Assignment 2 - BASC

BINARY ANALYSIS AND SECURE CODING - UniGE

Kevin Cattaneo - S4944382

Index

Extract

“In this assignment, you'll practice your skills in shellcoding and basic exploitation. We provide the executable programs running on the remote server, so you can reverse them and try your exploits locally. However, the flags are only present in the remote services, so your exploits must work remotely too to capture the flags.”

BOF-Demo

Tool used: pwntools

For this exercise I started from the one given in class as a template. After that I used pwntools and Python to perform the following exploit.

As first I received the message until reaching the very end before asking for the input, and as input I send a line that contains 88 letter ‘a’, that was found via cyclic (that allows to build strings for see at which offset we overwrite critical section of our program, that is the old RSP); this string of 88 bytes allows to identify the starting to put the jump to a pre-defined address of the JMP_RSP instruction that allowed me to return in the stack and execute the shellcode created built-in by pwntools via `shellcraft.sh()`.

After spawning a shell in remote, I cat the flag.txt, obtaining:
$ cat not-real-flag.txt
In real challenges you'll find a "flag.txt", or similarly named file, containing the actual flag.

BOF-101

Tool used: pwntools

In the first place I received the message until the output of the x address, that I will use as offset. From that I played with offset looking at the stack behaviors during several runs. Similarly to the BOF-Demo, I position myself on the cell that represents the return address, in that I put the shellcode again generated by pwntools via `shellcraft.sh()`.

After spawning a shell in remote, I cat the flag.txt, obtaining:
$ cat flag.txt
BASC{Congratz_U_3Xpl0it3d_your_f1r5t_BOF}

Smallcode x64

Tool used: pwntools, nasm

As first operation I printed the length of shellcode crafted by pwntools, discovering that it is greater by far with respect to the maximum requested.

Then I started constructing a shellcode manually, compiling several times the code with the nasm assembler and the linker, together with a lot of research online with referrals to https://www.exploit-db.com/exploits/46907. After obtaining the binary shellcode, I run objdump -d <name_of_binary> to achieve the bytes values that then I put into a byte string on the python script with pwntools.

Since I could provide input at maximum 25 bytes, I also provided two versions of code. One that is long 23 bytes + terminator, that uses push and pop, and one of 24 bytes + terminator that uses the mov instruction, that is coded into a longer byte string (in the code I call this “not optimized”).

After spawning a shell in remote, I cat the flag.txt, obtaining:
$ cat flag.txt
BASC{0pT1miZin9_5h3llc0d3_iS_n0T_s0_H4rD}

No syscalls x86

Tool used: pwntools, nasm (try), gdb + pwndbg plugin

The main idea behind this exploit is to have some self-rewriting code, that acts by re-writing two NOP bytes, previously inserted, with the interrupt instruction INT 80 that trigger the interrupt to make a syscall in 32 bit system, after pushed the stack with the correct values to make an execution of /bin//sh. Since I need to jump back in the instructions, I also need to obtain the correct stack pointer (esp) somewhere and to do so, I use a call instruction to another label to get it.

The first trials of this exercise have been done by writing manually assembly with referrals to https://www.exploit-db.com/shellcodes/49768. For some motifs, I could not get to have the code working, in particular I continually had SEGFAULT when writing on the content pointed by the address contained by the esp, even if I was sure that it was correct. After a lot of hours I decided to give up and, after other research and comparison, I passed directly on pwntool script, writing directly in the asm() function the assembly code, and of course the syntax was a little different from the one used by nasm.

To better inspect the behavior of my code, I also used GDB with the pwndbg plugin, which showed me better the memory and register status during the run. And also I discovered that my non-working solution with nasm, worked here... 🙂

After realizing that the writing was correctly working, I crafted the shellcode to spawn the shell. In particular, as first instruction I call the label to get the stack pointer to the following instruction (that is the start of *shell* label), following the exercise suggested in class, that will be used to jump back and then I reserve two bytes by putting NOP (0x90) that can be overwritten later. After that I pushed the value to make the execution of the characters of the string /bin//sh, splitted in two pushes, since the maximum pushable is 4 bytes (so /bin, and //sh). Then I jumped back to the overwritten bytes and so the shell is executed.

After spawning a shell in remote, I cat the flag.txt, obtaining:
$ cat flag.txt
BASC{s4ndb0X1n9_AiNt_3a5y}

Optional challenges

Note: the following **optional** challenges have been completed during the given extended deadline week (so after the former deadline, still before the secondary one).

Going-up

Tool used: pwntools, online disassembler and hex-to-int converter
Disassembler: https://defuse.ca/online-x86-assembler.htm#disassembly
Converter: https://asecuritysite.com/principles/numbers01

The idea of this program is to provide 4-bytes increasing integers that will be converted into machine code that will be executed, so the code of a shell must be provided to achieve the objective.

To exploit, I started from the asm referred at this link https://shell-storm.org/shellcode/files/shellcode-827.html, and from that I constructed different sets of instructions of 4 bytes size. To achieve that I have to try different combinations and then check the integer number, focusing on one instruction at a time.
So the flow can be described as follows:
I have the one-byte instruction: push eax.
Then I fill the other three bytes with three nop instructions.
And then I disassemble them with the online disassembler, and after that I check the resulting number in little-endian. Finally I take note of the number.
But maybe the combination found is not the one that suits the best, that's because the number found must be greater of the previous one but smaller of the next one (that I will compute afterwards (*)). So rearrangement may be necessary.
Indeed the combination described in the example with respect to the code:

```
push    eax
nop
nop
nop
```

Is not valid, and I need to rearrange it as follows:

```
nop
nop
push    eax
nop
```

And I take note of the number -1873768304.
(* that also caused a lot of back and forth in the combination arrangement by trying to swap crafted instruction (e.g. nop) within the combination of the different sets).

Another major problem was also caused by the two push instructions to push the "/bin/sh" string into the stack, since the single push instruction occupied 5 bytes each (4 bytes for the half string + the push instruction byte). To obtain a correct result I have to build up a multiplier of 4 bytes, so that I can "split" them in different integral set of 4

bytes; but also I can't insert any instruction between the two pushes otherwise I would break the final result, the only way is (as far as I see) to add two instructions, one before and one after both of one-byte long. As a result I obtained three sets (so three numbers representing the pieces) out of those pushes.
And of course they have to match the rule "greater-smaller" so I need to find the right instruction, that is discovered to be respectively a nop and an inc.

And more or less the same pattern has been applied to the other instruction till the end...or not.
Indeed at the very end one strange error occurs: EOF was raised after telling that 2421885872 > 2425389261. After doubting my own math experience I realized that overflow surely was occurring since the maximum positive value that I could provide to a signed int was 2147483647 (2^32/2). Lucky that was solved by rearranging, again, the instructions within the two final sets and also by substituting one nop with a pop in the last block
The full shellcode split in sets can be found in the exploit file, with some comments and also the two final non feasible sets that gave overflow.

After taking note of all the integers, I put them into an array, I appended the terminal value -1 and then I looped on it and passed the inputs to the program, making a shell spawn.

After spawning a shell in remote, I cat the flag.txt, obtaining:
$ cat flag.txt
BASC{ggWP_I_C_U_sorted_it_0uT}

Going-up (another challenge)

After accomplishing the going-up challenge, I tried the same work on another smaller shellcode, that was the same referral as the no_syscall_x86 challenge: https://www.exploit-db.com/shellcodes/49768.
Unfortunately I got into a stall where I couldn't get out:

```
nop
push   0x0068732f --> problematic
push   0x6e69622f
inc ebx
N = 1932486800 --> too big
N = 795371635
N = 1131309410
```

The problem was caused by the fact that the first number out of that set was already too big in comparison with the following two: I could get to make the third greater but not the second, since it was pretty fixed to that integer value. I got into a stall but I left the idea in the script exploit_going-up_STALL.py.

Blockchain

Tool used: pwntools, online disassembler
Disassembler: https://defuse.ca/online-x86-assembler.htm#disassembly

The idea of this challenge is to provide bytes that represent integral instructions that then will be split in blockchain blocks of 5 bytes each. It is not needed to loop over and over 5 bytes each time (more or less like the integer flows of Going Up challenge), since the program takes the input and splits it by itself.
The first trials of that challenge were failing, mainly caused by two reasons:

- I didn't get at first in which sense the blockchain compacted the input, so I thought that just a few bytes were selected and put in the block (yes... like *real* blockchain) and I was needed to figure out how
- lack of vision on building blocks of instructions

I returned to this challenge after having solved the Going Up, so constructing blocks was easier (or at least I was already warmed up). The online disassembler was used again to see the amount of bytes occupied by each instruction.

I started from the smallcode_x64 reference code at this link: https://www.exploit-db.com/exploits/46907, and from here I started to build blocks of instructions that do not mix up: similarly to Going Up I isolated single instructions and fill the remaining bytes with nop operations; afterwards I did some cleaning and removed the excessive duplications.

The major problem was caused by the long mov instruction that inserted into rdi the string "/bin/sh". Since I could not mix it up, I need to split it in some way. Two ideas came into my mind:

- i can push and pop into registers the two half into the stack
- i can mov the two half directly into the registers

I chose the second one but still the problem was that the shellcode takes one register as reference to perform the call, so I have to join two registers into one in some way. After some search online, a possible solution is to mov the halves into two 32 bit registers (and not into 64 ones) and then, with some shift instruction, fill the zeros of one register with the half contained by the other one.
With reference to this link: https://stackoverflow.com/questions/8580838/how-to-move-two-32-bit-registers-in-to-one-64-bit, the idea is to shift by 32 left the content of the register, that is one half of the string, while creating 32 zeros, and then shift right by filling the zeros with the other half.
As follows:

```
mov    ebx, 0x6e69622f
mov    eax, 0x68732f2f
shl    rbx, 32 →shift to left and leave zeros behind
0x6e69622f00000000
shrd   rbx, rax, 32 → shift to right filling with the other half
the zeros 0x68732f2f6e69622f
```

Taking the example above I thought about another way that could exploit the shifts as follows:
I perform the first mov to ebx, but I will put the /bin half, so 0x68732f2f; then I left shift the rbx, after that I mov into ebx again the missing half //sh, so 0x6e69622f. That last mov should go directly into the 32 bit zeroed part of the register, filling the rbx register with the value 0x68732f2f6e69622f.
Unfortunately I could not get it to work, so I'm missing something...

After getting to work the split of the big mov I continued the writing of the instruction of the provided shellcode without much more effort, adding a last nop operation before the syscall to avoid sending a new line manually when going into interactive mode before getting the flag. Finally by sending the shellcode the shell spawns correctly.

After spawning a shell in remote, I cat the flag.txt, obtaining:

```
$ cat flag.txt
BASC{gr8_w0rK_s0_far_tH3_l4sT_ch4ll_aWa1t5}
```