Return Oriented Programming

Anonymous Submission

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 Gadgets are code segments that sit before a `ret` instruction, that is an instruction that uses the address on the stack to return to a previous stack frame and therefor a previous level in the call hierarchy. This means we can arbitrarily chain these gadgets and achieve arbitary code execution if we find gadgets for our purpose.

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. We could do 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. Using the following command Lst. 1 under Linux we can dump all gadgets to a file and search in it using regular expressions, `ROPgadget` can be found in most package managers or can be downloaded directly from https://github.com/JonathanSalwan/ROPgadget.

Listing 1: Dumping all gadgets into a file

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

Using 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, 0x39ffffd ; 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 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 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. If for example we want to write some value into ecx we could use 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, we could use something like this to accomplish that: `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 arbitary 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`.

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}retn`
- int 0x80 → `^.{0,20}int 0x80`
- xor eax, eax → `^.{0,20}xor eax, eax.{0,20}retn`

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

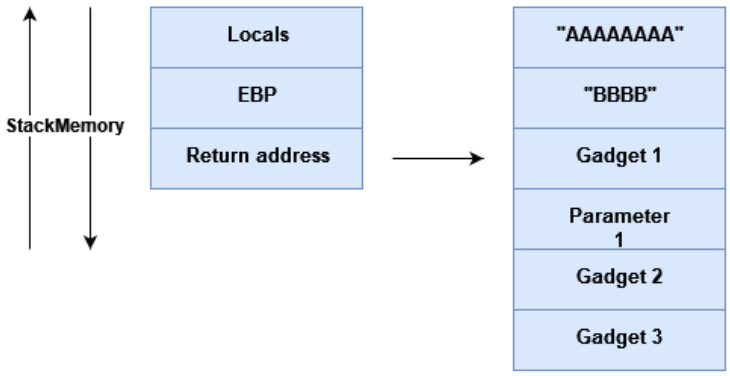

Figure 1: The stack when injecting the payload

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

```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 use the following command to compile the target program

Listing 4: The compliation command

```
gcc -o vuln -g -m32 -D_FORTIFY_SOURCE=0 -fno-pie -fno-stack-\
  protector -static vuln.c
```

4.2 Phases of developing the attack

Phases The attack consists of several phases

1. Specify concrete goal with required program state and instructions
2. Generate desired list of instructions and arguments (abstract payload)
3. Extract gadgets using tools
4. Search gadgets for instructions
5. 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. 5
6. Insert payload into target

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.

Extract and search gadgets After extracting the gadgets using one of the above mentioned methods we can search for gadgets

As described above you should

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. 5. 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. 5. 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 5: 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])
```

5 Results

6 Protection

7 Discussion