

1 An Introduction to Return Oriented Programming

2 Maximilian Heim¹

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

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

6 **Keywords:** ROP · Return Oriented Programming · Buffer Overflow · Binary
7 Exploitation

8 1 Introduction

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

18 2 Gadgets

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

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

Listing 1: Exporting gadgets with `ROPgadget`

33 `ROPgadget --binary ./vuln --nojob > gadgets`

34 This command produces an output with results similar to this.

Listing 2: Output of `ROPgadget`

```

35 0x08059ee3 : mov word ptr [edx], ax ; mov eax, edx ;
36     ret
37 0x08071e4e : mov esp, 0xc70cec83 ; ret 0xffe0
38 0x0807faa3 : sti ; xor eax, eax ; ret
39 0x0808b285 : pop edx ; xor eax, eax ; pop edi ; ret
40 0x080539e7 : mov esp, 0x39fffffd ; ret
41 0x0804b8d4 : xchg eax, esp ; ret
42 0x08095aef : mov esi, eax ; pop ebx ; mov eax, esi ;
43     pop esi ; pop edi ; pop ebp ; ret
44 0x0806ceec : pop es ; add byte ptr [ebx - 0x39], dl ;
45     ret 0xffd4
46 0x0804a444 : or eax, 0xffffffff ; ret
47 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. 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 arbitrary memory locations. 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. 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` stand for an interrupt, the interrupt `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 arbitrary programs. In combination with `pop`, `mov` and other instructions we can specify the concrete system call. 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 i was able to find at least a few suitable candidates. 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 using them.

3 Theory

3.1 Stack

The following graphic 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.

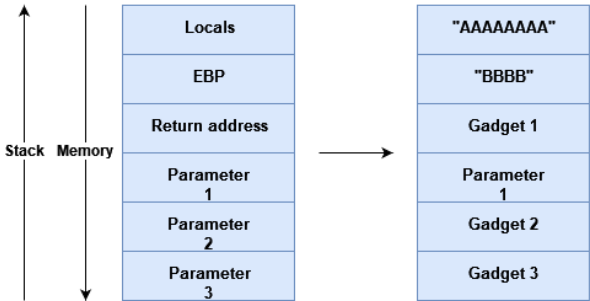


Figure 1: The stack when injecting the payload

3.2 ROP Chain

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.

Listing 3: The Target Program

```
114 1 #include <stdio.h>
115 2 #include <string.h>
116 3
117 4 int main(int argc, char *argv[]) {
118 5     char buffer[8] = {0};
119 6     if (argc != 2) {
120 7         printf("A single argument is required.\n");
121 8         return 1;
122 9     }
123 10    strcpy(buffer, argv[1]);
124 11    return 0;
125 12 }
```

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 compilation command

```
138 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 bytes are needed to override the base pointer `ebp`
6. Determine position of a writable data segment

- 147 7. Generate payload using the gadgets according to the the abstract payload while
 148 making sure gadgets dont interfere with our desired program state. This step can be
 149 done using Python which we will show in a later section [Lst. 6](#)
- 150 8. Insert payload into target using a vulnerability

151 **Goal and abstract payload** After specifying the goal and possibly simplifying it we have
 152 to write a list of instructions and arguments that achieve the goal, for this its favorable
 153 to directly use the format of the final payload except for using instructions instead of
 154 addresses as this will then allow to simply insert the found gadgets into this abstract
 155 payload. For the example in this paper we want to open a shell, for that the simplest way
 156 is to execute an `execve` system call. The following program state [Fig. 2](#) has to be achieved
 so the interrupt `int 0x80` causes a shell to be opened.

Registers		Memory	
EAX	0x0B (11 ₁₀)	0x080e5020	"/bin"
EBX	0x080e5020 (.data)	0x080e5024	"//sh"
ECX	0x00 (0 ₁₀)	0x080e5028	"\0" + ???
EDX	0x00 (0 ₁₀)	0x080e502C	????

Figure 2: Required program state for the `execve` syscall

157

158 Extract and search gadgets

159 **Determining the padding** Compilers optimize stack alignment and without providing
 160 options to change that the simplest way to determine the padding required is to test the
 161 program until it crashes with a payload increasing by 1 word in each iteration. This can
 162 be automated in a Python script [Lst. 5](#). This script applies the method mentioned above
 163 with the `os.system` function. The return value of that function is the exit code of the
 164 program that has been executed and is either 0 when the execution ended without any
 165 errors and non 0 when an error or exception occurred during startup or runtime. This
 166 means we can increase the input by "AAAA" in each iteration until the return value is non
 167 zero. At this point the base pointer `ebp` has been overridden causing the program to crash.
 168 Now reducing the padding by 1 word results in the correct amount.

Listing 5: A Python Script to Determine the Required Words

```

169 1 import os
170 2 import sys
171 3
172 4 def determine_word_count(target_program_path: str, buffer_size: int) -> int:
173 5     for words in range(1, buffer_size + 64):
174 6         if os.system(target_program_path + ' ' + 'AAAA' * words):
175 7             return words - 1

```

```

176 8     return -1
177 9
178 10 if __name__ == '__main__':
179 11     word_count = determine_word_count(sys.argv[1], int(sys.argv[2]))
180 12     print('Required words: ' + str(word_count))
181 13     print('String: ' + 'AAAA' * word_count)

```

we have overwritten the return instruction pointer with "BBBB". Reducing the amount of A's by 4 the first gadget overrides the return address of the function call and on `ret` the execution gets redirected to the first gadget.

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 arbitrary values as seen in line 2 [Lst. 6](#). 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. 6](#). 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 6: How to use `struct.pack`

```

196 1 from struct import pack
197 2 p = bytes('AAAAAAAABBBB', 'ascii')
198 3 p += pack('<I', 0x0802840)
199 4 print(str(p)[2:-1])

```

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 whether 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

Luckily we had to disable several security mechanisms to make this attack possible, especially

7 Discussion

Sources:

217 [https://www.exploit-db.com/docs/english/28479-return-oriented-programming](https://www.exploit-db.com/docs/english/28479-return-oriented-programming-(rop-ftw).pdf)
218 -(rop-ftw).pdf [https://guyinatuxedo.github.io/5.1-mitigation_aslr_pie/index](https://guyinatuxedo.github.io/5.1-mitigation_aslr_pie/index.html)
219 .html

220 References

221 [ES] Saif El-Sherei. Return oriented programming (rop ftw) - exploit-db.com.