyzes the execution in a kernel module and stops the execution if an ROP-attack is found. It will start an ROP-check based on two different triggers which will be described:

Sliding Window A page window is set and all code outside this window is set as non-executable. If code outside the window is tried to be executed a page fault exception will occur, which will trigger the ROP-detection check. If the check does not fault it will slide the window to include the new page. A page size of between 8KB and 16KB is recommended by the authors.

Risky System Calls Risky system calls are those system calls that let the user disable the non-executable property outside the window, (e.g `mprotect` and `mmap2`). To protect against this a trigger to the ROP-protection will be placed before every such system call. The system call table is modified to insert calls to the detection code.

ROP-detection

The ROP-detection check first checks the LBR-stack to see which branches has been taken recently. The LBR contains the source- and target addresses for the last branches taken. To detect a gadget chain in the LBR it starts with the most recent branch. It checks whether it is an indirect branch and if its target address targets a possible gadget. Both of these information are found in the database created in the offline phase. When one of the two conditions fails it records the number of gadgets found.

It then checks the predicted future branches. For return-based branches the pre-processor can calculate what the branch target will be by following the stack and the stack pointer. This information will be stored in the database which the kernel module can look up at runtime.

For jump-branches the branch targets have to calculated at runtime. To do this an emulator environment is loaded in which a snapshot of the current process' memory is copied. The instructions preceding the jump instruction is emulated in this environment to calculate the branch target. If the number of gadgets in the LBR-analysis and the future branches analysis combined is higher than a specified threshold ROPecker will detect the ROP-attack and abort execution.

3.16.2 Implementation details

The preprocessor is a very simple disassembler and can thus be implemented using an existing linear-sweeping disassembler such as `objdump` or `diStorm`. The runtime analyzer part is implemented as a kernel module for x86 32-bit Ubuntu 12.04 with kernel `3.2.0-29-general-pae`.

A kernel module is needed to be able to read the LBR.

To intercept page fault exceptions, that are used in the sliding window, and risky system calls the kernel module inserts hooks in the Interrupt Descriptor Table (IDT). For the sliding window technique the memory outside of the window is set as Never eXecute (NX), that is a memory table permission. When a memory address with the NX-bit set is executed a page fault exception is thrown.

The exception will be caught by the ROPecker kernel module which will check where the exception was thrown. If the exception was thrown outside the window but in otherwise valid memory location the ROP-detection will be started. If the detection turns out negative the window will slide to its new location and execution will continue.

The emulator is used when ROPecker is calculating jump-branch targets. To make the emulation more space- and time-efficient, a technique called read-on-write is used. In this technique the virtual address space of the process is used by the emulator and set as read only. The emulator can freely read the virtual address space. When the emulator needs to write to the address space, it will instead create a mapping table to another place in the memory and place the new values there. Every time a read or write is done the emulator will first check the table mapping.

3.16.3 Analysis

Strengths

- Lightweight implementation which has low performance overhead (2.6%).

Weaknesses

- Cannot protect against Gadget Gluing Attacks. Gadget Gluing Attack is where two gadgets are chained together via a direct branch instruction. Since the gadget chain detection stops when encountering a direct branch instruction, it will not detect these kinds of gadgets.
- If only long gadgets are used in a ROP-attack it will be hard for ROPecker to detect it. Since there is a number of gadget instructions threshold that must be met for it to be counted as a gadget.

If ROPecker would be combined with the gadget-removal tool G-free, the sliding window could be significantly larger and the overhead even smaller.

3.17 Zero-sum Defender

Zero-sum Defender [17] is a simple instruction monitoring technique, which inserts code to count all call - return pairs in a program during compilation time.

3.17.1 Main features

Zero-sum Defender takes advantage of the assumption that in normal program flow all call instructions will be followed by a return instruction. In a ROP-attack the number of return instructions executed will be higher than the number of call instructions. To check that this assumption is correct, Zero-sum Defender counts every executed call instruction and compares it with the number of executed return instructions.

To do this a, counter is used, that is incremented at every call and decremented at every return. After each decrementation the counter is tested for being negative. If the counter is negative, there has been more returns than calls, and a ROP-attack is detected.

3.17.2 Implementation details

Zero-sum Defender is implemented as a transformation pass in the LLVM compiler infrastructure. In the middle-end of the compiler, the assembler code is rewritten by inserting the detections at the start and end of all functions. In total five instructions are added to each function. The counter can be placed in an array with a run-time random index. The longer array that is used, the larger entropy is gained. The index can be initialized randomly in the start of the program.

There is a problem with multiple threads that uses the same counter, which will make the value of the counter incorrect. To deal with this the counter has to be made thread-local. If an array is used, the whole array has to be thread-local.

A more detailed description of the implementation is given in section 4.3.

3.17.3 Analysis

Strengths

- Lightweight implementation with low performance overhead (1.67%) and low binary overhead (4.5%).

Weaknesses

- The authors have not mentioned special cases like non-local gotos which are common in C++ exception handling.
- Does not detect the use of unaligned gadgets.

Other

The authors claim this protection can defend against JOP-attacks, but they do not specify how.

3.18 Summary

All the protections presented in this chapter have been summarized in three different tables.

Table 3.1 gives a general presentation of the protections. The first column presents what kinds of ROP-gadgets it can protect against. *Ret-based* means that it only protects against ROP-attacks with gadgets ending in a return instruction, and *All* means all possible gadgets. The second column tells if the protection requires the source code of the protected application. The third column tells if the protection needs to rewrite the binary of the application.

Table 3.2 presents the overheads the protection gives on the original application. The first column presents the binary overhead. This is the size increase of the binary files that are generated by the protection. If the protection does not make changes to the binary files, this figure is missing. The second column presents the run-time overhead. This is the extra time a protected application takes to run, compared to the original application.

Table 3.3 presents the effectiveness of the protection. Its only column shows how many percent of the gadgets are removed and/or detected by the protection.

All the information presented in these tables are taken from the protections' authors and are their own results. The information has to be interpreted with that in mind. Not all the authors have presented experimental results, which is why some figures are missing in the tables.

[2]Scans input data with a speed of 120Mbit/s.

[3]Average of the SPEC CPU 2006 benchmark.

[4]From 0% and up. If a pool of different binaries is used, up to a several hundred percent.

[5]4.29% for C programs, 8.54% for C++ programs (due to exceptions-handling).

[6]All ROP-attacks were detected.

[7]Of which 9.5% is eliminated and 67.4% is semantically broken.

[8]One ROP-attack is tested which is detected in four different ways.

[9]All ROP-attacks were detected.

[10]Only one attack is tested, which is detected.

	ROP Types	Source Code	Binary Rewriting
CFI	Ret-based	no	yes
DROP	Ret-based	no	yes
G-Free	All	yes	no
Return-less Kernel	Ret-based	yes	no
ROPDefender	Ret-based	no	yes
CFLocking	All	yes	no
ROPScan	All	no	yes
Binary Stirring	All	no	yes
ILR	All	no	yes
IPR	All	no	yes
KBouncer	All	no	yes
Marlin	All	no	yes
ROPGuard	Ret-based	no	yes
CCFIR	All	no	yes
CFI for COTS	All	no	yes
ROPecker	All	no	no
Zero-sum Defender	Ret-based	yes	no

Table 3.1: Overview of the protections. ROP-gadget types, requires source code and rewrites binary files.

	Binary overhead (avg)	Run-time overhead (avg)
CFI	8.00%	16.00%
DROP	-	530.00%
G-Free	25.90%	3.10%
Return-less Kernel	9.40%	15.15%
ROPDefender	0.00%	217.00%
CFLocking	-	4.60%
ROPScan	-	Depends on input [2]
Binary Stirring	73.00%	1.60%
ILR	104MB [3]	16%
IPR	Depends on usage[4]	0.00%
KBouncer	-	1.00%
Marlin	-	~4 seconds
ROPGuard	-	0.48%
CCFIR	30.00%	3.60%
CFI for COTS	139.00%	4.29% [5]
ROPecker	19MB	2.60%
Zero-sum Defender	4.50%	1.67%

Table 3.2: Comparison of the overheads of the protections.

	Gadgets Removed/Detected
CFI	100.00% of ROP
DROP	100.00% of ROP
G-Free	100.00%
Return-less Kernel	100.00% of ROP
ROPDefender	100.00% of ROP
CFLocking	100.00%
ROPScan	- [6]
Binary Stirring	99.99%
ILR	96.00%
IPR	76.90% [7]
KBouncer	93.60%
Marlin	-
ROPGuard	-[8]
CCFIR	100.00%
CFI for COTS	92.68%
ROPecker	- [9]
Zero-sum Defender	- [10]

Table 3.3: Comparison of the efficiency of the protections.

Chapter 4

Detailed Analysis of ROPDefender and Zero-sum Defender

4.1 Introduction

A part of the results of this thesis is a deeper analysis and implementation of a couple of known protections. This chapter will present a deeper study of the ROPdefender (presented in section 3.5) and ZeroSum (presented in section 3.17) protections.

The motivation of doing a detailed study of existing protections was to understand them better, do experimental tests to compare the results, and to become inspired for new ideas. These two protections are chosen partly because of their ROP-detection techniques and partly because of the frameworks they build upon. They are both relying on fundamental properties of computer programs, which they use in order to detect abnormalities. These properties are very easy to examine, which makes them interesting for using in other protections.

The frameworks they build upon, PIN and LLVM, are both very open and extendable frameworks. These frameworks can be used to implement other protections involving instruction monitoring or CFI, which makes them interesting to look into and test.

The first two sections in this chapter, 4.2 and 4.3 describe the implementation of the ROPdefender and ZeroSum respectively.

Section 4.4 presents an experimental evaluation of the two implementations.

4.2 Implementing ROPDefender

ROPDefender is implemented as a Pintool for the dynamic binary instrumentation tool PIN by Intel. Pintools are written in C++. PIN can be used to instrument and analyze programs during run-time. PIN loads a compiled program binary into memory and runs the code with a JIT-recompilation. When the code is recompiled it can insert new instructions that are described in a Pintool. Callback routines can be specified in a Pintool, that will be called when certain events occur.

ROPDefender is a Pintool that will intercept call instructions and return instructions. The callback routines that are defined in ROPDefender are

`push_ret_address(VOID *r_ip)`

and

`check_ret_address(VOID *r_ip)`

The first one will be called when a call instruction is executed, and the second when a return instruction is executed.

4.2.1 Registering callbacks

First the callback routines are registered in the Pintool.

```
VOID instrument_instruction(INS ins, VOID *v)
{
        if (INS_IsCall(ins)) {
                INS_InsertCall(ins, IPOINT_TAKEN_BRANCH, (AFUNPTR)
                push_ret_address, IARG_RETURN_IP, IARG_END);
        } else if (INS_IsRet(ins)) {
                INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)
                check_ret_address, IARG_BRANCH_TARGET_ADDR, IARG_END);
        }
}
```

The PIN API offers functions to determine if an instruction is a call instruction or a return instruction.

The callbacks are inserted with the function:

`INS_InsertCall (INS ins, IPOINT action, AFUNPTR funptr,...)`

The callback routine for the call instruction is inserted with the parameter:

`IARG_RETURN_IP,`

which means that the return address of the function call will be given as a parameter to the callback routine. The other callback routine is inserted with the parameter:

`IARG_BRANCH_TARGET_ADDR,`

which will give the branch target address as parameter to the callback routine.

4.2.2 Callback routines

ROPDefender holds a global variable `retstack` which is an Standard Template Library (C++) (STL)-stack that holds instruction addresses `ADDRINT`. In the first callback routine the address of the call instruction will simply be pushed on the top of the stack.

```
VOID push_ret_address(VOID *r_ip)
{
        ADDRINT *address = (ADDRINT*)r_ip;
        retstack.push(*address);
}
```

In the second callback routine the top address on the stack is compared to the target address of the return instruction. If the addresses do not match, the next will be popped until there is a match or until the stack is empty. If the stack gets empty an ROP-attack is found.

```
VOID check_ret_address(VOID *r_ip)
{
        ADDRINT *addrp = (ADDRINT*)r_ip;
        BOOL equal;
        do {
                ADDRINT addr = retstack.top();
                retstack.pop();
                equal = *addrp == addr;
        } while (!equal && !retstack.empty());
        if (!equal){
                ROPDetected = 1;
                cout << "ROP DETECTED!!!" << endl;
        }
}
```

The reason that the stack is popped until it is empty is because of potential longjumps, which disobey the normal call-return pairs.

4.3 Implementing Zero Sum Defender

Zero Sum Defender is implemented as an LLVM transformation pass. LLVM is a collection of compilers for different languages such as C and C++. LLVM provides a framework for writing transformation passes for the intermediate code used in the compiler. A transformation pass suits very well for the implementation of this technique since the modifications done are not architecture specific. The modifications can be done on the intermediate code, which in turn can be compiled into assembly code for different architectures. The framework is written in C++.

4.3.1 LLVM Intermediate language

A program in the LLVM intermediate language is represented as a `Module`, which consists of one or many `Functions`. A `Function` consists of one or many `Basic Block`s, which consists of one or many `Instruction`s. There are many different kinds of `Instruction`s which represent assembly instructions, e.g branch instructions, binary instructions and call instructions. A `Basic Block` must end in a terminating instruction, and may only contain one terminating instruction. A terminating instruction is e.g. a return instruction.

There are several different types of transformation passes in LLVM, which operate on different parts of the code. The different transformation passes operates on different parts of the program. Zero Sum Defender is written as a `FunctionPass` since it operates on functions. The `FunctionPass` offers the possibility to modify every function in the program. Everything that is contained in the function can be modified, but not anything around the function. At first, an initializing method is run to set up the environment, with a reference to the program's module. A functionPass method is then run on every function in the program, with a reference to that function.

4.3.2 Variables

Zero-sum Defender uses a global variable that is used as a counter. The global variable is incremented at the start of each function and decremented before the return instruction. After the decrementation the counter is compared with a constant zero. If the counter is less than zero, an ROP-attack is detected and an exit-function is called.

To initialize the global variable, the doInitialization method is implemented.

```
virtual bool doInitialization(Module &M)
```

The initialization method creates a random variable name, which is used to reference the global variable throughout the pass. It generates a new name for the variable until a unique name is found. When the `GlobalVariable` is created it will be inserted into the supplied `Module`, and will be referencable via the `Module` later in the pass using the random string, `counterRef`, that was generated.

```
IntegerType *intType = IntegerType::get(M.getContext(), 32);
ConstantInt * zero = ConstantInt::get(intType, 0);
new GlobalVariable(M, intType, false, GlobalVariable::ExternalLinkage, zero,
 counterRef, NULL, GlobalVariable::LocalExecTLSModel);
```

The variable is created with an integer type of 32 bits, and initialized with the constant zero. The variable has `LocalThreadMode`, which means that the variable will be unique for every thread in the program.

A global string variable will also be created in this function, that is initialized as a message that will be displayed when an ROP-attack is detected. This global string will be printed in the exit-block.

4.3.3 Putting it Together

The `runOnFunction` method will be implemented to do the modifications on every function in the program.

```
virtual bool runOnFunction(Function &F)
```

The global counter is fetched from the `Module` that is referenced using the `getParent()` method. In the intermediate language a load instruction is created to load the global variable. An add instruction is then created, with the load

instruction as one parameter and a constant, one, as the other parameter. A store instruction is then created to store the value at the global variable.

These three instructions are created after the first instruction in the function. All instructions are created with the class IRBuilder that is initialized with a pointer to the instruction the newly created instructions will supersede.

```
ConstantInt *one = builder.getInt32(1); // Create constant = 1
Value *load = builder.CreateLoad(counter); // Load counter
Value* add = builder.CreateAdd(load, one); // counter = counter + constant (1)
builder.CreateStore(add, counter); // Store counter
```

At the end of the function the same instruction sequence is created, but with a subtracting instruction instead of an adding instruction. A compare instruction is then created that compares the global counter with the constant zero.

```
load = builder2.CreateLoad(counter); // Load counter
Value* sub = builder2.CreateSub(load, one); // counter = counter - constant (1)
builder2.CreateStore(sub, counter); // Store counter
ConstantInt * zero = builder2.getInt32(0); // Create constant = 0
Value * cmp = builder2.CreateICmpSLT(sub, zero); // Signed Less Than Compare
// counter with constant (0)
```

A conditional branch is then created that will jump to either an exit-block or a return-block. The branch will jump to the exit-block if the counter is less than zero. The return-block will be jumped to otherwise.

```
Instruction *branch = builder2.CreateCondBr(cmp, exitBlock, returnBlock);
// Conditional branch. If true -> exitBlock else -> returnBlock
```

Both the exit-block and the return-block are newly created basic blocks, which are inserted at the end of the function. The exit-block gets two instructions inserted; a call instruction that will call the system call exit, and an unreachable instruction.

```
BasicBlock *exitBlock = BasicBlock::Create(context, "Exit-block");
ConstantInt* exitArg = exitBuilder.getInt32(1);
exitBuilder.CreateCall(printf, errorMsg);
exitBuilder.CreateCall(exitFunction, exitArg);
exitBuilder.CreateUnreachable();
```

The unreachable instruction is a terminating instruction that is known to never be reached. Since every basic block needs to end with a terminating instruction, this will be used. The return-block will only contain a copy of the function's original return instruction.

```
BasicBlock *returnBlock = BasicBlock::Create(context, "Ret-block");
Instruction * retInst = lastInstr->clone();
returnBlock->getInstList().push_back(retInst);
```

4.4 Experimental Results

4.4.1 Performance Results

The ROP protections implemented in this chapter were tested in the performance benchmark suite `UnixBench`. The performance results of ROPdefender is presented in table 4.1 and ZeroSum in table 4.2. The scores presented in the first and second column represent how many iterations the test program ran in a given period of time. The iterations are then weighted with an index, but are still directly comparable for calculating the performance overhead. The performance overhead is given in percentage.

4.4.2 Binary Results

For ROPdefender the binary overheads for the protected applications are zero, since the binaries are not modified from the original. In order to run `PIN`, though, 248,760KB is needed.

For ZeroSum, the average binary overhead compared to the original `gcc` compiled binaries is negative, -1.44%. The binaries were also compiled with `clang` without the ZeroSum transformation pass. The average binary overhead compared to these binaries was also negative, -1.92%. The binary size results can be seen in table 4.3.

4.4.3 Analysis

ROPdefender

For the ROPdefender analysis, the different test programs were run under the instrumentation of `PIN`, together with the implemented ROPdefender `Pintool`. The results shows us that it is very slow to run an application under the instrumentation of `PIN`. Compared to original execution, the applications take five times as long to run in average. This is much larger than the performance overhead given by the authors, Davi et.al., which was only up to two times in average. This shows that the impact of `PIN` varies for different systems.

The performance overhead is ROPdefender's biggest weakness, and is what makes it impossible to use for most applications today.

ZeroSum

For the ZeroSum analysis, the different programs had to be compiled with `clang`, using the implemented ZeroSum transformation pass. The tests used for the ZeroSum evaluation differ from the first, because of compability issues. Some tests were not possible to transform in LLVM bitcode format, because of their inconsistent CFG. This shows a great weakness of ZeroSum, which has to rely on the LLVM optimizer to be able to traverse the CFG of the protected program.

The binary overhead for the ZeroSum binaries is very interesting. According to the authors, Jeehong et.al., the binary overhead should be around 4.5%. The

Test	Original	ROPdefender	Performance overhead
Whetstone-double	4,627.9	405.468	1,041.37%
dhry2reg	435,253,593	82,042,026	430.52%
hanoi	36,370,733	6,967,711	421.99%
spawn	255,222	2,070	12,229.57% [1]
pipe	20,754,844	1,383,758	14.00%
double	81,683,469	63,684,875	28.26%
int	116,194,150	107,515,273	8.07%
long	35,158,200	34,775,398	1.10%
short	110,993,421	108,165,493	2.61%
average			416.73%

Table 4.1: Performance evaluation of the protection ROPdefender. The performance overhead is given in percentage.

negative overhead compared to the `gcc`-compiled binaries can be motivated by that `clang` is a more efficient compiler than `gcc`.

The negative overhead compared to the `clang`-compiled binaries is harder to explain. In order to run the ZeroSum transformation pass an LLVM application `opt` is run. `opt` is an optimizer and analyzer, which can run different transformation and analysis passes depending on the input parameters. The invokation of `opt`, together with the input parameters for running the ZeroSum transformation pass, is the only difference between the two binary versions. This means that `opt` performs optimizations that affect the binary sizes, even if it is not given as an input parameter.

[1] The high output is due to extra calls to `printf`. This is not included in the average.

Test	Original	ZeroSum	Performance overhead
Whetstone-double	4,627.9	4,449	4.02%
dhry2reg	435,253,593	393,011,264	10.75%
execl	2,804	2,844	-1.41%
copy1024/2000	1,182,547.5	1,101,264	7.38%
copy256/500	325,073.9	299,290	8.62%
copy4096/8000	2,690,116.6	2,601,921	3.39%
pipe throughput	2271896.3	2,202,871	3.13%
pipe contextswitch	128,077.9	136,941	-6.47%
process creation	8,393.4	9,040	-7.15%
average			2.47%

Table 4.2: Performance evaluation of the protection ZeroSum. The performance overhead is given in percentage.

Binary	`gcc`	`clang`	`clang` with ZeroSum
arithoh	8,882	8,792	8,315
context1	10,856	10,766	10,766
dhry2	11,552	11,536	11,536
dhry2reg	11,560	1,1536	11,536
double	9,010	8,888	8,474
execl	18,754	20,685	25,154
float	9,010	8,856	8,473
fstime	14,514	14,451	16,708
gfx-x11	26,411	26,411	26,411
int	8,946	8,792	8,311
long	8,946	8,792	8,312
looper	10,750	10,676	4,656
register	8,946	8,792	8,316
short	8,946	8,792	8,313
syscall	10,890	10,816	10,205
whetstone-double	15,153	16,085	20,966
average binary overhead		-1.44%	-1.92%

Table 4.3: Binary size evaluation of the protection ZeroSum. The sizes are given in KB and the binary size overhead is given in percentage.

Chapter 5

ORPScan: Combining Techniques for Improved Performance

5.1 Introduction

ROP almost always originate in vulnerabilities in the handling of input data of a program. The payload of an ROP-attack is often inserted into a program's memory via different inputs such as network streams or file streams. The idea of ORPscan is to detect ROP attacks before they enter the targeted program. To do this, ORPScan scans input data and differentiates ROP payload data from normal data, for the target application.

Since an application or library binary contains a large amount of gadgets, ORPScan uses the ORP (section 3.10) technique in order to make it more memory efficient. ORP is a technique that removes up to 80% of a binary's gadgets, which makes it good in combination with ORPScan.

In this chapter the ORPScan technique will be thoroughly described. The ROP payload structure will be described, followed by a description of the techniques ORPScan uses to first remove gadgets, and then detect them.

5.2 Background

There are many different data scanners present that detect code-injection attacks. Code-injection can be done through many different data inputs, network traffic, process buffers or memory dumps. Scanners can have a static or dynamic code analyze method. Static code analyzing scanners are scanners that look for attributes in remotely injected shellcode, e.g. [32] that looks for NOP-sleds or sledges and [18,34] that looks for other attributes in the shellcode. Dynamic code analyzing scanners, e.g. [24], uses emulation and is able to detect obfuscated shellcode that is self-modifying or uses polymorphism.

ROPScan [25], described in section 3.7 on page 33, is a dynamic code analyzing scanner which emulates possible addresses in order to find ROP-gadgets. ROPScan which uses a CPU emulator to emulate possible shellcode attacks. It creates a copy of the process' memory space with registers and memory segments. Every memory address found in the input data, that is within the process' executable memory space, is treated as if it was going to be executed as a potential ROP-gadget. If the

execution ends in a return-instruction within a certain number of instructions, the address is a potential gadget-address. ROPScan uses a sliding window technique to read the input data. The window slides 8 bits over the input data, and interprets the following 32 bits as an address.

ROPScan is a very effective technique of scanning input data, with speeds of 120Mbit/s in average. The part of ROPScan that is the most CPU-intensive, and thus slows down the scanning process the most, is the emulated execution of the code. This is the part that ORPScan tries to remove.

ORP is an In Place Randomization technique described in section 3.10 on page 40. In Place Randomization refers to making changes in a binary which does not remove the functionality of the program, but removes the functionality of the ROP-gadgets. The changes can be done in many different ways, and are chosen randomly, in order to create a unique copy of the original binary file.

ORP offers many different In Place Randomization techniques, such as atomic instruction substitution, instruction reordering and register reassignment. According to [22], ORP is able to remove the functionality of up to 80% of the gadgets in a binary. It has also very low performance overheads, since the changes made to the binary are not causing any significant changes on the application's execution.

ORP can be used as an ROP-protection by itself, either by randomizing binaries before distribution, or before running.

5.3 Motivation

One of the goals of this thesis work is to design a new protection technique, e.g. by combining parts of different techniques to create a better solution. ORPScan has been designed by combining the technique In Place Randomization with an Input Scanner. The In Place Randomization part has been used as it is, with some modifications.

The Input Scanner is inspired by ROPScan, but with a more efficient gadget lookup method. The idea is that by rewriting and removing gadgets from the target program, the gadget lookup will be more efficient.

A more detailed description of the design and security is given in the end of this chapter.

5.4 ORP

The source code of ORP has been used in the implementation of orpscan. In this section ORP and the relevant parts of the source code that have been used will be described.

ORP is an application that implements the ideas of In Place Randomization. It is written in `Python` and depends on the `Python` libraries `Pygraph`, `PEfile`, the external library `Libdasm` and the binary disassembler `IDA Pro`. ORP is written for 32-bit Windows and requires the `.NET Framework`.
Here is a list of the different components of ORP.

Main file

`orp.py` - Main executable which can be called to randomize a library and generate coverage and exploit evaluation.

Class files

`insn.py` - Contains the classes `Operand` and `Instruction`.

`gadget.py` - Contains the classes `SimpleGadget` and `Gadget`.

`bbl.py` - Contains the class `BasicBlock`.

`func.py` - Contains the class `Function`.

Algorithm files

`equiv.py` - Responsible for switching equivalent instructions with each other.

`preserv.py` - Responsible for reordering the pushing and popping of callee-saved registers.

`swap.py` - Responsible for doing live analysis on registers and to swap registers.

`reorder.py` - Responsible for computing the dependency graph for the basic blocks and to reorder instructions within them.

Test files

`test.py` - Contains tests for the different classes.

Other files

`inp.py` - Responsible for computing relocation information of the randomized version and to write randomized changes to the copy of the binary.

`inp_dump.py` - Responsible for dumping debug information.

`inp_ida.py` - Interacts with `IDAPro`.

5.4.1 Main execution

`orp.py` can be run directly and takes different options for doing randomization, coverage- and payload evaluations and dumping the CFG of the binary to a file. Along with the option it takes a filename to the binary that is being randomized or analyzed. `orp.py` in its turn invokes the appropriate method from the different components depending on which option is passed.

5.4.2 Randomization

Orp gets all the functions in the binary from `inp.py` and classifies, analyzes and iterates over them. The classification is done to resolve the call relationships and will give every function a level value depending on what call-level it is on. The function analysis does analysis that will be used in the randomizations. It analyzes the function's register-arguments, register return values, preserved registers and the pairs of pops and pushes that preserves them.

The functions are then iterated to be randomized them one by one. The different randomization algorithms in the files `swap.py`, `preserv.py`, `equiv.py`

and `reorder.py` are called to analyze and create new variations of the functions in the way described earlier. All variations are stored in a vector for each function, from which one is randomly chosen to be used and put in a global diff-vector. To patch the original binary, `inp.py` is called with the global diff-vector as parameter. `inp.py` creates a new binary which is a copy of the original one being processed and writes the random changes to it.

5.4.3 Coverage evaluation

The coverage evaluation does the same as randomization, but instead of creating a new binary it gathers information about the possible gadgets in the binary and which gadgets that have been eliminated or broken in any way. `orp.py` calls the module `eval.py` to do the evaluation. `eval.py` gets, analyzes, classifies and randomizes the functions as before. Every type of change is stored in a different vector, and every change is saved for every function. It then calls `gadget.py` in order to find all possible gadgets in the binary.

All gadgets are then iterated over to count which were possible to change. The ratios of changed over total gadgets are printed to the screen. Note that these statistics show every possible change that can be done to every function. When the randomization is done, only one change is done per function.

5.4.4 ORP usage in ORPScan

Since ORP eliminates a large part of the gadgets of a binary with a very low cost, it suits well to be used in combination with other methods that have a complexity based on the number of gadgets. ORP is used in orpscan by using in-place-randomization of a binary and then recording which gadgets are still left that needs to be protected against. ORP has functionality to do in-place-randomization and exact analysis of gadget elimination, which makes it easy to use for these purposes.

Different solutions have to be considered depending on how ORP is used. ORP can mainly be used in two ways. Either the copies of an application or library are pre-randomized per distribution, or they are randomized at loading time. If they are pre-randomized, a list of vulnerable gadgets could be distributed together with the binary. The list will be a complete list of gadgets that the binary always is vulnerable to.

If the binary is randomized at load time, to get an exact list of remaining gadgets the list has to be generated at load time. A second alternative is to distribute a list with gadgets that are guaranteed not to be randomized during load time. This means that at different run times there will be different gadgets that are still left to use, but not on the list. It should be impossible for an adversary to know which these gadgets are. Entropy could also be taken into account by recording gadgets under a certain entropy level.

5.5 ROP payload

An ROP payload consists of a mixture of gadget addresses, data that is supposed to be loaded in the attack and dummy data. Dummy data can have different purposes, e.g. as data that is popped from the stack as a side effect of a gadget.

A ROP payload may also include shellcode that is supposed to be run in the attack. Gadget addresses are addresses to ROP-gadgets that belongs to the target application or a library that is loaded by it. The addresses must be present in the running application's address space.

In order to detect an ROP-payload among other data these three things can be looked for; gadget addresses, attack data and dummy data. Attack data and dummy data will continuously be called payload data. A gadget address will be directly followed by another gadget address or payload data. It is hard to identify payload data, since they can contain any value and be of different lengths. Most often the length of the data is multiples of the length of a memory address. The length of the data is most often also limited in size. By performing statistical calculations on known ROP-payloads a limit can be determined for the size of the payload data.

By combining what is known about gadget addresses and payload data, a couple of rules can be defined for classifying ROP-payloads.

- The data must contain min_g of gadget addresses.
- The gadget addresses must be separated by at most max_d bytes of data.

min_g and max_d are integers which can be determined by doing statistical analysis of existing ROP-attack payloads and comparing with non-payload data. An example of this will be described later in this chapter, with results that show that these values can be determined with high accuracy.

5.6 ORPScan Design

ORPScan is divided into two major phases; the randomization phase and the scanning phase. The randomization phase is done offline, and can be done either before distributing the application or before every run of the application.

The scanning phase takes place before every time data is being read by or imported in the application. An application that opens files during startup will use the scanner only at startup, while e.g. a web-server will use the scanner frequently for every request that arrives.

5.6.1 Randomization phase

The randomization phase is built on top of ORP. Several different binary rewriting techniques are used in order to shuffle the instructions and registers used, without changing the result of the execution. The design of ORP and it's rewriting techniques are described in section 3.10 and in [22]. The randomization phase creates a copy of the original binary file, which can directly replace the original. The new binary file has up to 80% of the gadgets destroyed, which means that an

ROP-attack with a payload written for the original binary will not work for the rewritten binary.

The randomization phase starts by statically analyzing the binary to extract its functions and CFG. This is done with the statically analyzing disassembler `IDA Pro`. Using the disassembled information about the binary, the binary rewriting is performed. During the rewriting of the binary, the possible ROP-gadgets of the binary are also determined. In particular, the ROP-gadgets that are not affected by the rewriting are noted. The addresses of these gadgets are written to a file, later to be used by the scanner.

5.6.2 Scanning phase

The design of ORPScan's scanner originates in the design of ROPScan. ORPScan uses the same scanning technique as ROPScan, i.e. a sliding window. Every 32 bits the window slides over is interpreted as an address.

When ORPScan is initialized it reads the gadget-file produced during the randomization phase. The gadget-file contains all the addresses to the ROP-gadgets that are still present in the binary. These gadgets-addresses are saved in a map in memory. Each time the scanner slides over a 32-bit value in the input data, it performs a lookup in the gadget map to check whether it is a gadget address present in this binary.

While scanning, ORPScan keeps count of the number of gadgets found, with the maximum distance max_d between them. If ORPScan finds min_g gadgets with this property, an ROP attack has been found, and the target application will be notified.

5.7 Experimental Evaluation

In order to evaluate ORPScan the scanning has been tested on many different data. ORPScan is used as an input scanner for different data inputs, such as `.pdf-files` or network streams. ORPScan has been tested with arbitrary data of these kinds, in order to evaluate ORPScans efficiency and ability to detect ROP-payloads.

Two different attributes have been measured during the evaluations; number of gadget addresses found and the distance between the addresses in bytes. The goal of the experiments are to determine the optimal values for those parameters when tuning the scanner. The experiment results will show if it possible to choose parameters that will distinguish ROP-payloads from arbitrary data, and if so, what the parameters optimal values are. A description of the different binaries, with their corresponding ROP-payload, used in the experiments is given in table 5.2.

The scanner was first evaluated for arbitrary data. This arbitrary data consisted of `.pdf`-files and random generated data, as can be seen in the leftmost column of table 5.3. The different data were evaluated on a few different libraries, as can be seen in the top row of the same table. For the experiments with arbitrary data, the total number of gadgets and the minimum and average distance between the gadgets were recorded.

The scanner was then evaluated against real ROP-payloads. At first a payload was created that used gadgets from all the possible gadgets in the binary. Most of

File	Description
MSRMFilter03.dll	Is shipped with the application Mini-stream RM-MP3 Converter^{TM} [1]. ROP-attack from [14].
msvcr71.dll	Is shipped with Java JRE 1.6 for Windows. ROP-attacks from [33].
hxds.dll	Is shipped with Microsoft Office 2010 Windows. ROP-attack from [33].

Table 5.1: Description of the different binaries used in the experimental evaluation of ORPScan

File	Size
IntelManual.pdf	2,473KB
Geology.pdf	3,513KB
Bible.pdf	9,158KB
IKEA.pdf	19,044KB
rand100MB	100MB
rand1GB	1GB

Table 5.2: Sizes of the different input data used in the experiments.

these gadgets will be removed during the randomization phase of ORPScan, and the payload will not work the same way. The payload was then translated into using only the new, smaller gadget space. The payloads were only rewritten to use the new gadget-addresses, but to keep the payload structure the same. This was done to imitate a ROP-payload written to attack an already randomized binary.

The new payload was then evaluated against the different libraries. For these evaluations the total number of gadgets found and the maximum and average distance between them were recorded.

The performance experiments are done on a Windows 8.1 Pro with Intel Core i5 3.40Ghz, with 8Gb RAM.

5.8 Security Evaluation

ORPScan is using a combined security of ORP during the randomization phase and an input scanner during the scanning phase. The combination of the two systems does not only create a more effective system, but a system with multiple levels of security.

As mentioned before, the randomization phase can be performed at different times in the lifetime of a binary file. The distributor may randomize the binary before every distribution. The user of a binary may randomize the binary once, when receiving the binary, or frequently, e.g. each time it is started or once a week. Each time the binary is randomized, all earlier crafted ROP-payloads will

File	MSRMFilter03.dll	msvcr71.dll	hxds.dll
IntelManual.pdf	6	5	16
Geology.pdf	37	8	19
Bible.pdf	25	11	73
IKEA.pdf	44	17	110
rand100MB	275	175	850
rand1GB	1,792	1,536	5,120

Table 5.3: The number of gadgets from the binaries in the top, found in the input files to the left.

File	MSRMFilter03.dll	msvcr71.dll	hxds.dll
IntelManual.pdf	143,718	33,830	23,191
Geology.pdf	752	28,188	37,770
Bible.pdf	1,457	216,226	569
IKEA.pdf	3,409	7,415	38
rand100MB	29,172	73,325	4,224
rand1GB	157,555	174,985	9,344

Table 5.4: The minimum distance between the gadgets from the binaries in the top, found in the input files to the left.

File	MSRMFilter03.dll	msvcr71.dll	hxds.dll
IntelManual.pdf	290,898	300,029	120,744
Geology.pdf	91,159	237,985	162,901
Bible.pdf	338,247	718,643	117,291
IKEA.pdf	391,610	890,958	168,424
rand100MB	340,023	581,339	120,056
rand1GB	371,873	681,426	204,804

Table 5.5: The average distance between the gadgets from the binaries in the top, found in the input files to the left.

Attack	# of gadgets	max. distance betw.	Avg. distance betw.
MSRMFilter03.dll	22	28	3.8888
msvcr71.dll (1)	16	7	5.1875
msvcr71.dll (2)	13	8	5.1539
hxds.dll	18	8	4.3889

Table 5.6: The number of gadgets found and the maximum and average distance between them, for the different ROP-payloads.

be obsolete. It is often only necessary to break one gadget in a payload to remove the functionality of the whole payload.

The next level of protection is the scanning phase. This phase will detect every payload that is customly crafted for the running randomized instance of the binary. Since the scanning phase only looks for payloads that are targeted specifically against the randomized binary, the scanning will add an extra layer of protection. This will protect against any attacks where the randomized version has been compromised.

5.9 Results and Discussion

When the scanner was evaluated with arbitrary data, the total number of gadgets found and the minimum and average distance between them were recorded. The number of gadgets can be seen in table 5.3, the minimum distance can be seen in table 5.4 and the average distance can be seen in table 5.5.

For the ROP-payload experiments the total number of gadgets found and the maximum and average distance between them were recorded. The results from these experiments can be seen in table 5.6.

5.9.1 Scanning Speed

During the scanning part, the scanning speed was measured. The result was in average 80.1 Mbit/s.

The resulting scanning speed is close to the scanning speed of ROPScan, which has a scanning speed of 120Mbit/s in average. This result is promising but not satisfying, since the motivation of ORPScan was to find a scanner with a better scanning speed. The difference is hard to explain, since ORPScan should be faster in theory.

The scanning part can be divided into two parts, iteration and ROP-detection.

The iteration of the input data is done in the same way in both techniques. A sliding window is used, which slides one byte at a time to find 4 byte addresses. The iteration speed of ORPScan was also measured, in order to see how much time was spent on actually iterating the input data.

The ROP-detection part is differing in the two techniques. In ROPScan every address is compared to an interval. If the address is contained in the interval, it is emulated in an emulation environment. The first part is constant, and the second part is an extra constant overhead. In ORPScan every address is looked up in a hash table, to find a match. This is always constant.

Based on this analysis, the iteration part should be the same speed for both techniques. The ROP-detecton part should be equal or faster for ORPScan. The iteration speed for ORPScan is 94.90Mbit/s in average, which is 25Mbit/s less than the total scanning speed of ROPScan. This suggests that ROPScan is implemented in a more efficient way and in a more efficient environment.

Hopefully ORPScan can be implemented in a more efficient manner in the future.

5.9.2 Number of Gadgets

The results from the experiments show that the number of gadgets in arbitrary data varies by a large amount. A quick overview of the figures indicates that the number of gadgets found in an input data is not enough to determine if it contains an ROP-payload. For instance, the second ROP-attack on `msvcr71.dll` contains only 13 gadgets, while the IKEA catalogue contains 3409 gadgets.

5.9.3 Distance between Gadgets

The distance between the gadgets gives a better indicator of a ROP payload. The maximum distance between gadgets in the payloads can be compared with the minimum distance between gadgets in the arbitrary data.

The largest distance among the payloads is 28 bytes (`MSRMFilter03.dll`) and the smallest distance among the arbitrary data is 38 bytes (IKEA). This means that a definite limit can be chosen, to separate all the payloads from the arbitrary data.

E.g. if we had chosen a limit of 32 bytes, we could say that all data with maximum distance below 32 bytes are payloads, and all data with minimum distance over 32 bytes are not payloads.

5.9.4 Average distance between Gadgets

The minimum distances between gadgets varies a great deal. For many input data the minimum distance is from one KB, up to a hundred KB. This suggests that the average distance between the gadgets may be an even better indication of a ROP payload. For all the arbitrary input data, the average distance between gadgets is between 100 KB and 1 MB.

When only relying on the average distance between gadgets there may be a problem if a payload is hidden in data much larger than the payload. For a large size arbitrary data there resides many gadgets, which means that average distance for the surrounding data will not be affected much by the smaller payload. E.g. if the `hxds-attack` is hidden inside the `Geology.pdf`, the total average distance will be 83,654.00.

A different strategy can be used to find payloads hidden in arbitrary data. Instead of calculating the total average distance between all gadgets in the input data, the average distance between the n last encountered gadgets can be calculated. The ROP attacks tested in this analysis contains in average 17 gadgets. An n smaller than that works well for the payload examples in this analysis.

Looking at the same example as before, the `hxds-attack` hidden inside the `Geology.pdf`, with an n of 10. The average distance for the last ten gadgets would be 162,901 in average before encountering the payload. When the payload is encountered, and the first ten gadgets are scanned, the average distance would drop to around 4.

5.10 Future Implementations

There are a number of features that will need to be implemented in order to make ORPScan efficient for use.

5.10.1 Multiple libraries

The prototype version of ORPScan works only on one library per application. In real scenarios there are often many libraries in the application's memory space that are vulnerable to ROP-attacks. A future version of ORPScan could have the possibility to find all non-ASLR libraries for an application and randomize them. When all the libraries are randomized, all the gadgets can be combined into one file. When scanning for ROP-payloads, all the gadgets in the file can be taken into account.

5.10.2 ROP-NOP

A ROP-NOP is the ROP-version of a no-operation instruction. It is simply a gadget consisting of only a return instruction. The only thing this gadget does is to decrease the stack pointer, to point to the next gadget. A sequence of ROP-NOP's in a payload can be used as a landing pad, if it is hard to predict exactly where execution will start in the beginning of the attack. The execution can start anywhere on the landing pad, and it will always lead to the first gadget after the pad. ROP-NOP's can also be used as dummy data to be loaded as a side effect of certain gadgets.

ROP-NOP's can make it harder for the scanner to detect the payload. There is a solution to this problem. During the randomization phase, the address of every return instruction in the binary can be recorded and saved in the same list as the regular ROP-gadgets.

This was tested on the payload for `MSRMFilter03.dll` in table 5.6. This attack payload contained six ROP-NOP's in the middle of the payload. After saving the address of this return-instruction, the maximum distance between gadgets became 8 instead of 28, and the number of gadgets became 28 instead of 22. For a payload with a large amount of ROP-NOP's, this technique could be necessary in order to be able to detect it.

5.10.3 Gadget detection

The gadget detection during the randomization phase needs to be fine-tuned. At the moment the gadget detection has a hard limit on the number of instructions that can be part of a gadget. The gadget detection also needs to be able to find gadgets which are separated in parts by branches other than return.

There is much work done in this field, e.g. Q [27]. Q is a ROP-compiler which can create payloads for given binaries. Q can produce working payloads for 80% of Linux `/usr/bin` programs larger than 20KB. The advanced techniques of finding ROP-gadgets used in Q could also be adopted in ORPScan, to be used in the randomization phase.

Chapter 6
Conclusions

The goals of this thesis work were to study and analyze different known protections against Return-Oriented Programming (ROP), to implement and test different known protections, to explore the possibility of designing a new protection by combining parts of different protections and finally to implement this new design.

In the first part the background of ROP was presented. The buffer overflow exploit lead to code-injection attacks by injecting code on the stack. The execution of the stack was prevented, which lead to code-reuse attacks. First they were in the form of Return-into `libc` (RILC), and in 2007 ROP was introduced.

Seventeen different ROP protections are presented in the second part. A comparison of performance and binary overhead is given in the end of this part. It can be concluded from this comparison that all the protections have different strengths and weaknesses. Different protections have different application areas. If performance is important, a faster but less covering protection such as ROPGuard can be chosen. If performance is not an issue, and a robust and reliable security solution is required a protection based on Control Flow Integrity (CFI) can be chosen.

In the third part, the implementations of ROPDefender and Zero-sum Defender were presented, together with experimental tests. This part has given more insight in how the implementations are done, and have also given the opportunity to examine the performance results, which were .

In the last part of this work the protection ORPScan was presented. ORPScan is a result of combining parts of different techniques to create a more efficient protection technique. Unfortunately the resulting speed was not satisfying, which may have to do with implementation issues. The detection rate was very satisfying with no false negatives and no false positives.

Acronyms

AIR	Average Indirect target Recordeduction. 15
API	Application Programming Interface. 28, 44–46
ASLR	Address Space Layout Randomization. 14, 15, 17, 37, 51, 83
CFG	Control Flow Graph. 19, 20, 30, 41, 53, 70, 75, 78
CFI	Control Flow Integrity. 15–17, 19, 23, 30, 31, 51–55, 65, 85
CISC	Complex Instruction Set Computing. 6
COTS	Commercial Off The Shelf. 35
CPU	Central Processing Unit. 13, 33, 34, 44, 45
DEP	Data Execution Prevention. 14, 15, 26
ELF	Executable and Linkable Format (Linux). 31, 54, 56
GTT	Global Translation Table. 55
IBT	Indirect Branch Target. 38
ICF	Indirect Control Flow. 16, 19, 53–55
IDT	Interrupt Descriptor Table. 58
ILR	Instruction Location Randomization. 39
IP	Instruction Pointer. 38
IPR	In Place Randomization. 41
JIT	Just In Time. 28, 37, 65
JOP	Jump-Oriented Programming. 3, 12, 13, 27, 37, 44, 45, 60
KLOC	Kilo Lines of Code. 39
LBR	Last Branch Record. 44–46, 57, 58
LOC	Lines of Code. 28

MTT Module Translation Table. 54

NOP No Operation Instruction. 10, 23–25, 27, 36
NX Never eXecute. 58

PC Program Counter. 55
PE Portable Executable. 42
PIC Position Independent Code. 14, 36, 55
PVM Per Process Virtual Machine. 38, 39

RILC Return-into `libc`. vii, 3–6, 23, 27, 52, 85
ROP Return-Oriented Programming. vii, ix, 1–3, 5–13, 15–17, 19–23, 26–28, 33, 34, 37, 44, 45, 47, 52, 57, 58, 60–62, 65, 67, 68, 70, 73, 74, 77–83, 85

SP Stack Pointer. 36
STL Standard Template Library (C++). 66

TCB Trusted Computing Base. 28, 29
TIB Thread Information Block. 47

VM Virtual Machine. 16, 28

XD eXecute Disable. 14
XN eXecute Never. 14

Bibliography

[1] http://www.mini-stream.net/rm-to-mp3-converter.

[2] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Trans. Inf. Syst. Secur.*, 2009.

[3] D. Blazakis. Interpreter exploitation. In *Proceedings of the 4th USENIX Conference on Offensive Technologies*, WOOT'10. USENIX Association, 2010.

[4] T. Bletsch, X. Jiang, and V. Freeh. Mitigating code-reuse attacks with control-flow locking. In *Proceedings of the 27th Annual Computer Security Applications Conference*, ACSAC '11. ACM, 2011.

[5] Bulba and Kil3r. Bypassing stackguard and stackshield. *PHRACK MAGAZINE*, Volume 0xa Issue 0x38, 2000.

[6] P. Chen, H. Xiao, X. Shen, X. Yin, B. Mao, and L. Xie. Drop: Detecting return-oriented programming malicious code. In *Information Systems Security*, volume 5905 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2009.

[7] Y. Cheng, Z. Zhou, Y. Miao, X. Ding, and H. R. DENG. Ropecker: A generic and practical approach for defending against rop attack. Research Collection School Of Information Systems, 2014.

[8] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7*, SSYM'98. USENIX Association, 1998.

[9] L. Davi, A.-R. 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*, ASIACCS '11, 2011.

[10] T. Durden. Bypassing pax aslr protection. *PHRACK MAGAZINE*, Volume 0x0b, Issue 0x3b, 2002.

[11] H. Etoh and K. Yoda. Propolice: Improved stacksmashing attack detect on. In IPSJ SIGNotes Computer SECurity 014, 2001.

[12] M. Frantzen and M. Shuey. Stackghost: Hardware facilitated stack protection. In *Proceedings of the 10th Conference on USENIX Security Symposium - Volume 10*, SSYM'01. USENIX Association, 2001.

[13] I. Fratric. Runtime prevention of return-oriented programming attacks, 2012.

[14] FuzzySecurity. Part 7: Return oriented programming. http://www.fuzzysecurity.com/tutorials/expDev/7.html, may 2014.

[15] A. Gupta, S. Kerr, M. S. Kirkpatrick, and E. Bertino. Marlin: Making it harder to fish for gadgets. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12. ACM, 2012.

[16] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. Davidson. Ilr: Where'd my gadgets go? In *Security and Privacy (SP), 2012 IEEE Symposium on*, 2012.

[17] K. Jeehong, K. Inhyeok, M. Changwoo, and E. Young Ik. Zero-sum defender: Fast and space-efficient defense against return-oriented programming attacks. In *IEICE TRANSACTIONS on Fundamentals of Electronics, Communications and Computer Sciences*, 2014.

[18] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Polymorphic worm detection using structural information of executables. In *Recent Advances in Intrusion Detection*, volume 3858 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2006.

[19] 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 '10. ACM, 2010.

[20] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, New York, NY, USA, 2007. ACM.

[21] 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 '10. ACM, 2010.

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

[23] 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)*. USENIX, 2013.

[24] M. Polychronakis, K. G. Anagnostakis, and E. P. Markatos. Comprehensive shellcode detection using runtime heuristics. In *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC '10. ACM, 2010.

[25] M. Polychronakis and A. D. Keromytis. Rop payload detection using speculative code execution. In *Proceedings of the 2011 6th International Conference on Malicious and Unwanted Software*, MALWARE '11. IEEE Computer Society, 2011.

[26] R. Roemer, E. Buchanan, H. Shacham, and S. Savage. Return-oriented programming: Systems, languages, and applications. *ACM Trans. Inf. Syst. Secur.*, 15(1), Mar. 2012.

[27] E. J. Schwartz, T. Avgerinos, and D. Brumley. Q: Exploit hardening made easy. In *Proceedings of the USENIX Security Symposium*, 2011.

[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 '07. ACM, 2007.

[29] H. Shacham, M. Page, B. Pfaff, E.-J. 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 '04. ACM, 2004.

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

[31] A. Srivastava, A. Edwards, and H. Vo. Vulcan: Binary transformation in a distributed environment. Technical report, 2001.

[32] T. Toth and C. Kruegel. Accurate buffer overflow detection via abstract pay load execution. In A. Wespi, G. Vigna, and L. Deri, editors, *Recent Advances in Intrusion Detection*, Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2002.

[33] P. Van Eeckhoutte. Corelan ropdb. https://www.corelan.be/index.php/security/corelan-ropdb/, may 2014.

[34] X. Wang, C.-C. Pan, P. Liu, and S. Zhu. Sigfree: A signature-free buffer overflow attack blocker. *Dependable and Secure Computing, IEEE Transactions on*, 2010.

[35] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, 2012.

[36] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou. Practical control flow integrity and randomization for binary executables. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13. IEEE Computer Society, 2013.

[37] M. Zhang and R. Sekar. Control flow integrity for cots binaries. In *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13. USENIX Association, 2013.

Appendix A

Appendix

File	Size	Description
IntelManual.pdf	2,473KB	Intel 64 and IA32 Architectures Software Developer's Manual Volume 1
Geology.pdf	3,513KB	Geology of Umnak and Bogoslof Islands Aleutian Islands Alaska, by F.M. Byers, JR. Investigations of Alaskan Volcanos, 1959
Bible.pdf	9,158KB	The Holy Bible, Kim James Version, 1611, Containing the Old and New Testaments
IKEA.pdf	19,044KB	IKEA Catalogue, 2014, USA
rand100MB	100MB	Random data.
rand1GB	1GB	Random data.

Table A.1: Sizes and descriptions of the different input data used in the experiments.

LUND
UNIVERSITY

http://www.eit.lth.se