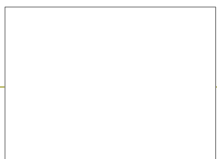


# Máster Universitario en Ingeniería Informática



## CyberSeguridad Tema 2: Ejecución remota

José Ismael Ripoll Ripoll



- § Introducción
- § Ataques de redirección de flujo
- § Pasos para la redirección
- § Buffer overflow: Stack buffer overflow
- § Ejecución en la propia pila
- § Return Oriented Programming (ROP)
- § Conclusiones ROP
- § Enlaces



- § En el tema anterior vimos la vulnerabilidades.
- § En este tema veremos cómo hacer uso de ellas.
  - ◆ Con ejemplos de explotación binaria.
  - ◆ Lo que se conoce como RCE (Remote Code Execution).
- § El objetivo es que seáis **capaces de valorar** las habilidades y conocimientos que disponen los atacantes.
- § No se espera que seáis capaces de explotar un fallo.
- § La explotación binaria es una de las áreas más complejas de la ciberseguridad.



- § En este tema se van a estudiar las estrategias que **utilizan los ATACANTES**.
- § Por tanto, hay que ser conscientes de que lo que aprendamos **NO** debemos emplearlo contra objetivos que no sean nuestros.
- § Seremos los malos.
- § La ignorancia es la madre de la felicidad.

*El arte de la guerra*

*"Si conoces a los demás y te conoces a ti mismo, ni en cien batallas correrás peligro..."*

Sun Tzu 400 a.C



§ Consisten en conseguir que un programa “normal” pase a ejecutar instrucciones de código útiles para el atacante:

- ◆ Bien inyectando código nuevo y saltando a él.
- ◆ Bien reutilizando código ya presente en el programa.

§ Se suele considerar que la redirección se produce a nivel de código máquina:

- ◆ Se redirige el Program Counter (PC) o Instruction Pointer (IP).

§ Curiosamente, el fallo SQL-injection es también una redirección de flujo (para ejecutar código inyectado).



- ◆ El SQL-injection lo veremos en el siguiente tema.



§ Primero se tiene que conseguir “controlar” el contador de programa (IP: Intruction Pointer):

◆ Esto es, asignar el valor deseado al IP.

§ Se puede tomar el control del IP de varias formas. Tantas como instrucciones máquina hay que cambian el flujo de un programa:

◆ Saltos indirectos:	<b>jmp</b> <b>*%eax</b>	} Se necesita Modificar registros 
◆ Llamadas indirectas:	<b>call</b> <b>* (%edx)</b>	
◆ Retorno de subrutina:	<b>ret</b>	} Se necesita Modificar la pila 
◆ Retorno de interrupción:	<b>iret</b>	



§ “Buffer overflow errors are characterized by the **overwriting of memory fragments** of the process, which should have **never** been modified intentionally or unintentionally.” OWASP

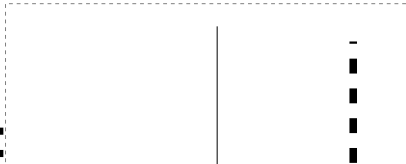
§ Los datos de los programas están en:

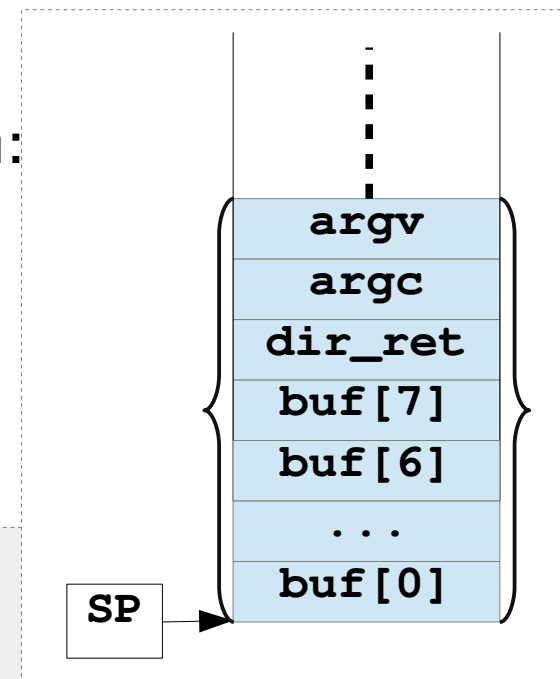
- ◆ Variables globales en el heap: **Heap-buffer-overflow**
- ◆ Variables locales en la pila (stack): **Stack-buffer- overflow**

§ El ataque más directo y sencillo es sobre-escribiendo la pila.





- § Se define el **stack-frame** como la zona de pila propia de cada función.
- § El stack-frame contiene, en este orden:
- ◆ Los parámetros recibidos por la función.
  - ◆ La dirección de retorno.
  - ◆ Las variables locales.
- 
- The diagram illustrates the structure of a stack frame. It shows a vertical stack of four light blue rectangular boxes. The top box is labeled 'argv', the second 'argc', the third 'dir\_re', and the bottom 'buf[7]'. A dashed vertical line extends upwards from the top of the 'argv' box. A large curly brace on the left side of the stack groups all four boxes together.



```
#include <stdio.h>

int main(int argc, char **argv) {
    char buf[8]; // buffer for eight characters
    gets(buf);    // read from stdio (sensitive function!)
    printf("%s\n", buf); // print out data stored in buf
    return 0;     // 0 as return value
}
```

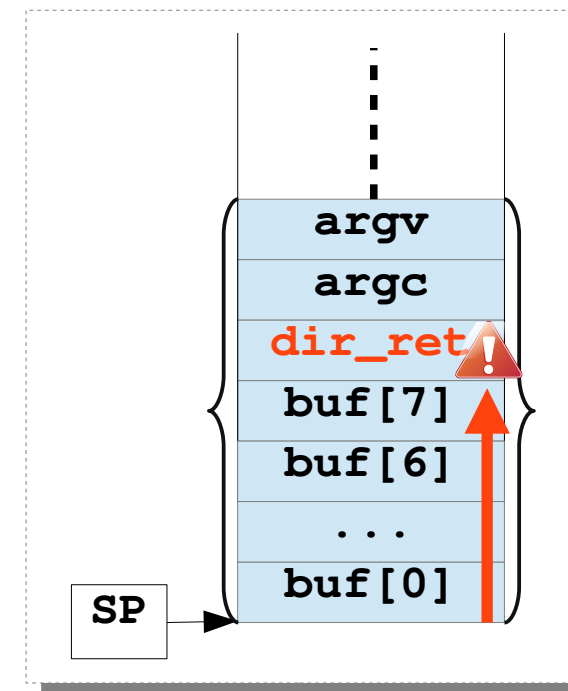




## § ¿Qué pasa si nos salimos del rango del vector “buf”?

- ◆ Que “machacamos” la dirección de retorno que hay en la pila.

§ Entonces, cuando la función en curso trate de “retornar” para continuar por donde se quedó la que la había llamado...



GOTO ¿?



§ Típicamente, sobrescribir la dirección de retorno acaba mal: SEGMENT FAULT.

§ Puesto que se salta a una dirección que:



- ◆ La página destino **no tiene permisos** de ejecución (NX)

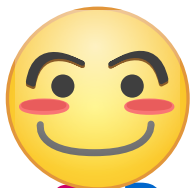
- Se produce una excepción del procesador y el SO mata el proceso SIGSEGV.

- ◆ La instrucción es un **opcode inválido**.

- Se produce una excepción de operación inválida y.....

- ◆ La instrucción es válida pero **hace cosas extrañas**.

- Podría saltarse a ejecutar una función que no estaba prevista.



§ Pero también es posible saltar a sitios interesantes!!!



¿Qué parámetros hay que pasar a este programa para ejecutar la función hacked?

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

char *cmd[]={"/bin/sh","-i",0x0};

void hacked(){
    printf("+-----+\n");
    printf("+      Premio      +\n");
    printf("+-----+\n");
    execve(cmd[0], cmd, 0x0);
    exit(0);
}

void dump_pila( unsigned long *base,
                int size){
    int i;
    for (i=0; i< size; i++){
        printf("%p: 0x%016lx\n", base, *base);
        base ++;
    }
}
```

```
void vuln(unsigned long address, int reps ){
    int i;
    unsigned long ptr[10];

    for (i=0; i<10; i++) { ptr[i]=i; }

    for (i=0; i<reps; i++) { ptr[i]= address; }
    dump_pila(ptr, 15);

    return;
}

main(int argc, char *argv[]){
    int reps;
    unsigned long dir;

    dir=strtoul(argv[1], NULL, 16);
    reps=atoi(argv[2]);

    printf("Empiza el juego!!! \n");
    vuln( dir, reps);
    printf("Termina el juego!!! \n");
}
```



## § Antes de la introducción de la técnica NX/DEP

- ◆ NX: Non eXecutable.
- ◆ DEP: Data Execution Prevention.

## § Era posible seguir sobre-escribiendo la pila con las instrucciones que el atacante quería, y luego saltar a ese código inyectado:

- ◆ Inyección de código + ejecución.

## § AMD introdujo en los primeros procesadores de 64 bits (Opteron) con un bit nuevo en la tabla de páginas que permite **prohibir la ejecución** (fetch) en esas páginas.



## § La mayoría de sistemas incluyen el NX, por lo que **ya nadie** fabrica *exploits* inyectando código.

Es muy educativo fabricar exploits que ejecuten código inyectado



## § ROP es una forma de escribir programas que:

- ◆ NO es programación convencional.
- ◆ NO consiste en emitir instrucciones (tal como hacen los compiladores).
- ◆ NO necesita permisos de ejecución para controlar el contador de programa.
- ◆ Es necesario **saber programar en ensamblador!**.
- ◆ Es necesario conocer el **ABI del proceso target!**
- ◆ Existen “generadores de programas” ROP automáticos.
  - Aunque suele tener una componente artesanal.



§ Un ROP se puede entender como una forma de “**reorganizar**” las instrucciones máquina presentes en un proceso para que hagan lo que el atacante quiere.



- § ROP se vale de las instrucciones de retorno (ret) para cargar el contador de proceso.
- § Si el atacante es capaz de manipular el contenido de la pila, entonces puede:
  - ◆ Sobre-escribir la dirección de retorno del proceso.
  - ◆ Y una vez se ha tomado el control del contador de programa, la idea es seguir dirigiendo el flujo para ejecutar instrucciones lo que quiere el atacante.

**Para entender ROP  
es necesario conocer  
cómo los procesos utilizan la pila**





## § Cómo construir un programa ROP:

- 1) Se construye el programa en ensamblador que queremos ejecutar.
  - Este se utilizará como base o modelo del programa.
- 2) Se buscan los *gadgets* para “componer nuestro programa”.
- 3) Se construye el contenido de la pila necesario para cada uno de los *gadgets*.
- 4) Se secuencian todos los *gadgets* y se añade el padding necesario para forzar el desbordamiento hasta justo la dirección de retorno, que será el inicio de nuestro programa ROP.
- 5) Se envía el “programa ROP” al proceso target y a esperar el premio.





## § Se construye el programa en ensamblador que queremos ejecutar.

- ◆ Se puede escribir en "C" y luego solo compilarlo, pero no ensamblarlo. Ya tenemos el código ensamblador que buscamos.

```
$ gcc -S -static exec-shell.c
$ cat exec-shell.s
```

Google: "Linux syscall convention registers"

- ◆ También podemos generar el ELF ejecutable y desensamblarlo:

```
$ gcc -static exec-shell.c -o exec-shell
$ objdump -d exec-shell
```

```
000000000040105e <main>:
40105e: 55          push    %rbp
40105f: 48 89 e5    mov     %rsp,%rbp
401062: 48 8b 05 07 e0 2b 00    mov     0x2be007(%rip),%rax # 6bf070 <cmd>
401069: ba 00 00 00 00    mov     $0x0,%edx
40106e: be 70 f0 6b 00    mov     $0x6bf070,%esi
401073: 48 89 c7    mov     %rax,%rdi
401076: e8 75 24 03 00    callq   4334f0 <__execve>
40107b: 5d          pop     %rbp
40107c: c3          retq

.....
00000000004334f0 <__execve>:
4334f0: b8 3b 00 00 00    mov     $0x3b,%eax
4334f5: 0f 05          syscall
.....
```

%rdi,  
%rsi,  
%rdx,  
%r10,  
%r8 and  
%r9

/usr/include/  
x86\_64-linux-gnu/  
asm/unistd\_64.h

Más



## § Buscar gadgets

- ◆ “Son fragmentos de código del proceso target que terminan con una instrucción de retorno”
- ◆ Puesto que acaban con un “ret”, es posible decidir a donde se “retorna” poniendo la dirección destino en la pila.
- ◆ Desde cada *gadget* se puede saltar a otro gadget que queramos.
- ◆ Podemos “enganchar” los *gadgets* como queramos.
- ◆ Por tanto, podemos secuenciar las instrucciones de los gadgets a voluntad.

§ Un compilador normal, genera una secuencia de código.

§ Nosotros vamos a **resecuenciar el código** que generó el compilador.



## § Buscar gadgets

- ◆ Si nuestro target fuera el proceso “cat” podríamos ver los gadget inspeccionando el fichero ejecutable (/bin/cat):

```
$ objdump -d /bin/cat | egrep -B5 "[^ ]c3 "
```

40159c:	48 8b 05 55 9a 20 00	mov	0x209a55(%rip),%rax # 60aff8 <__sprintf_chk@plt+0xxx>
4015a3:	48 85 c0	test	%rax,%rax
4015a6:	74 05	je	4015ad <__uflow@plt-0x23>
4015a8:	e8 a3 02 00 00	callq	401850 <__gmon_start__@plt>
4015ad:	48 83 c4 08	add	\$0x8,%rsp
4015b1:	c3	<b>retq</b>	
--			
402636:	48 2d 18 b3 60 00	sub	\$0x60b318,%rax
40263c:	48 83 f8 0e	cmp	\$0xe,%rax
402640:	48 89 e5	mov	%rsp,%rbp
402643:	77 02	ja	402647 <__sprintf_chk@plt+0xc17>
402645:	5d	pop	%rbp
402646:	c3	<b>retq</b>	
--			
402bc8:	4c 89 e0	mov	%r12,%rax
402bcb:	5b	pop	%rbx
402bcc:	5d	pop	%rbp
402bcd:	41 5c	pop	%r12
402bcf:	41 5d	pop	%r13
402bd1:	c3	<b>retq</b>	



## § Gadgets

- ◆ Buscar gadgets “a mano” es complejo.
- ◆ Existen aplicaciones para buscar gadgets según la semántica deseada.
- ◆ Se ha demostrado que la potencia expresiva de los gadgets que se encuentran en la mayoría de ejecutables y librerías es suficiente como para poder construir cualquier programa.
  - Vaya, que se cuenta con un juego de instrucciones completo!



§ La forma de ordenar los gadgets es “poniendo las direcciones de cada gadget en la pila”



```

1215c5: 66 66 2e 0f 1f 84 00    data32 nopw %cs:0x0(%rax,%rax,1)
1215cc: 00 00 00 00
1215d0: b8 02 00 00 00         mov     $0x2,%eax
1215d5: c3                      retq

121d0f: 48 89 72 20             mov     %rsi,0x20(%rdx)
121d13: c3                      retq

11f9fb: 41 5e                   pop     %r14
11f9fd: 41 5f                   pop     %r15
11f9ff: 5d                      pop     %rbp
11fa00: c3                      retq

ed243: 3d 00 f0 ff ff         cmp     $0xffffffff00,%eax
ed248: 0f 47 d1               cmova   %ecx,%edx
ed24b: 89 d0                  mov     %edx,%eax
ed24d: c3                      retq

36e3e: b9 0d 00 00 00         mov     $0xd,%ecx
36e43: 48 89 c6               mov     %rax,%rsi
36e46: 89 c8                  mov     %ecx,%eax
36e48: 0f 05                  syscall

36f25: 41 89 90 88 00 00 00    mov     %edx,0x88(%r8)
36f2c: 48 8b 54 24 38          mov     0x38(%rsp),%rdx
36f31: 49 89 90 90 00 00 00    mov     %rdx,0x90(%r8)
36f38: 48 81 c4 d0 00 00 00    add     $0xd0,%rsp
36f3f: c3                      retq

0000000000037690 <_sigdelset>:
37690: 8d 4e ff               lea     -0x1(%rsi),%ecx
37693: 8b 01 00 00 00         mov     $0x1,%eax
37698: 48 63 d1               movslq  %ecx,%rdx
3769b: 48 d3 e0               shl     %cl,%rax
3769e: 48 c1 ea 06            shr     $0x6,%rdx
376a2: 48 f7 d0               not     %rax
376a5: 48 21 04 d7            and     %rax,(%rdi,%rdx,8)
376a9: 31 c0                  xor     %eax,%eax
376ab: c3                      retq

000000000003c920 <abs>:
3c920: 89 fa                  mov     %edi,%edx
3c922: 89 f8                  mov     %edi,%eax
3c924: c1 fa 1f               sar     $0x1f,%edx
3c927: 31 d0                  xor     %edx,%eax
3c929: 29 d0                  sub     %edx,%eax
3c92b: c3                      retq

000000000003c930 <labs>:
3c930: 48 89 fa               mov     %rdi,%rdx
3c933: 48 89 f8               mov     %rdi,%rax
3c936: 48 c1 fa 3f            sar     $0x3f,%rdx
3c93a: 48 31 d0               xor     %rdx,%rax
3c93d: 48 29 d0               sub     %rdx,%rax
3c940: c3                      retq

0000000000048880 <mbstowcs>:
48880: 48 83 ec 28            sub     $0x28,%rsp
48884: 48 89 74 24 08         mov     %rsi,0x8(%rsp)
48889: 48 8d 4c 24 10         lea     0x10(%rsp),%rcx
4888e: 48 8d 74 24 08         lea     0x8(%rsp),%rsi
48893: 48 c7 44 24 10 00 00    movq    $0x0,0x10(%rsp)
4889a: 00 00
4889c: e8 ff b8 05 00         callq   a41a0 <mbsrtowcs>
488a1: 48 83 c4 28            add     $0x28,%rsp
488a5: c3                      retq

489be: 5b                      pop     %rbx
489bf: 8b 40 58               mov     0x58(%rax),%eax
489c2: c3                      retq

```

3

2

5

4

1

0x036e3e

0x-----

0x03c920

0x-----

0x-----

0x-----

0x1215d0

0x-----

0x11f9fb

0x-----

0x-----

0x048889

0x.....

0x.....

0x.....

0x.....

g5

g4

g3

g2

Ret\_addr (g1)

Overflow



- § Si somos capaces de introducir en la pila de proceso target la secuencia de bytes que acabamos de construir, entonces cuando el proceso target retorne...
- § Se ejecutaría nuestro programa ROP.
- § Un desbordamiento de pila es PERFECTO para nuestros intereses...
- § EXACTAMENTE nos permite escribir en la pila TODO lo que queremos.





§ Si el atacante **puede redirigir el flujo** del programa...

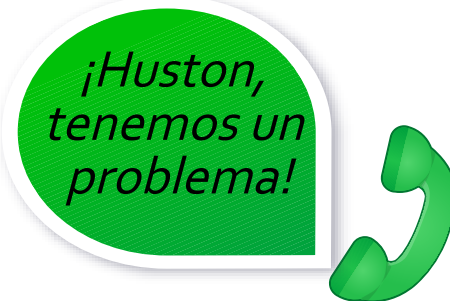
§ ¿Pero cuán malo es la ejecución remota?

§ Si el proceso atacado está escuchando en un puerto TCP:

◆ Lo cual es lo más típico.

§ Entonces el atacante puede (y lo hace SIEMPRE) reutilizar la misma conexión que está abierta para comunicarse con el *target*.

◆ Por lo que aunque el target tenga una dirección privada (192.168...), y no sea visible desde el exterior, **el atacante puede acceder!**





- § [https://www.owasp.org/index.php/Buffer\\_overflow\\_attack](https://www.owasp.org/index.php/Buffer_overflow_attack)
- § <http://cybersecurity.upv.es/downloads/frop.pdf>
- § [https://www.trailofbits.com/resources/practical\\_rop\\_slides.pdf](https://www.trailofbits.com/resources/practical_rop_slides.pdf)

