EXPLOITS

Carles Mateu

GREiA Research Group
Universitat de Lleida

Outline

Outline

Exploits

Exploits

Programs and tools that, take profit from a vulnerability (usually a programming error) to gain access, scalate privileges, etc. on a system.

Outline

Computer Memory (execution) - Basics

Endianness

Byte order when storing multibyte data in memory.

Little Endian

L1 L2 H1 H2

Big Endian

H1 H2 L1 L2

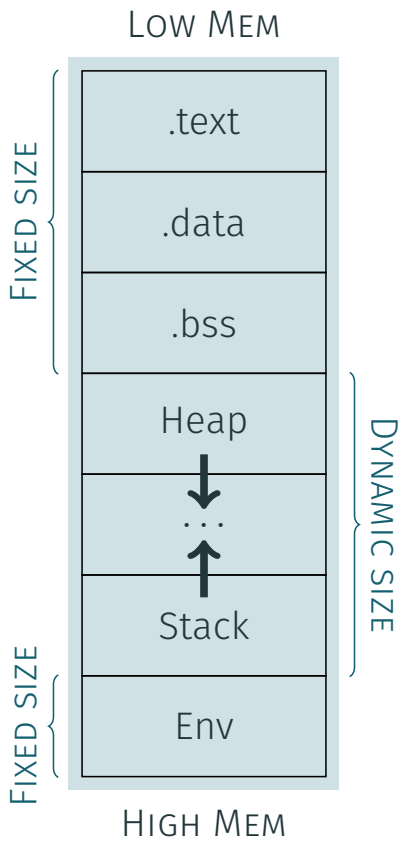
Examples

- x86/x64:Little Endian
- Motorola/ARM: Big Endian
- IP Networks: Big Endian

Computer Memory (execution) - Segments

- **.text:** Executable code. RO and Fixed Size.
- **.data:** Global initialized variables. Fixed Size.
- **.bss:** (below stack section). Global NON-initialized variables. Fixed Size.
- **Heap:** Dynamic allocated space. Grows from low -> high. (malloc, free).
- **Stack:** Dynamic. Grows from high -> low. Keeps calling stack and local variables.
- **Env:** System environment variables and program arguments.

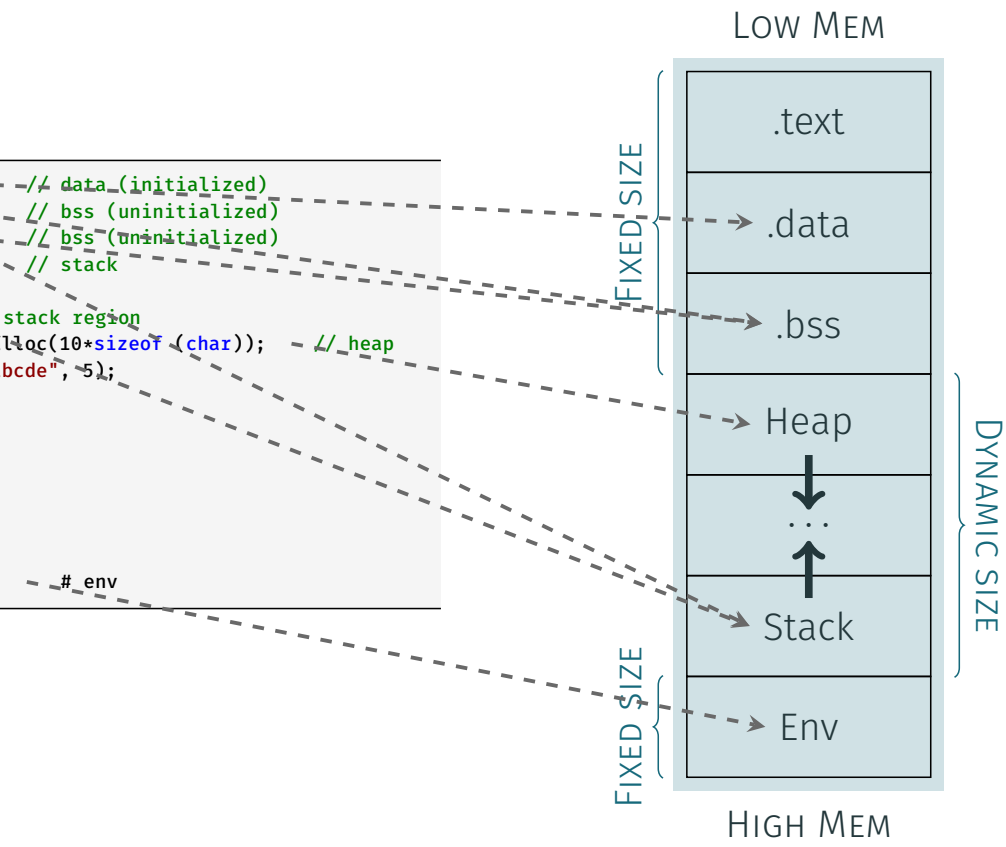
Computer Memory (execution) - Segment layout



Segment Layout Example

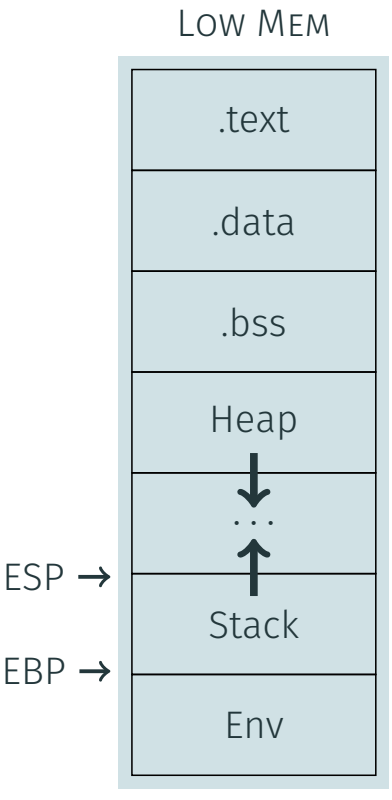
```
int index = 5;    // data (initialized)
char * str;       // bss (uninitialized)
int nothing;      // bss (uninitialized)
void fun(int c)   // stack
{
  int i=c;   // stack region
  str = (char*)malloc(10*sizeof(char));   // heap
  strncpy(str, "abcde", 5);
}
void main ()
{
  fun(1);
}


$ main argument1    # env
```



Stack Operation

- LIFO (FILO) Operation.
- Controlled by two registers: EBP, ESP.
- EBP points where the stack begins.
- ESP points where the next stacked value will be placed.



Outline

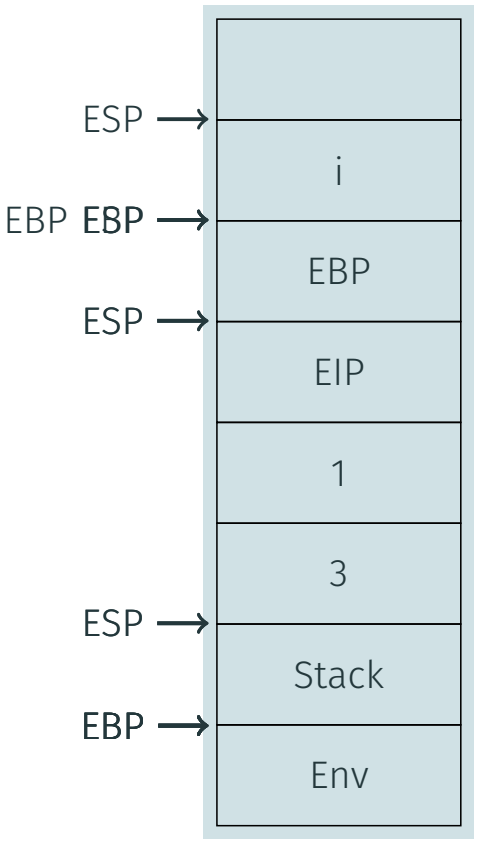
C Calling convention

- How a program keeps state and variables (local) when jumping to a function, and how it resumes back when returning (and how to return).
- Calling code:
  - Places parameters on stack.
  - Saves EIP (Instruction pointer) on stack.
  - Executes call (jumps to new instruction).
- Called code:
  - Saves EBP on stack.
  - ESP $\rightarrow$ EBP
  - ESP -= Local variables space (places local vars on stack)

C Calling convention

```
void fun(int c, int d)
{
int i;
....
}

void main ()
{
  Fun(1,3);
  printf("Hello World\n");
}
```



Outline

Buffer overflow

- Situation where an allocated buffer gets more data that it can handle.
- If the stack is corrupted (with buffer overflows), we end disrupting program function.

It's the **core dump** situations that happen sometimes when programming

Buffer overflow

Normal Operation

```
1  void test()
2  {
3    char buff[16];
4    strcpy(buff,"0123456789012345");
5    print("Hello world\n");
6  }
7
8  int main(int argc, char* argv[])
9  {
10   test();
11   return 0;
12 }
```

Buffer overflow

```
1  → void test()
2    {
3  →   char buff[16];
4  →   strcpy(buff,"012345678901234567890123");
5      print("Hello world\n");
6  → }
7
8    int main(int argc, char* argv[])
9    {
10 →   test();
11     return 0;
12   }
```

From buffer overflow to security breach....

How an overflow let's an attacker in:

If the data provoking the overflow can be sent externally: is a parameter, is received from network, from a file, etc. We can send data that will crash the program: it will try to execute code from a wrong address.

From buffer overflow to security breach....

How an overflow let's an attacker in:

That wrong address is one we sent to the attacker: can we send a "right" address pointing to somewhere we can send code? and can we send code?

From buffer overflow to security breach....

How an overflow let's an attacker in:

We can send code: we send data that lands in the stack (the overflow).

We can guess a right address for the new EIP: somewhere on the stack (don't have to be exact).

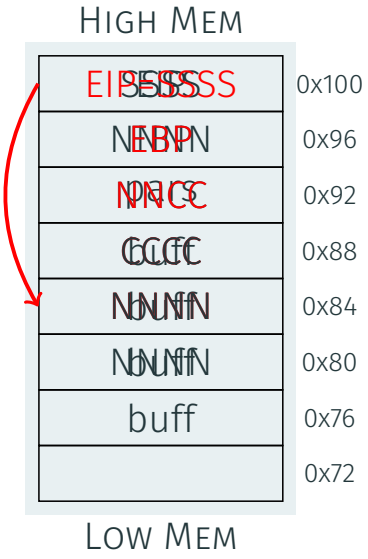
Buffer overflow exploit

Exploit:

```
1  void test(char *pars)
2  {
3    char buff[16];
4    strcpy(buff,pars);
5    print("Hello world\n");
6  }
7
8  int main(int argc, char*  argv[])
9  {
10   test(argv[1]);
11   return 0;
12 }
```

Set by user

C: code, N: NOP, SS: New EIP



What to inject?

To exploit this buffer overflow we will need:

- Something that will give us control of the machine once is run.
- The right address for the new EIP to overwrite the old EIP on the stack.

We will send some formatted data that will allow us to:

- “Guess” the right address for the EIP
- Take control of the system

Shellcode

This something (code) that gives us control is called a shellcode

Program code (assembly) that calls/executes useful stuff (traditionally creates a shell) designed to be injected as data and run from/on the stack segment.

Many available on the web:
`https://www.exploit-db.com/shellcodes`

Shellcode

They have been around for quite some time (1996), introduced by aleph1: Smashing the stack for fun and profit
`http://phrack.org/issues/49/14.html#article`

The name cames because the firsts ones did open a shell on the remote machine. That allowed:

- In case of local exploit: issue commands as exploited user on the attacked system.
- In remote exploits: most daemons worked either by inetd or by dup'ing to 0,1 their sockets, so a shell whould be usable remotely.

Shellcodes

They are, usually, written in assembly, and encoded to the minimum possible size.

They are getting quite complex: have to avoid ASLR, NX, etc,

ASLR Address space layout randomization randomizes location of Stack, Heap, .text, etc. on main memory.

NX Allows processors to mark as *Non Executable* parts of the memory (Intel: **XD**, AMD: **EVP**, ARM: **XN**).

Shellcode

A simple shell can be (`http://shell-storm.org/shellcode/files/shellcode-603.php`):

```
; execve("/bin/sh", ["/bin/sh"], NULL)

section .text
        global _start

_start:
        xor     rdx, rdx
        mov     qword rbx, '//bin/sh'
        shr     rbx, 0x8
        push    rbx
        mov     rdi, rsp
        push    rax
        push    rdi
        mov     rsi, rsp
        mov     al, 0x3b
        syscall
```


Shellcode

As you can see *easily* in the shellcode(😉), it simply calls the syscall: execve (0x3b), with the parameter: '/bin/sh'.

That is, substitutes the running process .text segment by that of /bin/sh (roughly).

NOP Sled

The other thing the exploit has to do, is find the *address* of the code, so it can push it into where the EIP is.

That need can be avoided by using a 'NOP Sled'.

NOP

NOP is an instruction that **does nothing**. Most languages and processors have instructions that are merelly placeholders (you know *pass* from python).

In x86/x64 assembly, NOP, is that instruction, assembled into 0x90.

NOP Sled

The trick is, then, to place, **before**, our shellcode a bunch of NOPs. Then, our new EIP will only have to be some address that lands **inside** the NOP Sled.

Processor then, will slide through all the NOPs until it reaches the shellcode. It won't matter if we miss our address by some bytes, as all them will be NOPs.

Safeguards

As one of the most dangerous vulnerabilities: they let attacker execute code on target system, some safeguards have been developed:

- **Hardware**:
  - **NX** Hardware to block executing code from certain memory areas(stack). Effective against code-in-stack exploits.
  - **ASLR** Address space randomization
- **Software**:
  - Compilers add code to produced binaries (for server software) to detect and prevent stack execution.
  - Better memory handling for software libraries.
  - Projects to detect (and correct) existing code vulnerabilities (`https://googleprojectzero.blogspot.com/p/about-project-zero.html`)

Safeguard avoidance

Much of the hardware/software safeguards have been avoided via newer techniques (jump-to-libc, etc.). So, the best safeguard, still is to program well:

```
1 void test(char *pars)
2 {
3   char buff[16];
4   strcpy(buff,pars);
5 }
```

```
1 void test(char *pars)
2 {
3   char buff[16];
4   strncpy(buff,pars,15);
5 }
```