unROP: Creating Correct Backtrace from Core Dumps with Stack Pivoting

by

Yuchen Ying

(Under the direction of Kang Li)

Abstract

Return-oriented Programming (ROP) has become the most common way to exploit bugs in application, and stack pivoting is a common techniques for facilitating the attack. Stack pivoting poses a challenge in finding the root cause of the exploitation because it is hard to trace the execution flow and identify the exact trigger point of exploitation. This thesis presents several ways to do stack pivoting and designed methods to traceback in different situations. We tested our methods with real system crash dumps and evaluate the effectiveness of our approaches. Our solution is expect to help malware researchers to debug and defend against ROP-based attacks.

Index Words: Return-Oriented Programming, ROP, Stack Pivot

unROP: Creating Correct Backtrace from Core Dumps with Stack Pivoting

by

Yuchen Ying

B.E., Beijing University of Posts and Telecommunications, 2010
M.A., University of Georgia, 2014

A Dissertation Submitted to the Graduate Faculty
of The University of Georgia in Partial Fulfillment
of the
Requirements for the Degree

Master of Science

Athens, Georgia

2014

© 2014
Yuchen Ying
All Rights Reserved

unROP: Creating Correct Backtrace from Core Dumps with Stack Pivoting

by

Yuchen Ying

Approved:

Major Professor: Kang Li

Committee: Kang Li
Roberto Perdisci
Krzysztof J. Kochut

Electronic Version Approved:

Maureen Grasso
Dean of the Graduate School
The University of Georgia
May 2014

unROP: Creating Correct Backtrace from Core Dumps with Stack Pivoting

Yuchen Ying

May 23, 2014

Acknowledgments

I would like to express my deepest appreciation to my major professor, Dr. Kang Li, for his continual and helpful guidance during the research and the writing of this thesis.

In addition, a thank you to Xiaoning Li and Ling Huang, from Intel Labs, for their insightful help in giving advice during the process. I also thank Lee Harrison for sharing his previous research and providing programming help.

Contents

Acknowledgements **v**

List of Figures **vii**

List of Tables **viii**

1 Introduction **1**

2 Background **3**

2.1 Direct Code Injection 3

2.2 Code Reuse Attack 4

2.3 Return-Oriented Programming 5

2.4 Stack Pivoting 7

2.5 Thread Environment Block 8

2.6 Core Dumps 8

3 Problem Definition **9**

3.1 Goal and Assumptions 9

3.2 Determining the Stack Pivoting Target 12

3.3 Types of Stack Pivoting 13

3.4 The Proposed Approach for Finding the Old `ESP` Value 15

3.5 Challenges in Finding the Old ESP Value 18
3.6 Plan for Recreating the Call Stack 21

4 System Overview and Implementation **23**
4.1 Debugging Symbol Preprocessor 25
4.2 The ESP Value Finder 27
4.3 The Stack Frame Analyzer 27

5 Implementation **28**
5.1 Debugging Symbol Preprocessor 28
5.2 ESP Finder 29
5.3 The Stack Frame Analyzer 29

6 Evaluation **30**
6.1 Manually Creating Core Dumps 30
6.2 Evaluation Results 31
6.3 Discussion of the Results 32

7 Discussion **33**
7.1 Correctness of the Pivot Target 33
7.2 Relying on the Debugging Symbol 33

8 Conclusion and Future Works **35**
8.1 Deterministic Algorithm to Search for the Old ESP Register Value 35
8.2 Other Stack Pivoting Scenarios 36

List of Figures

2.1 Return-Oriented Programming 6

3.1 How Stack Pivoting Works 13

4.1 System Design 24

4.2 A Calling Graph 26

List of Tables

3.1 Stack Pivoting Techniques and Solutions 16

6.1 Evaluation Results 32

Chapter 1

Introduction

Buffer overflows, among all memory vulnerabilities, have been exploited by attackers since the wide adoption of `C` and `C++` Programming Languages. The attacker will inject code into memory (usually called shellcode since it will grant a shell to the attacker) and execute it. To mitigate this kind of attack, techniques like W`XOR`X and Data Execution Prevention, were developed and widely adopted.

The focus of exploit then turned to Code Reuse Attack (CRA), of which, instead of injecting code, the attacker will direct control flow through existing code in memory and get a malicious result. Return-into-libc and Return-Oriented Programming are two examples of CRAs.

Due to the complexity of making a function call on the Windows system, most of the `ROP` attacks on the Windows system will construct a carefully crafted memory region, and then modify the `ESP` register to point to that address. In this way, the application will use that memory region when it needs to load arguments to function calls or when returning from previous function call and jump back to caller. Essentially, the new memory region is considered a stack. This whole procedure of modifying the `ESP` register to utilize another stack was called stack pivoting.

From the point of view of the security analyst, stack pivoting poses a challenge in finding the trigger point of the exploit. The attacker needs to utilize several small gadgets to construct an ROP chain, to prepare a special stack, and to do the redirection. We want to find the trigger point of exploitation when a crash occurs. We would like to trace it back as far as we can, find out if the stack pivot exists or not, and if it exist, determine if it is possible to find the original stack.

In this thesis, we first introduce the background around the history of ROP attacks as well as introduce stack pivoting. Next, we describe in detail how we would reconstruct backtrace from a core dump with stack pivoting. First, we use our algorithm to find the starting point of the ROP chain. Then, depending on the type of stack pivoting, we proposed the approach of finding the original stack. We then use this old ESP register value to reconstruct the backtrace. We present our test result of our tool in the next chapter. Lastly, we discussed several corner cases that we did not discuss in detail, and give suggestions about future research.

Chapter 2

Background

2.1 Direct Code Injection

Direct Code Injection has a long history in vulnerability exploiting. Stack smashing attack is one kind of attack in this category. Programming languages like `C` and `C++` do not have a built-in check for accessing illegal memory locations. Specially, when accessing an array in these languages, it is possible to read and write outside the boundary of this array while compiler happily produce the binary for you. This kind of illegal memory access often leads to overwriting partial of stack memory, this is from where the name stack smashing comes.

```c
#include <string.h>
void foo(char *bar){
    char c[12];
    strcpy(c, bar); // Insecure function call
}
int main(int argc, char **argv){
    foo(argv[1]);
    return 0;
}
```

Listing 2.1: A Program Vulnerable to Stack Overflow Attack

Take Listing 2.1 as an example. In function `foo`, the local variable `c` was allocated 12 bytes of memory space. If `argv[1]` is longer than 11 characters (1 byte for the ending `NULL` character), the `strcpy` will do the copy as is, until it reaches the end of the `bar` variable.

This kind of attack is a thing of the past. Nowadays operating systems are able to mark certain memory region (e.g., the stack memory) as "non-executable"; thus, even if the attacker can inject code onto the stack, the code cannot be interpreted as CPU instructions, and consequently, the Direct Code Injection attack is prevented. This technique is called Data Execution Prevention (DEP) on Windows systems, it has been in place since Windows XP Service Pack 2. OpenBSD has a similar technique called $W \oplus X$, in which the operating system will mark writable memory locations as "non-executable", and mark executable memory locations as "non-writable".

2.2 Code Reuse Attack

Because of DEP, even if the attacker find a way to inject the code into memory, it cannot be executed. Because of this, the attacker starts focusing on reusing existing code in memory,

which introduced the Code Reuse Attack (CRA). In general, all modules loaded into the virtual memory space of the running process can be used in CRA, including the `C` runtime library (`libc`).

`libc` provides system calls like creating new processes and reading/writing files, and is usually the stepping stone in the whole exploitation process. The kind of CRA focusing on `libc` is called Return-into-libc attack, as the attacker will modify the return address on the stack and invoke a series of functions by the `RET` instruction.

Later research shows that reusing the whole function code is not necessary in achieving a successful attack. The attacker can also run a chain of a small set of consecutive instructions and deliver the attacker. This introduces the Return Oriented Programming (ROP).

2.3 Return-Oriented Programming

Return-Oriented Programming (`ROP`) is a way to chain small pieces of existing code (called gadgets) together to do arbitrary computation, usually by using specific system call or Application Programming Interface (API).

The name comes from the fact that each gadget ends with an "`RET`" instruction. Instead of modifying the stack to let it contain instructions, `ROP` focuses on modifying the memory region that is pointed out by the Extended Stack Pointer (`ESP`) register, so that the execution flow will be redirected to an arbitrary memory location.

Take Figure 2.1 as an example. The green boxes on executable memory are the normal instructions, and the red ones are some short sequences of instructions that ends with `RET`.

The attacker first gets control of the stack memory and puts addresses of gadgets onto the stack. Then using a normal `RET` instruction (step 1 in the figure), the execution flow will jump to address of gadget A (step 2 in the figure), and so on.

To find the proper gadgets, one can use tools like `mona.py` and `metasploit` to scan the

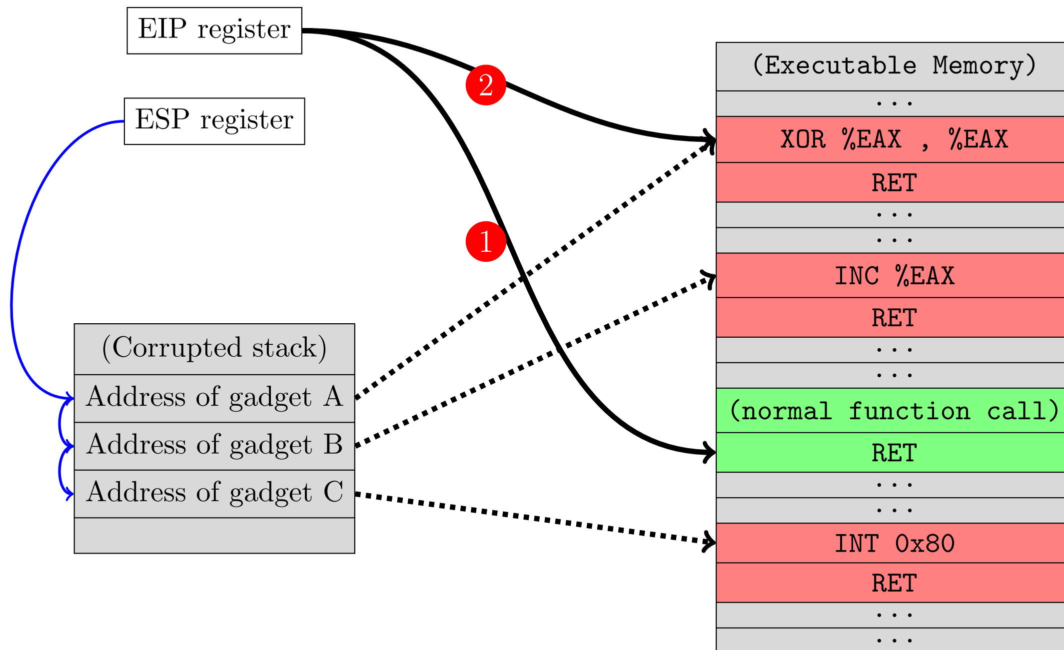

Figure 2.1: Return-Oriented Programming

text segment of a program and all the modules that will be loaded into memory. The basic algorithm for searching gadget involves disassemble the text segmentation, and checking if sequences of instructions can be found which ends with a `RET` instruction. To find more available gadgets, these tools will start disassembling with a certain offset. Listing 2.2 shows an example of this algorithm: without the starting offset, the original hex code can be disassembled into two instructions, but if we ignore the first one byte (hex number `f7`), we can get four completely different instructions, with a `RET` instruction at the end.

```
; Original hex code and disassemble result
f7 c7 07 00 00 00    test $0x00000007, %edi
0f 95 45 c3          setnzb -61(%ebp)

; Same hex code but disassemble with one byte offset
c7 07 00 00 00 0f    movl $0x0f000000, (%edi)
95                   xchg %ebp, %eax
45                   inc %ebp
c3                   ret
```

Listing 2.2: Example of disassemble with a one-byte offset

2.4 Stack Pivoting

It has been proven that ROP is Turing-complete, which means that given a long and complex enough ROP chain, it is capable of any complex computation that a normal programming language can do. But in real world examples, it is hard to craft a chain of gadgets to accomplish even a small task.

In a real world ROP attack, a technique called stack pivoting is often been used to facilitate the delivery of successful attacks. Instead of modifying the stack memory directly, the attacker can prepare a memory region, place the gadget chain into that memory region, and then use a small gadget to modify ESP register value, and fool the Operating System into using that memory region as a new stack.

The fact that the new stack can be anywhere in memory significantly simplifies the construct of the ROP chain. The attacker can feed the ROP chain to the vulnerable process as a normal input value, instead of carefully crafting the chain on the original stack.

2.5 Thread Environment Block

Thread Environment Block (TEB) is a thread-specific in memory data structure that holds information related to that particular thread.

Information that is presented in TEB data structure is filled by the Operating System when a new thread is created. Things like Process ID (PID), Thread ID (TID), and Last Error Number will be stored in this data structure.

Among all this information, there is something that could help in our research, the stack top and bottom address. These addresses are the stack region known to the operating system, not necessarily the corresponding register values (values in `ESP` and `EBP`). In the case of a stack-pivoted attack, the stack range in the TEB data structure is likely not the same as the values in the registers.

2.6 Core Dumps

When a process crashes on a Windows system, if installed with a debugger, the user will get a chance to create a core dump. The core dump is a snapshot of all relevant information of a running process. Typical information that can be found in a core dump includes the complete virtual memory layout, the register values of the CPU. With a proper configuration, the core dump can even contain thread information like the time created and running time of the thread, the unloaded module list, etc.

On a Windows System, a user can use `taskmgr` or `WinDBG` to create dumps from the running process.

Security analysts heavily rely on core dumps to analyze new exploitations and find bugs in software.

Chapter 3

Problem Definition

3.1 Goal and Assumptions

The goal of this study is to infer and recover the call stack right before the execution of the `ROP` chain.

A call stack is the control flow of a program indicated by the addresses of the stack frame. In x86 architecture calling conventions, when a function is called through `CALL` instruction, the memory address of the next instruction will be pushed onto the stack and then the `EIP` register points at the beginning of the function.

Because of this feature, at any point in the execution of a program, it is possible to determine how the execution flow reached the current code, by examining the return addresses on the stack. For example, developers often use the `backtrace` command in `gdb` to show how the process reaches the current breakpoint. The call stack is often represented by a series of addresses that indicate the functions and addresses of call sequences.

```
(gdb) backtrace
#0  func2 (x=30) at test.c:5
#1  0x80483e6 in func1 (a=30) at test.c:10
#2  0x8048414 in main (argc=1, argv=0xbffffaf4) at test.c:19
#3  0x40037f5c in __libc_start_main () from /lib/libc.so.6
```

Listing 3.1: The GDB backtrace command

Listing 3.1 illustrates an example of a call stack. The first line starts with #0, indicating the innermost function call, which is `func2` with argument `x=30` defined in `test.c` line 5. Caller of `func2` is `func1`, whose caller is, in turn, the `main` function.

The reason we want to infer the call stack before the `ROP` execution is to find out how the exploitation works. Specifically, we want to know how the control flow of a vulnerable program can be manipulated to jump to a `ROP` chain. The use of `ROP` to bypass DEP protection is a common practice, and `ROP` is often (as observed by us and others) used as the first step of modern exploitation. From a security analyst's point of view, detecting the `ROP` execution is a good way to know that exploitation has occurred. However, finding out how the `ROP` attack was triggered is critical if we hope to identify the vulnerability in software and infer how the attack happens.

A common method for determining how an exploitation occurs is to get the call stack, e.g., run `backtrace` at the crash point. However, practical `ROP` exploitations almost always involve stack pivoting, which means the current call stack (that runs the `ROP` gadget chain) is not the stack which leads to the execution of the `ROP`.

In the normal running of a program, the calling convention is always honored. This means that each function call will create a new stack frame by pushing the return address and old `EBP` register value onto the stack. This makes it relatively easy to do a `backtrace`: the `EBP` register points to the base address of the current stack frame, the next 8 bytes are the outer frame's base address value; then, another 8 bytes for the return address. But it is

not always the case when ROP is used in an exploitation, specifically stack pivoting is used to start the ROP chain execution.

Loosely speaking, any modification to the ESP register to let it hold another memory address can be called stack pivoting, including all stack related benign instructions like PUSH/POP. The reason for modifying the value in the ESP register is that, after modifying that register, any POP/PUSH/RET instructions will operate on the new stack whose address is determined by the value in the ESP register. The attacker can place multiple gadget memory addresses anywhere in the memory, use stack pivoting to start using that memory range as the stack, and finally chain those ROP gadgets together.

Even if we have the range of the previous stack section by checking the Thread Environment Block (TEB) information in the dump, the stack points (such as the EBP, and ESP) are no longer valid. ESP is definitely being overwritten, and likely EBP as well.

So if we look directly at a crash dump that occurs during the execution of the ROP chain or after, the first thing we will notice is that the ESP register may contain a memory address that is not in the stack range from the TEB (thread environment block). This can be used as proof that stack pivoting has really happened.

We used a set of deterministic and heuristics algorithms to infer what happened before the ROP chain, especially how the stack pivot happens.

Our work is based on the following assumptions:

1. We have a core dump that is generated during (or right after) the execution of a ROP chain.
2. A stack pivot occurred before the execution of the ROP chain.
3. The stack pivoting target is at the start of the ROP chain.
4. The old stack has not been destroyed by the execution of the ROP chain.
5. We have access to the binary and the debug symbols of the vulnerable program.

3.2 Determining the Stack Pivoting Target

During the process of stack pivoting, the final value in the `ESP` register is the top of the new stack. We call this value the stack pivoting target. As per our assumption, the stack pivoting target can be determined by inferring the start of the `ROP` chain.

The start of the `ROP` chain can be inferred from the core dump. The `ESP` register value in the core dump is somewhere below the stack pivoting target (in a higher memory location compare to the stack pivoting target). We can start scanning memory from the final `ESP` register value towards lower memory locations, and identify gadget addresses during the scan.

The general algorithm to determine if the address is a gadget address is described as follows:

- The memory location identified by this value is executable.
- When disassembling that memory location, we can find a `RET` instruction less than *threshold* instructions.

In order to determine a proper threshold, we analyzed the entire loaded Dynamic Link Library (DLL) of Internet Explorer 8 on the Windows 7 SP1 32bit System. Of all the gadgets with 20 less instructions, 88% of them are less than 10 instructions. We use 10 as our *threshold* value.

As the memory region is used as a stack, we are aware that values we meet are not always memory addresses. Some gadget, e.g. "`XOR %EAX , %EAX ; POP %EBX ; RET`" will need to have one byte of compensation on the stack due to the `POP` instruction. We took this into consideration when scanning gadget addresses.

We scanned towards the lower memory, found as many gadget addresses as possible, and stopped at the memory location containing the last gadget address we could find. We used this resulting memory location as the stack pivoting target address.

3.3 Types of Stack Pivoting

This section describes the known stack pivoting methods. The purpose of stack pivoting is to change the stack pointer register to a new memory region (rather than grow/push on the current stack).

The current active stack frame of a running program is captured by the values of the two special purpose registers, `ESP` for the top of the current active stack frame, and `EBP` for the base of the current active stack frame.

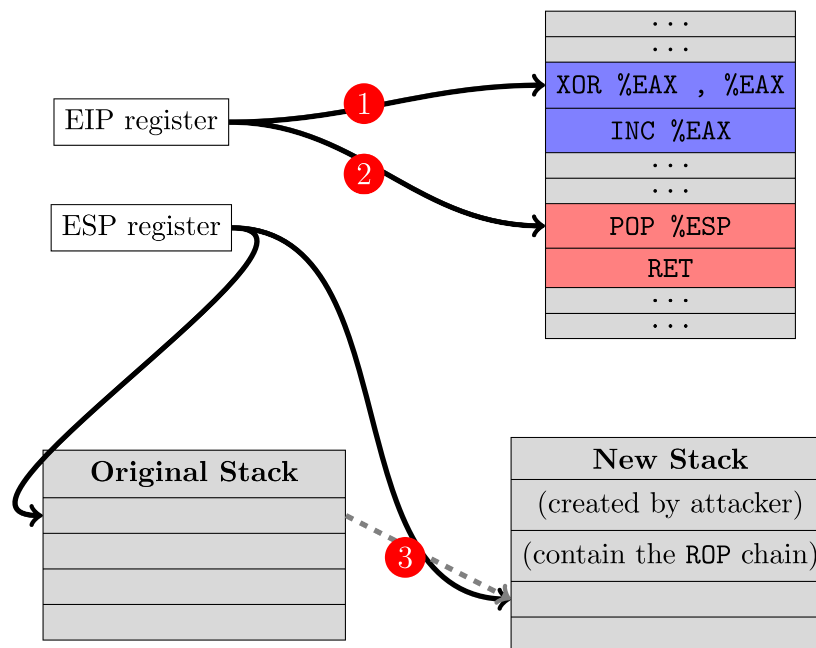

Figure 3.1: How Stack Pivoting Works

In Figure 3.1 we show a typical stack pivot used in an attack. In Step 1, the `EIP` register is pointing at a normal instructions in the memory. As the CPU continues execution, the stack pivot gadget eventually gets executed in step 2. After the `RET` instruction, the `ESP` register will contain the value of an arbitrary new memory address, and thus the stack range will be modified by the attacker.

Stack Pivoting is commonly achieved by changing the `ESP` to point to a new memory

place that is filled with content controlled by the attacker. The location of the "new stack" can be arbitrary memory locations like the heap, loaded dynamic modules, or even inside the original stack. The point of stack pivoting is that it makes it easier for an attacker to prepare ROP chain in a new memory range instead of modifying the original stack.

Many instructions are not affected by the change in the ESP register. Instructions like "MOV/SUB/ADD" after the ESP register value change will work as expected since they do not interact with the stack at all. But instructions like "PUSH/POP/CALL/RET" will start using the memory location pointed out by the ESP register immediately. In most situations the new stack will only contain the ROP chain, so any modification to the ESP register will likely be followed by an RET instruction to start running the ROP chain.

In order to cover all available cases of stack pivoting, we chose Mona.py to generate gadgets that are available for stack pivoting and to group them by their features. The following sections describe popular ways to pivot the stack.

3.3.1 Through POP %ESP

A popular way to change the ESP register value to a new stack region is to trigger a POP operation and assign some value on the stack to the ESP register. For example, the sequence "PUSH %EAX ; POP %ESP " will PUSH value containing in the EAX register onto the top of stack, and then the POP instruction load that same value from the stack to the ESP register.

3.3.2 Through arithmetic operation over the ESP register

The attacker can also use arithmetic instructions to increase/decrease values in the ESP register. Examples of these instructions are: ADD/SUB %ESP , 0x20, and INC/DEC %ESP . If the distance between the current stack and the target stack can be pre-calculated, the attacker can carefully design a chain of arithmetic instructions that involves the ESP register, and

modify its value to a new stack location.

3.3.3 Through an exchange instruction with another register

Controlling the `ESP` register is not that easy. Modifying the `ESP` register should be done in instructions as short as possible since modification of the `ESP` register will immediately affects the results of the instructions related to the stack. But, the general registers like `EAX` does not have that constraint. One can load a value to the `EAX` register, then do some `ADD`/`SUB` operation to get the specific values he/she wants, and finally exchange the value in the `EAX` register with the `ESP` register.

3.3.4 Through other stack operations

Some stack related instructions related to stack operation will implicitly change the `ESP` value. For example, `POP` instruction will increase the `ESP` value by 4 bytes, and `PUSH` instruction will decrease it by 4 bytes. The batch push/pop instruction, `PUSHAD` and `POPAD`, will modify the `ESP` value by 32 bytes.

A variant of the `RET` instruction, `RETN`, accepts an operand as the offset when calculating the return address. While the `RET` instruction will jump to $retAddr$ directly, "`RETN 0x8`" and "`RETN -0x8` will jump to $retAddr + 8$ and $retAddr - 8$, respectfully.

These seemingly harmless instructions can be used by attackers too.

3.4 The Proposed Approach for Finding the Old `ESP` Value

We plan to look at each of the pivoting methods mentioned above and to try to infer the old `ESP` value. When we have a crash dump, we do not really know which pivoting method

is being used. Here we just first assume that we know the method, and describe the plan of recovery from there. In practice, we plan to try all our heuristics and find as much potential old ESP register value as possible.

Of course, we need to use a lot of heuristics in our program. For some of the pivot techniques, we only handle only the simplest situation. For the other techniques, we may get multiple candidates. Our plan is to reconstruct a call stack for each of the candidates and let the user decide which one is correct.

In the end, we discuss a general way to reconstruct the call stack in case there is no potential candidate returned from our heuristic algorithm.

Solutions about different stack pivot techniques are listed in Table 3.1.

Table 3.1: Stack Pivoting Techniques and Solutions

Name	Solutions		
	Deterministic	Heuristic	Fallback solution
POP instruction	✓	✗	
Arithmetic operations	✗	✓	✓
XCHG instruction	✗	✓	
Other stack related operations	✗	✗	

3.4.1 POP %ESP

This case is relatively easy. Our assumption is that we know the beginning address of the new stack, so we know what memory address will be popped from the stack to the ESP register. Because the pivot is achieved by a POP instruction, it means the new stack address is likely still on the old stack. So the deterministic solution is to search for the value of the pivoting target address in the old stack region. Once we find the memory address that contains the stack pivot, we know that the ESP was once pointed at the next higher memory address, and we discover the old ESP value.

For example, in the case of "PUSH %EAX ; POP %ESP ", the address was passed by the EAX register, and it was pushed on the stack (which means it is in the memory) before the POP instruction get executed. Therefore, the old ESP register value can be identified by searching the new stack address on the old stack.

3.4.2 Arithmetic Instruction over ESP register

The arithmetic operation on the ESP register makes a "short distance jump" to a new memory address relative to the current ESP value.

To restore the old ESP value, first we need to scan all loaded modules in core dump to get potential arithmetic gadgets used for stack pivoting. Through the WinDBG's API, we can get the memory range for each loaded modules, and then we can disassemble the binary content to get the CPU instructions.

Each of these potential gadgets has an offset value. For example, "INC %ESP " will have an offset of +1, and "SUB %ESP , 0x08" will have an offset of −8.

As per our assumption, we assume the original stack memory is not destroyed by the execution of the ROP chain. Therefore, we do not consider the possibility that the attacker placed the ROP chain inside the original stack region, because this would likely have destroyed the original stack. In fact, in reality if the attacker is able to place the ROP chain on the stack, it is highly possible that stack pivoting is not needed at all.

The default stack reservation size used in the Windows system is 1MB, so we consider only gadgets with an absolute offset value greater than 0x20000000, because any number less than this value will have less than a 40% possibility of "jumping" outside of original stack region. The calculation of this number comes from the following inequity:

$\frac{minOffset}{1024^3 bytes} \geq 0.4 \rightsquigarrow minOffset \geq 0\text{x}20000000.$

We treat all of these gadget as potential gadgets. For each of these gadgets, we calculate the potential old ESP register value. Based on this old ESP register value and the old stack

memory information, we are able to infer the details of the stack frame containing the `ESP` value. Then we validate if the potential gadget is indeed from that function.

3.4.3 Exchange Instruction

As was discussed previously, controlling a general purpose register like the `EAX` is much easier than controlling the `ESP` register directly.

There is one heuristic we can use in this scenario: "`POP %EAX` ; `XCHG %ESP` , `%EAX` ". This is similar to the case we discussed in Section 3.4.1: the attacker modifying the `EAX` register using `POP`, and exchanging the values in `ESP` and `EAX` register.

Since the value in `EAX` was `POP`'ed from the stack, the literal value is likely still on the old stack. We can scan the stack range presented in the `TEB`, find any literal values equal to the start address of the new stack, and use the next higher memory as the starting point of our call stack recovery.

3.4.4 Other Stack-related Instructions

We do not consider these operations as viable to accomplish a successful stack pivot, thus we ignore this case in our research.

3.5 Challenges in Finding the Old `ESP` Value

The previous section describes basic approaches to finding the old `ESP` register value. Here we discuss the challenges we discovered in the process.

3.5.1 An inaccurate Stack Pivoting Target Value

A stack pivoting gadget has a target value, which is, in most cases, the starting point of the gadget chain. Our research depends on the assumption that we have already gotten the accurate value. In reality, this is not always the case.

To find the starting point of the gadget chain, we use the following algorithm:

- Get the current `ESP` register value in the core dump.
- Examine the lower memory addresses and find as many potential gadget addresses as possible.
- Consider the smallest memory address which contains a potential gadget address the starting point of this gadget chain

There are some flaws in this algorithm that may result in an incorrect stack pivoting target value. The problems in this algorithm are summarized below:

Determining if an Address points to a Gadget

When given a memory address, we first need to determine if the permission of the target memory is executable or not. If the target memory address is not executable, it is guaranteed not to be a gadget address.

If the memory address is executable, we will disassemble it from that address and get up to 10 instructions. We may get fewer instructions if not all of the addresses are executable. Nevertheless, we will check if there is any control transfer instructions (e.g. `RET`, `JMP`, `JNE` etc.) in the following instructions. If there is, we consider the memory address is pointing to a gadget.

We use the number 10 as the maximum number of instructions to be checked for control transfer instructions. This number is due to the fact that most of the gadgets contains

fewer than 10 instructions. To determine the upper limit of gadget length, we use `Mona.py` to generate `ROP` gadget list from Internet Explorer and all its loaded modules (DLLs), and then group the gadgets together according to their length. Of all the gadgets, 88% of them contains fewer than 10 instructions.

Gadget Chain Prefixed by NOP instructions

The gadget may be intentionally prefixed by a number of the `RET` instructions serving as the No-Operation (NOP) gadget. In this way, when doing stack pivoting, the attacker does not need to modify the `ESP` register to reflect the exact starting point of the `ROP` chain. One can modify the `ESP` register value to an address in the range of NOP gadgets, and still gets the actual `ROP` chain running.

In this case, when we look for the start of `ROP` chain, we may find the first NOP gadget as the starting point.

Our solution to this flaw is that, when we look for the stack pivoting target value on the original stack, we allow a range instead of just a number. If we have the start of the gadget chain at location $addr_1$, we will look for a number on the original stack that is in the range of $(addr_1 - delta, addr_1 + delta)$. The $delta$ we use is `0x10000`.

3.5.2 Multiple Potential Value vs. No Potential Value

Ideally, when using our heuristic to search the original stack memory, we will get one result and reconstruct the backtrace using that old `ESP` register value.

But it is possible that we will find multiple values that match our criteria or no results at all.

If we find multiple values, we will construct the backtrace for each value and use a calling graph to filter some incorrect cases; we will then output the remaining backtrace to let the user decide which one is correct.

If no result is returned, we will use a fallback algorithm to scan the whole original stack memory region, as described in Section 3.6.

3.6 Plan for Recreating the Call Stack

This subsection describes the plan to recover the whole call stack when we have the stack section memory range, the top of the stack (`ESP`) and the binary executable of the running process, along with the debug symbol. Essentially we are creating a "`backtrace`" that is aware of possible stack pivoting and creates the trace accordingly.

3.6.1 Identifying Return Address

A stack consists of multiple stack frames, each frame represents a function call and contains all of the local variable information. One way to identify the stack frame is to identify the return addresses, which were pushed on the stack for each function call. Therefore, the first effort is to scan the stack section for all possible return addresses.

A value is considered a return address if it meets these criteria: 1. It points to an executable memory location and 2. There is a `CALL` instruction right before that memory address.

Therefore, an intuitive approach is to scan the memory range from the bottom to the memory address pointed out by the `ESP` register, to identify all of the return addresses, and then to create a call stack by consulting the debug symbols.

Unfortunately, things are a little bit complicated. The values on the stack can be either a memory address (the return address of function calls) or literal value (the arguments of function calls). The literal value on the stack can accidentally be equals to a memory address that point to executable memory locations, or in some situations, a function pointer was passed to another function as a callback function.

Fortunately if we have the debug symbol, we are able to create a calling graph containing every pair of callers/callees in the program and every loaded modules. Using this calling graph, we can filter most of the false results. For example, if we find that `func1` is calling `func2` by scanning the stack, but we see that there is no such function invocation in the calling graph, we can be sure that is a false result. Details about creating this calling graph will be discussed in a later chapter.

3.6.2 Solution Wrap-up and the Fall-back Plan

If we find the old `ESP` register value through our `ESP` value finder, we can scan from that address towards higher memory addresses (from top to bottom) and identify stack frames along the way, until we reach the outermost function call.

The plan is straightforward, for each return address we identify, we consult the debug symbols to translate it into function names, and optionally parse the function variables pushed onto the stack.

As we mentioned above, our method of inferring the old `ESP` register value could return no candidate at all. In this case, we will fall back to scan the whole stack range, but from higher memory addresses towards lower memory addresses (from bottom to top). We can use the calling graph and stop at a position where the caller-callee does not make sense.

The fall-back plan will start scanning from the bottom of the stack to the top. For each identified stack frame, we ask the question: does the outer function (the function of previous identified stack frame) has an edge to the current function in the calling graph? We stop at the point when there is no such invocation relationship by consulting the calling graph, and output the stack information as our backtrace result.

Details about how to create the calling graph will be discussed in Section 4.1.

Chapter 4

System Overview and Implementation

In this chapter, we discuss the design and implementation of our system.

Figure 4.1 shows the overall design of the system. Details about each component will be discussed in the following sections.

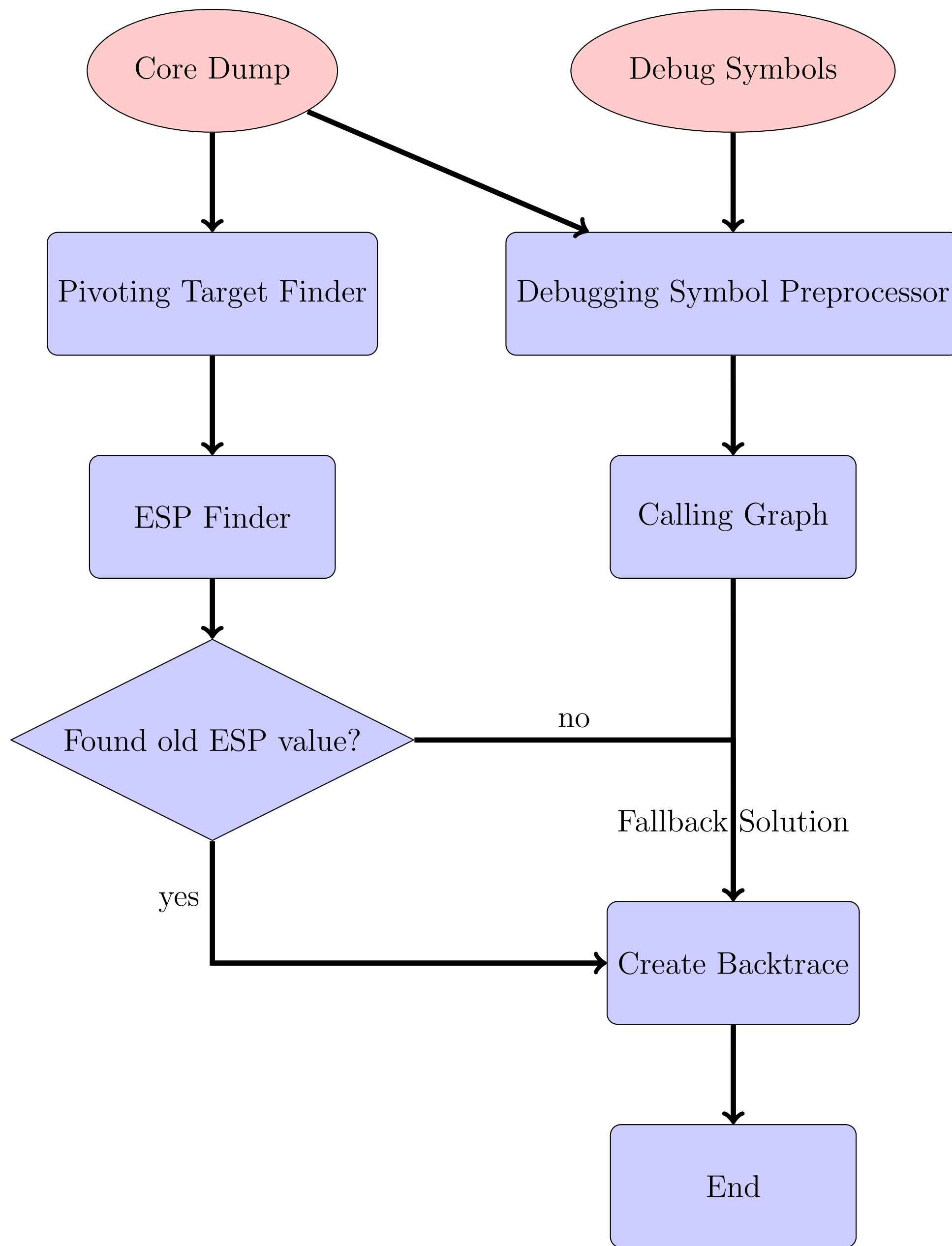

Figure 4.1: System Design

4.1 Debugging Symbol Preprocessor

In order to fully reconstruct the call stack, we need to pre-process the debug symbol together with the core dump to construct a calling graph.

The calling graph is a directed graph that contains potential circles. Each edge of the graph represents a calling relation. Take Listing 4.1 as an example.

```c
#include<stdio.h>
int fib(int n) {
    switch (n) {
        case 0:
        case 1:
            return n;
        default:
            return fib(n-1) + fib(n-2);
    }
}
int main() {
    int seq = 0;
    printf("Input␣the␣sequence␣number:\n");
    scanf("%d",&seq);
    printf("Fibonacci␣series\n");
    for (int i=1; i<=seq; i++) {
        printf("%d\n", fib(i));
    }
    return 0;
}
```

Listing 4.1: A Simple C Program to Calculate Fibonacci

In this example, there are roughly four functions involved: `main`, `printf`, `scanf` and `fib`.

The `fib` function is a recurring function which will call itself an arbitrary number of times. The corresponding calling graph is shown in Figure 4.2.

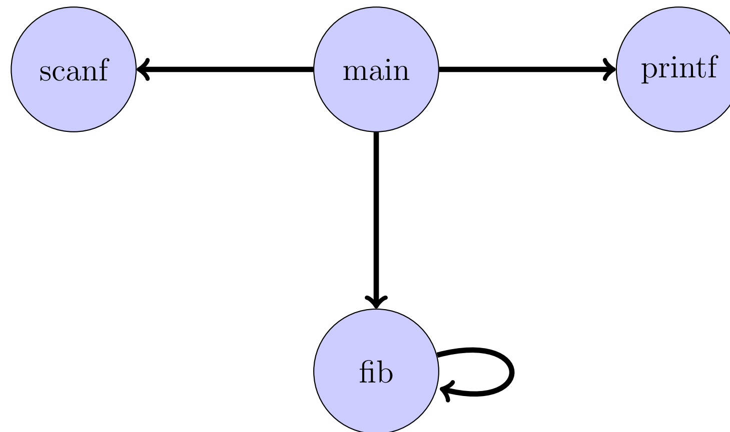

Figure 4.2: A Calling Graph

Bear in mind that we do not intend to walk through all possible branches in a function body as a symbolic virtual machine does. As long as there is a `CALL` instruction in the function body, we will add an edge from caller to callee. It is possible that a function call is surrounded by an `if (false)` statement and is, therefore, never actually been called, but in our research, we will add an edge to our graph.

Since we do not have access to the source code, the calling graph can be extracted only from the core dump and the debug symbol. The process can be described below.

1. Disassemble all memory ranges of loaded modules in core dump

2. For each instruction:

   (a) If this is a `CALL` instruction, create a tuple $(addr_1, add_2)$, meaning "memory address 1 contains a `CALL` instruction to memory address 2"

   (b) Consult the debug symbol to translate $(addr_1, addr_2)$ into $(func1, func2)$

   (c) Store the result into the `SQLite` database to allow for faster retrieval

4.2 The ESP Value Finder

For each stack pivot method, we have a proposed heuristic to infer the old `ESP` register value. We will need to run all finders since we do not know which method was used in stack pivoting.

4.3 The Stack Frame Analyzer

By running the `ESP` Finder, we are able to find the original stack pointer value. We will run the stack frame analyzer starting with that address to construct the actual call stack.

We will use the generated calling graph to filter the incorrect results, so that the researcher do not need to manually confirm the correctness of the backtrace result.

Chapter 5

Implementation

5.1 Debugging Symbol Preprocessor

The preprocessor was written in Python. We use the `pykd` Python module to access the `WinDBG`'s C++ interface. `WinDBG` offers an easy way to disassemble the memory content; therefore we can use it to create the calling graph easily.

We first list all loaded modules in the core dump. A module is essentially a loaded Dynamical-link Library (`DLL`) or the program binary itself. Each module comes with a memory region representing the binary of that module. The preprocessor will then disassemble each module.

Because `WinDBG` does not provide an easy way to get the text segment range of a loaded module, we need to do disassembling in a dumb way. That is, we will first check the permission of the memory address to be disassembled. If the memory is not executable, we will ignore it and continue disassembling the remaining memory content.

The result calling graph will be stored in an `SQLite` database for latter querying.

5.2 ESP Finder

The ESP Finder is a set of Python classes which implement the infer method. This method will return a list of memory addresses containing the potential old ESP value. If the finder cannot find a potential value, an empty list will be returned.

In order to ease the implementation of the ESP Finder, we use meta programming in Python to automatically register new Finders once they are coded.

5.3 The Stack Frame Analyzer

The stack frame analyzer will look for return addresses on the stack memory range and reconstruct the call stack.

The analyzer will need to query the calling graph data in the SQLite database and filter out the false results. This is especially important in the situation when no old ESP value was inferred by the ESP Finder and we have to use the stack range specified in the TEB information.

Chapter 6

Evaluation

In order to test our program, we manually generated several core dumps of Internet Explorer 8 on the Windows 7 SP1 32bit operating system, and tested all stack pivoting methods. Our result is summarized in Table 6.1.

6.1 Manually Creating Core Dumps

In order to test our program, we generated several core dumps using `pykd`, which is a Python library used to interact with `WinDBG`'s C++ Application Programming Interface (API).

We use Internet Explorer 8 on Windows 7 SP1 32bit as our victim program. To create core dumps, we first launch the program, and then attach our program to the running process using `pykd`. The debugger will inject into the running process as a separate thread. As the side effect of attaching to a running process, the whole program will pause and stop at the default breakpoint at `ntdll DbgBreakPoint`.

Then we will find the return address of all paused threads and place a stack pivot gadget at that return address. Before resuming the process, we will backup the backtrace result for later comparison.

We did not carefully craft our stack pivot gadget so the stack pivoting target is likely a random number (e.g. the "`POP %ESP` " stack pivoting gadget will pop a random number at the top of original stack into the `ESP` register). So when our program detached from the process and let it resume running, the process will likely return to a non-executable random memory location, which in turn will crash the process. We create a core dump using `taskmgr.exe` after we get the alert window indicating that the program has a memory access violation.

6.2 Evaluation Results

Since the program was crashed because of a memory access violation, the value of `ESP` register in the core dump is considered the stack pivoting target address.

We use this target address as the input to infer the old `ESP` value. Of all the stack pivoting methods, only the one using an `XCHG` instruction cannot recreate the backtrace.

After getting the backtrace, we compare it to the backup backtrace obtained right before the core dump was created to verify the accuracy of our tool.

Regarding the accuracy of the tool, if they meet either of the following criteria, we consider the result accurate.

- Their length is the same, and only the innermost function call is different.
- Their length is not the same; the shorter one excluding the innermost function is the subset of the longer one.

With the above rules, we consider the following pairs of backtrace results as accurate:

- $func_{fish} \rightarrow func_{cat} \rightarrow func_{dog}$
  $func_{fish} \rightarrow func_{cat} \rightarrow func_{husky}$
- $func_{fish} \rightarrow func_{cat} \rightarrow func_{dog}$
  $func_{fish} \rightarrow func_{cat} \rightarrow func_{husky} \rightarrow func_{corgi}$

The results are shown in Table 6.1.

Table 6.1: Evaluation Results

Tested Pivoting Gadget	Result		
	# of Core dumps	# of Backtrace created	Accurate?
`POP %ESP`	2	2	2
`INC %ESP` , $0x40000000$	2	2	2
`POP %EAX` ; `XCHG %EAX` , `%ESP`	2	2	2
`XCHG %EAX` , `%ESP`	2	2	0

6.3 Discussion of the Results

It is no surprise that the core dump generated by "`XCHG %EAX` , `%ESP` " cannot generate the accurate backtrace. Our heuristic for this stack pivoting method assumes that the other register involves in the `XCHG` operation get the value `POP`'ed from the stack, so the stack pivoting target should remain on the old stack region. If we do the stack pivoting use only the `XCHG` gadget, our heuristic algorithm will return no old `ESP` register value, and the fall back solution gives an inaccurate backtrace.

Chapter 7

Discussion

There are many issues in our research that have not been solved. We discuss these issues in this section.

7.1 Correctness of the Pivot Target

We refer to the target address of a Stack Pivot sequence as the Stack Pivot Target. Our research rely on the assumption that the pivot target address can be known before running the program.

The pivot target is usually the beginning of the `ROP` chain, but this is not always the case. Even this is true, the automate tool used to recognize the start of `ROP` chain may give an incorrect result. This in turn may affect the correctness of our backtrace result.

7.2 Relying on the Debugging Symbol

Our research heavily relies on the debug symbols to create the calling graph, as well as the result backtrace.

For programs provided by Microsoft, e.g. Internet Explorer, we have a public accessible

debug symbols server. But for other program, e.g. Adobe Reader, we do not have these debug symbols, which means we cannot create a calling graph. We have to use the fallback method to construct backtrace. Furthermore, the resulting backtrace will contain only offset numbers to loaded modules, instead of actual function names.

Chapter 8

Conclusion and Future Works

In this paper, we developed a full working program to automate the analysis of core dumps which contains Stack Pivoting. We discussed the details of `ROP` and Stack Pivoting, analyzed the features of different stack pivoting methods, and finally designed and implemented a program to automate the whole process. We also tested our program against manually generated core dumps to prove that it worked.

This research may be improved in multiple ways. We list possible future work in the following sections.

8.1 Deterministic Algorithm to Search for the Old `ESP` Register Value

Our current solution to stack pivoting, except for the `POP` instruction case, are all heuristic solutions. We have preliminary thought of solving the rest of the cases deterministically by integrating an assembly language emulator into our tool and find the old `ESP` register value with higher confidence.

Taking the `XCHG` stack pivoting gadget as an example: "`XCHG %EAX , %ESP` ", the deter-

ministic solution would be finding how the specific value loaded into `EAX`.

We can go back several steps and find an instruction that gives `EAX` a concrete value. For example, "`MOV %EAX , 0x08`" will give `EAX` a value of `0x8`, and "`XOR %EAX , %EAX` " will result in `EAX` register containing zero. Then we follow the instructions until we reach the potential pivot gadget and we see if the `EAX` register contains the stack pivoting target address.

8.2 Other Stack Pivoting Scenarios

8.2.1 Chain of Gadgets to do Stack Pivoting

In our research we consider only those situations in which the stack pivoting is done by one gadget (with possible small numbers of prepare gadgets). But it is possible that the attacker might use an arbitrary length of a gadget chain to do stack pivoting.

8.2.2 Stack-related Instructions as Stack Pivoting Method

We think it is not viable to use normal stack-related instructions to conduct stack pivoting. But a carefully crafted chain of stack pivoting target do have the possible of resulting in successful stack pivoting.