CI-0118 Lenguaje Ensamblador, g01
Tarea programada #3
I Semestre/2021

Motivación

Esta tarea pretende mostrarle distintas vulnerabilidades que pueden ser explotadas cuando los programas no se protegen apropiadamente contra problemas como el del “buffer overflow”.

Al finalizar, tendrá una mejor noción de cómo escribir programas más seguros, además de conocer cómo los compiladores y los sistemas operativos pueden hacer menos vulnerables a los programas.

Adicionalmente, esta tarea le permitirá un profundo entendimiento del uso de la pila (incluyendo paso de parámetros) y del cómo las instrucciones en lenguaje ensamblador están codificadas. Todo esto en arquitecturas x86-64 bits. También tendrá la oportunidad de ganar más experiencia usando depuradores para Linux como los que ya utilizó en la primera tarea (la de la bomba).

Esta tarea involucra cinco ataques en total a dos programas que tienen distintas deficiencias de seguridad. El propósito de estos ataques es meramente académico y busca ayudar a entender mejor la importancia del lenguaje ensamblador en ciertos tipos de problemas. El conocimiento aquí obtenido no debe ser usado en accesos no autorizados a equipos de terceros en ninguna circunstancia.

Esta tarea está basada en el material del libro Computer Systems: A programmer's perspective. Se recomienda leer las secciones **3.10.3 y 3.10.4** antes de comenzar a resolver esta tarea.

Paso 1: Obtener los archivos

Este proyecto es individual. Al igual que en la tarea primera tarea, necesita obtener los programas que se usarán del siguiente servidor:

http://100.25.143.128:15513

Cada conjunto de programas será distinto para cada estudiante. El servidor construirá y devolverá un archivo llamado `targetk.tar`, tal que k es el número único asociado a sus programas asignados.

Para comenzar a trabajar, extraiga los archivos contenidos de targetk.tar (con el comando `tar -xvf targetk.tar`) en una carpeta en su Linux favorito. Sólo debe descargar un conjunto de programas. Si por alguna razón necesita descargar otro, deberá comenzar a trabajar sobre esos programas de nuevo y desechar los anteriores.

Los archivos que incluidos en `targetk.tar` son:

- `README.txt`: es una descripción de los contenidos de la carpeta.
- `ctarget`: un programa ejecutable vulnerable a ataques de inyección de código (CI)
- `rtarget`: un programa ejecutable vulnerable a ataques de programación orientada al retorno (ROP, por sus siglas en inglés).
- `cookie.txt`: un código hexadecimal de 8 dígitos que deberá usar como identificador único en todos sus ataques.
- `farm.c`: el código fuente de los gadgets que serán usados por los ataques ROP. (Nota: los gadgets son pequeñas secuencias de instrucciones que terminan en un ret que se invocarán eventualmente durante alguno de los ataques).
- `hex2raw`: un utilitario que le servirá para generar cadenas de ataques.

Estos archivos deben ser copiados en un directorio local protegido y todos los programas se ejecutarán únicamente en esa misma carpeta.

Observaciones importantes (probablemente tengan más sentido cuando esté más adelante en la solución de la tarea, por ahora sólo téngase en cuenta lo siguiente):

- Esta tarea deberá realizarse en un ambiente similar al ambiente en el que fueron generados los archivos target (Ubuntu 16, ver instrucciones de instalación por aparte).
- Sus soluciones no pueden usar ataques para eludir el código de validación en los programas. Específicamente, cualquier dirección que incorpore en una cadena de ataque para que la use una instrucción ret debe estar en uno de los siguientes destinos:
  - Las direcciones de las funciones `touch1, touch2` o `touch3.`
  - Las direcciones de su código inyectado.
  - Las direcciones de uno de los gadgets de la granja de gadgets.
- Solo debe construir gadgets desde el archivo `rtarget` con direcciones en el rango entre las funciones `start_farm` y `end_farm.`

Paso 2: entender los programas objetivo (target)

Los programas `ctarget` y `rtarget` usan la entrada estándar para capturar datos, específicamente usan la función getbuf. La misma se define de la siguiente forma:

```
1 unsigned getbuf()
2 {
3     char buf[BUFFER_SIZE];
4     Gets(buf);
5     return 1;
6 }
```

La función `Gets` es similar a la función de la biblioteca estándar `gets` (que lee una cadena de la entrada estándar terminada en `\n` o `eof)` y lo almacena junto con un caracter null al final,

en el destino especificado. En este código el destino específico es el `array buf`, el cual fue declarado con `BUFFER_SIZE` bytes. El tamaño de `BUFFER_SIZE` es una constante que depende de la versión específica de los programas personalizados que le fueron asignados.

Las funciones `Gets()` y `gets()` no tienen forma de determinar si sus buffers destinos son lo suficientemente grandes para almacenar la cadena de caracteres que están leyendo. Estas funciones solamente copian secuencias de bytes en su destino, probablemente pasándose de los límites de sus variables destino.

Si la cadena escrita por el usuario, y luego leída por `getbuf` es suficientemente corta, entonces `getbuf` devolverá un 1. El siguiente es un ejemplo de cómo se vería ejecutando `ctarget`:

```
unix> ./ctarget
Cookie: 0x1a7dd803
Type string: Keep it short!
No exploit. Getbuf returned 0x1 Normal return
```

El error ocurre cuando la cadena no cabe en `buf`, como en el siguiente ejemplo:

```
unix> ./ctarget
Cookie: 0x1a7dd803
Type string: This is not a very interesting string, but it has the
property ...
Ouch!: You caused a segmentation fault!
Better luck next time
```

El programa `rtarget` tendrá exactamente el mismo comportamiento. Como el mensaje de error lo indica, hubo desborde del buffer que corrompió el proceso y lo llevó a un error memoria (segmentation fault).

Su misión en esta tarea es escribir cadenas apropiadas para `ctarget` y `rtarget` que hagan cosas interesantes. Estas cadenas son las que se aprovechan de las vulnerabilidades para hacer sus fechorías. De ahora en adelante llamaremos a esas cadenas “exploit strings”.

Los archivos ctarget y rtarget tienen los siguientes argumentos que puede usarse al invocarse en la línea de comandos:
`-h:` ayuda que incluye la lista de argumentos.
`-q:` no envía los resultados al servidor (para su debida calificación).
`-i` archivo: donde archivo es un archivo de texto con la entrada, esto para evitar escribir datos directamente en la entrada estándar.

Dado que los exploit strings que usará no necesariamente corresponden a valores imprimibles de la tabla ASCII, entonces se usará el programa `hex2raw` para generar las cadenas crudas (los valores ASCII como tal). En el Anexo A de este documento hay más información sobre `hex2raw`.

Observaciones importantes:

- Sus exploit strings no podrán contener el valor `0x0A` en posiciones intermedias de la cadena. Recuerde que `0x0A` corresponde al cambio de línea `(\n)`. Cuando `Gets` encuentre este valor, supondrá que la cadena de entrada (exploit string) ya terminó.
- `hex2raw` espera valores hexadecimales de dos dígitos separados por uno o más espacios en blanco. Si quiere generar un byte con el valor decimal de 0, deberá escribir `00`. Si quiere crear la palabra `0xdeadbeef`, entonces deberá enviar a `hex2raw` "ef be ad de" (orden en little endian).

Cuando haya resuelto satisfactoriamente uno de los niveles, el programa le enviará automáticamente una notificación al servidor para asignar la calificación. Por ejemplo, se verá algo así:

```
unix> ./hex2raw < ctarget.l2.txt | ./ctarget
Cookie: 0x1a7dd803
Type string:Touch2!: You called touch2(0x1a7dd803)
Valid solution for level 2 with target ctarget
PASSED: Sent exploit string to server to be validated.
NICE JOB!
```

Al igual que en la tarea de la bomba, el servidor recibirá su exploit string y si funciona, entonces actualizará la tabla de calificaciones del servidor. Dicha tabla podrá verse en la dirección:

http://100.25.143.128:15513/scoreboard

En esta tarea no se penalizarán los intentos fallidos. Por tanto, siéntase en libertad de enviar a `ctarget` y a `rtarget` todos los exploit string que desee. **Tiene tiempo hasta el viernes 23 de julio a las 5.00pm para resolver todas las fases.**

La siguiente tabla resume las fases de esta tarea. Como puede ver, las primeras tres involucran ataques de inyección de código (CI) en ctarget, mientras que las últimas dos involucran ataques de programación orientada a retorno (ROP) en rtarget.

Fase	Programa	Nivel	Ataque	Función	Puntos
1	ctarget	1	CI	touch1	10
2	ctarget	2	CI	touch2	25
3	ctarget	3	CI	touch3	25
4	rtarget	2	ROP	touch2	35
5	rtarget	3	ROP	touch3	5

Paso 3: hacer los ataques CI

En las primeras 3 fases sus exploit strings deberán atacar `ctarget`. Este programa está configurado de manera que las posiciones de la pila sean consistentes de una ejecución a la siguiente y de modo que los datos en la pila puedan tratarse como código ejecutable. Estas características hacen que el programa sea vulnerable a ataques donde los exploit strings contienen las codificaciones de bytes del código ejecutable.

CI Nivel 1: Para la fase 1, no se inyectará código nuevo. En su lugar, su exploit string redirigirá el programa para ejecutar un procedimiento existente.

La función getbuf se llama desde ctarget por la función `test` que tiene el siguiente código en C:

```
1 void test()
2 {
3     int val;
4     val = getbuf();
5     printf("No exploit. Getbuf returned 0x%x\n", val);
6 }
```

Cuando `getbuf` ejecuta su retorno (línea 5 de `getbuf`), el programa normalmente reanudaría la ejecución dentro de la función `test` (en la línea 5 de esta función). Se desea cambiar este comportamiento. Dentro del archivo `ctarget`, está el código de la función `touch1`, la cual tiene la siguiente representación en C:

```
1 void touch1()
2 {
3     vlevel = 1; /* Part of validation protocol */
4     printf("Touch1!: You called touch1()\n");
5     validate(1);
6     exit(0);
7 }
```

La tarea es hacer que `ctarget` ejecute el código de `touch1` cuando `getbuf` ejecuta su instrucción de retorno, en lugar de regresar a `test`. Nótese que su exploit string va a corromper partes de la pila que no están relacionadas con esta etapa, pero esto no causará problemas, pues `touch1` hace que el programa finalice directamente.

Consejos:

- Toda la información que se necesita para crear el exploit string para este nivel se puede determinar examinando una versión desensamblada de `ctarget`. Use `objdump -d` para obtener la versión desensamblada.
- La idea es colocar una representación de bytes de la dirección de inicio para `touch1` de manera que la instrucción `ret` al final del código de `getbuf` transfiera el control a `touch1`.
- Tenga cuidado con el ordenamiento de los bytes.
- Se podría usar `gdb` para recorrer el programa hasta las últimas instrucciones de `getbuf` para asegurarse de que está trabajando correctamente.
- La posición de `buf` en el stack frame para getbuf depende de el valor en tiempo de compilación de la constante `BUFFER_SIZE`, así como de la estrategia de asignación que utiliza `gcc`. Se necesita examinar el código desensamblado para determinar esta posición.

CI nivel 2: La fase 2 involucra inyectar una pequeña cantidad de código como parte de su exploit string.

Dentro del archivo `ctarget` hay código para una función `touch2` que tiene la siguiente representación en C:

```
1 void touch2(unsigned val)
2 {
3     vlevel = 2; /* Part of validation protocol */
4     if (val == cookie) {
5           printf("Touch2!: You called touch2(0x%.8x)\n", val);
6           validate(2);
7     } else {
8           printf("Misfire: You called touch2(0x%.8x)\n", val);
9           fail(2);
10    }
11    exit(0);
12 }
```

El objetivo consiste en hacer que `ctarget` ejecute el código de `touch2` en lugar de retornar a `test`. En este caso, sin embargo, se debe aparentar a `touch2` que se pasó su `cookie` como su argumento.

Consejos:

- Se desea posicionar una representación en bytes de la dirección de su código inyectado de tal manera que la instrucción ret al final del código de getbuf transfiera el control a la misma.
- Hay que recordar que el primer argumento a una función (en x86-64) se pasa a través del registro `rdi`.
- El código inyectado debe cambiar el registro a su `cookie`, y luego utilizar la instrucción `ret` para transferir el control a la primera instrucción de `touch2`.
- No intente utilizar `jmp` o `call` en su exploit. La codificación de las direcciones de destino de estas instrucciones es complicada de formular. Use instrucciones `ret` para transferir el control, incluso cuando no esté retornando de un llamado a función.
- Revise la discusión en el Apéndice B sobre cómo utilizar las herramientas para generar la representación a nivel de byte de las secuencias de instrucciones.

CI Nivel 3: La fase 3 también involucra un ataque de inyección de código, pero pasando una hilera como argumento.

Dentro del archivo `ctarget` hay código para las funciones `hexmatch` y `touch3`, las cuales tienen las siguientes representaciones en c:

```
1 /* Compare string to hex represention of unsigned value */
2 int hexmatch(unsigned val, char *sval)
3 {
4     char cbuf[110];
5     /* Make position of check string unpredictable */
6     char *s = cbuf + random() % 100;
7     sprintf(s, "%.8x", val);
8     return strncmp(sval, s, 9) == 0;
9 }
10
11 void touch3(char *sval)
12 {
13    vlevel = 3; /* Part of validation protocol */
14    if (hexmatch(cookie, sval)) {
15          printf("Touch3!: You called touch3(\"%s\")\n", sval);
16          validate(3);
17    } else {
18          printf("Misfire: You called touch3(\"%s\")\n", sval);
19          fail(3);
20    }
21    exit(0);
22 }
```

La tarea consiste en hacer que `ctarget` ejecute el código de `touch3` en lugar de retornar a `test`. Debe parecer a `touch3` que se le pasó una representación en hilera de su `cookie` como argumento.

Consejos:

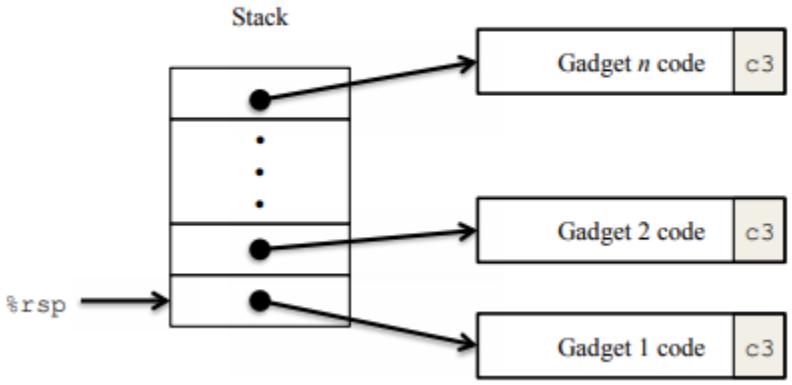
- Se necesita incluir una representación en hilera de su `cookie` en el exploit string. La hilera debe consistir en ocho dígitos hexadecimales (ordenados de más a menos significativo) sin el "`0x`" inicial.
- Hay que recordar que una hilera se representa en C como una secuencia de bytes seguidos por un byte con valor cero. Digite "`man ascii`" en cualquier línea de comandos de Linux para ver las representaciones en bytes de los caracteres que necesita.
- Su código inyectado debe modificar el registro `rdi` a la dirección de esta hilera.
- Cuando las funciones `hexmatch` y `strncmp` se llaman, ellas empujan datos dentro de la pila, sobreescribiendo porciones de memoria que contenían el buffer que utiliza `getbuf`. Como resultado, se debe tener cuidado de dónde colocar la representación de hilera del `cookie`.

Paso 4: hacer los ataques ROP

Realizar ataques de inyección de código en el programa rtarget es mucho más difícil que ctarget, pues usa dos técnicas para impedir dichos ataques:

- Utiliza aleatorización para que las posiciones de la pila difieran de una ejecución a otra. Esto hace imposible determinar en dónde se ubicará el código inyectado.
- Marca la sección de la memoria que almacena la pila como no ejecutable, de manera que si se pudiera ajustar el contador del programa al inicio del código inyectado, el programa fallaría con una falta de segmentación (segmentation fault).

Afortunadamente, personas ingeniosas han desarrollado estrategias para hacer cosas útiles en un programa ejecutando código existente, en lugar de inyectar nuevo código. La forma más general para esto se denomina programación orientada a retorno (ROP). La estrategia de ROP es identificar las secuencias de bytes en un programa existente que consisten en una o más instrucciones seguidas de la instrucción `ret`. Tal segmento se conoce como un gadget.



La figura anterior ilustra cómo la pila puede configurarse para ejecutar una secuencia de n gadgets. En esta figura, la pila contiene una secuencia de direcciones a gadgets. Cada gadget consiste en una serie de bytes de instrucción, y el del final es `0xc3`, que codifica la instrucción ret. Cuando el programa ejecuta una instrucción ret comenzando con esta configuración, iniciará una cadena de ejecuciones de gadgets, y la instrucción ret al final de cada gadget causa que el programa salte al principio del siguiente.

Un gadget puede utilizar el código correspondiente a instrucciones de lenguaje ensamblador generadas por el compilador, especialmente al final de funciones. En la práctica, hay algunos gadgets útiles con esta forma, pero no suficientes para implementar funciones importantes. Por ejemplo, es poco probable que una función compilada tenga `popq %rdi` como la última instrucción antes de ret. Afortunadamente, con un conjunto de instrucciones orientadas a

bytes, como el x86-64, un gadget puede encontrarse extrayendo patrones de otras partes de la secuencia de bytes de instrucciones.

Por ejemplo, una versión de `rtarget` contiene el código generado por la siguiente función de C:

```
void setval_210(unsigned *p)
{
      *p = 3347663060U;
}
```

La posibilidad de que esta función sea útil para atacar un sistema parece baja. Pero, el código desensamblado de esta función muestra una secuencia de bytes interesante:

```
0000000000400f15 <setval_210>:
400f15: c7 07 d4 48 89 c7  movl $0xc78948d4,(%rdi)
400f1b: c3                 retq
```

La secuencia de bytes `48 89 c7` codifica la instrucción `movq %rax, %rdi`. La secuencia es sucedida por el valor byte `c3`, que codifica la instrucción `ret`. La función comienza en la dirección `0x400f18`, que copia el valor de 64 bits en el registro `rax` al registro `rdi`.

Su código para `rtarget` contiene un número de funciones similares a `setval_210`, que se mostró arriba, en una región que llamaremos la granja de gadgets. Su trabajo consiste en identificar gadgets útiles en la granja de gadgets y usarlos para realizar ataques similares a los que realizó en las fases 2 y 3.

Importante: La granja de gadgets está demarcada por las funciones `start_farm` y `end_farm` en su copia de `rtarget`. No intente construir gadgets con otras porciones del código del programa.

ROP Nivel 2:

Para la fase 4, se repetirá el ataque de la fase 2, pero con el programa `rtarget` y usando gadgets de la granja de gadgets. Se puede construir la solución usando gadgets de los siguientes tipos, y usando sólo los primeros 8 registros de x86-64 (`rax-rdi`).

`movq`: ver siguiente figura.
`popq`: ver siguiente figura.
`ret`: se codifica con el byte `0xc3`.
`nop` : “No operación”, se codifica con el byte `0x90`. Su único efecto es que el contador de programa incrementa en 1.

A. Encodings of `movq` instructions

`movq` *S*, *D*

Source *S*	Destination *D*							
	%rax	%rcx	%rdx	%rbx	%rsp	%rbp	%rsi	%rdi
%rax	48 89 c0	48 89 c1	48 89 c2	48 89 c3	48 89 c4	48 89 c5	48 89 c6	48 89 c7
%rcx	48 89 c8	48 89 c9	48 89 ca	48 89 cb	48 89 cc	48 89 cd	48 89 ce	48 89 cf
%rdx	48 89 d0	48 89 d1	48 89 d2	48 89 d3	48 89 d4	48 89 d5	48 89 d6	48 89 d7
%rbx	48 89 d8	48 89 d9	48 89 da	48 89 db	48 89 dc	48 89 dd	48 89 de	48 89 df
%rsp	48 89 e0	48 89 e1	48 89 e2	48 89 e3	48 89 e4	48 89 e5	48 89 e6	48 89 e7
%rbp	48 89 e8	48 89 e9	48 89 ea	48 89 eb	48 89 ec	48 89 ed	48 89 ee	48 89 ef
%rsi	48 89 f0	48 89 f1	48 89 f2	48 89 f3	48 89 f4	48 89 f5	48 89 f6	48 89 f7
%rdi	48 89 f8	48 89 f9	48 89 fa	48 89 fb	48 89 fc	48 89 fd	48 89 fe	48 89 ff

B. Encodings of `popq` instructions

Operation	Register *R*							
	%rax	%rcx	%rdx	%rbx	%rsp	%rbp	%rsi	%rdi
popq *R*	58	59	5a	5b	5c	5d	5e	5f

C. Encodings of `movl` instructions

`movl` *S*, *D*

Source *S*	Destination *D*							
	%eax	%ecx	%edx	%ebx	%esp	%ebp	%esi	%edi
%eax	89 c0	89 c1	89 c2	89 c3	89 c4	89 c5	89 c6	89 c7
%ecx	89 c8	89 c9	89 ca	89 cb	89 cc	89 cd	89 ce	89 cf
%edx	89 d0	89 d1	89 d2	89 d3	89 d4	89 d5	89 d6	89 d7
%ebx	89 d8	89 d9	89 da	89 db	89 dc	89 dd	89 de	89 df
%esp	89 e0	89 e1	89 e2	89 e3	89 e4	89 e5	89 e6	89 e7
%ebp	89 e8	89 e9	89 ea	89 eb	89 ec	89 ed	89 ee	89 ef
%esi	89 f0	89 f1	89 f2	89 f3	89 f4	89 f5	89 f6	89 f7
%edi	89 f8	89 f9	89 fa	89 fb	89 fc	89 fd	89 fe	89 ff

D. Encodings of 2-byte functional nop instructions

Operation	Register *R*			
	%al	%cl	%dl	%bl
andb *R*, *R*	20 c0	20 c9	20 d2	20 db
orb *R*, *R*	08 c0	08 c9	08 d2	08 db
cmpb *R*, *R*	38 c0	38 c9	38 d2	38 db
testb *R*, *R*	84 c0	84 c9	84 d2	84 db

Consejos:

- Todos los gadgets que necesita se encuentran en la región de código de `rtarget` demarcada por las funciones `start-farm` y `mid_farm`.
- El ataque se puede realizar con dos gadgets.
- Cuando un gadget utiliza la instrucción `popq`, sacará datos de la pila. Como resultado, su exploit string debe contener una combinación de direcciones a gadgets y datos.

ROP Nivel 3:

Antes de abordar la fase 5, deténgase a considerar lo que ha logrado hasta este momento. En las fases 2 y 3, usted provocó que un programa ejecutara código de máquina de su propio diseño. Si `ctarget` fuera un servidor de red, usted podría haber inyectado código en una máquina distante. En la fase 4, usted traspasó dos de los principales dispositivos de seguridad que los sistemas modernos utilizan para prevenir ataques de buffer overflow. Aunque no inyectó su propio código, pudo inyectar un tipo de programa que opera uniendo secuencias de código existente. Ya obtuvo 95/100 puntos de esta tarea. Ese es un buen puntaje. Si tiene otras obligaciones, considere detenerse en este momento.

La fase 5 requiere que haga un ataque de ROP en `rtarget` para invocar la función `touch3` con un puntero a una representación de su `cookie`. Esto podría parecer no más difícil que hacer un ataque de ROP para invocar `touch2`, pero en realidad lo es. Además, la fase 5 vale solamente 5 puntos, lo cual no es una medida adecuada del esfuerzo que requiere. Considere que esto es un puntaje extra para los que quieran ir más allá de las expectativas de este curso.

Para resolver la fase 5, se pueden utilizar los gadgets en la región de código de `rtarget` demarcada por las funciones `start_farm` y `end_farm`. Además de los gadgets utilizados en la fase 4, esta granja ampliada incluye la codificación de algunas instrucciones `movl`. Las secuencias de bytes en esta parte de la granja también contienen instrucciones de 2 bytes que sirven como nop funcionales, o sea no hacen ningún cambio a registros ni a la memoria. Estas incluyen instrucciones como `andb %al, %al`, que operan sobre los bytes menos significativos de algunos de los registros, pero no cambian sus valores.

Consejos:

- Ud querrá revisar el efecto que tiene la instrucción `movl` sobre los 4 bytes superiores del registro, según lo descrito en la página 183 del libro de texto.
- La solución oficial requiere 8 gadgets (no todos son únicos).

¡Buena suerte y diviértase!

Anexo A: Sobre el uso de hex2raw

La aplicación hex2raw recibe como entrada cadenas formateadas como hexadecimales. En este formato, al igual que lo hemos visto en clase, cada byte se representa como dos dígitos hexadecimales. Por ejemplo, "123" se representa como "31 32 33 00". Es decir, los valores ASCII correspondientes a los números "123". Adicionalmente, la cadena debe terminar en el caracter nulo (0x00), el utiliza C/C++ para dar por terminada una cadena.

Los caracteres en hexadecimal que le pase al programa hex2raw deben estar separados por espacios en blanco o cambios de línea. Puede separar las distintas partes de su exploit string usando cambios de línea para una mayor legibilidad.

hex2raw tiene soporte para comentarios de bloque tipo C, es decir usando /* y */. Debe asegurarse eso sí, que haya al menos un espacio en blanco entre el inicio y el fin del comentario para que este sea satisfactoriamente ignorado. Por ejemplo, parte de su exploit string podría ser algo como:

```
48 c7 c1 f0 11 40 00 /* mov rcx, 0x40011f0 */
```

Puede almacenar su exploit string en un archivo para ser aplicado a ctarget o rtarget. Puede usarse de la siguiente forma (suponiendo que exploit.txt tiene su exploit string):

1. Puede hacer uso de pipes para pasar el string mediante hex2raw.

```
unix> cat exploit.txt | ./hex2raw | ./ctarget
```

2. Puede almacenar una cadena crida en un archivo y redireccionar la salida.

```
unix> ./hex2raw < exploit.txt > exploit-raw.txt
unix> ./ctarget < exploit-raw.txt
```

   Esto puede ser muy útil cuando esté usando gdb:

```
unix> gdb ctarget
(gdb) run < exploit-raw.txt
```

3. Puede almacenar una cadena crida en un archivo y pasar el nombre del archivo por la línea de comandos. Igualmente, esta forma de hacerlo puede ser útil para usar en conjunto con gdb.

```
unix> ./hex2raw < exploit.txt > exploit-raw.txt
unix> ./ctarget -i exploit-raw.tx
```

Anexo B: Generación de código máquina

Usar gcc como ensamblador y objdump como desensamblador es una forma más que conveniente para generar el código máquina que corresponde a las secuencias de instrucciones que se desean enviar como exploit strings. Por ejemplo, suponga que tiene el archivo ejemplo.asm con el siguiente código ensamblado:

```
# Example of hand-generated assembly code
pushq   $0xabcdef # Push value onto stack
addq    $17,%rax # Add 17 to %rax
movl    %eax,%edx # Copy lower 32 bits to %edx
```

Recuerde que el código podría tener una mezcla de instrucciones datos y comentarios.

Es posible ensamblar y desensamblar el archivo de la siguiente forma:

```
unix> gcc -c ejemplo.asm
unix> objdump -d ejemplo.o > ejemplo.d
```

El archivo recién creado, ejemplo.d tendrá lo siguiente:

```
ejemplo.o: file format elf64-x86-64
Disassembly of section .text:
0000000000000000 <.text>:
     0: 68 ef cd ab 00     pushq  $0xabcdef
     5: 48 83 c0 11        add    $0x11,%rax
     9: 89 c2              mov %eax,%edx
```

Las líneas del final muestran el código máquina generada para cada instrucción. Los números en hexadecimal (comenzando en 0) a la izquierda de la instrucción indican ese número. Por ejemplo, la instrucción "mov %edx, %eax" se representa en código máquina como "89 C2".

De esta forma, puede usar el archivo generado para pasarle a hex2raw la cadena de entrada (exploit string) para sus programas (ctarget y rtarget). Por tanto, tal y como se mencionó en el anexo A, podría convertir ejemplo.d en lo siguiente:

```
     68 ef cd ab 00   /* pushq  $0xabcdef  */
     48 83 c0 11      /* add    $0x11,%rax */
     89 c2            /* mov %eax,%edx     */
```

Esto le permitiría pasar sus programas mediante hex2raw sin necesariamente conocer el detalle de cómo se ensambla cada una de las instrucciones de ensamblador en código de la máquina.