

Seguridad

Sistemas Operativos
DC - UBA - FCEN

2do cuatrimestre de 2016

Dicho popular

Todo programa tiene bugs hasta que se demuestre lo contrario.

Problemas de seguridad

Dicho popular

Todo programa tiene bugs hasta que se demuestre lo contrario.

Veamos el **primer** programa de ejemplo de todo programador.

primero.c

```
1  #include <stdio.h>
2
3  int main(int argc, char* argv[]) {
4      printf("Hola mundo!\n");
5      return 0;
6  }
```

Este parece ser correcto.

Veamos el **segundo** programa de ejemplo de todo programador.

hola.c

```
1  #include <stdio.h>
2
3  int main(int argc, char* argv[]) {
4      char nombre[80];
5
6      printf("Ingrese su nombre: ");
7      gets(nombre);
8      printf("Hola, %s!\n", nombre);
9
10     return 0;
11 }
```

Este programa tiene un **bug de seguridad**.

Tenemos:

- Bugs “a secas” o bugs comunes,
- Bugs de seguridad

¿Cuál es la diferencia?

Tenemos:

- Bugs “a secas” o bugs comunes,
- Bugs de seguridad

¿Cuál es la diferencia?

Bugs de seguridad

Los **bugs de seguridad** son aquellos bugs que exponen **más funcionalidad** o distinta funcionalidad al usuario que la que el programa dice tener. (**Funcionalidad oculta**).

Desde el punto de vista de la **correctitud**:

- El programa escribe fuera de su memoria asignada.
- No interesa dónde escribe: Está fuera del buffer en cuestión.
- No respeta alguna precondition, postcondición, invariante, etc.
- Pincha, explota, se cuelga, no anda.

Desde el punto de vista de la **correctitud**:

- El programa escribe fuera de su memoria asignada.
- No interesa dónde escribe: Está fuera del buffer en cuestión.
- No respeta alguna precondition, postcondición, invariante, etc.
- Pincha, explota, se cuelga, no anda.

Desde el punto de vista de la **seguridad**:

- El programa hace algo que el programador no pretendía (ej: Escribir fuera del buffer.)
- Son importantes las cuestiones técnicas sobre qué hace **de más** el programa.
(ej: Qué había de importante donde escribe.)

Impacto de un bug de seguridad

Desde un punto de vista de seguridad hay, al menos, dos preguntas que siempre hay que hacer:

1. **¿Qué controla el usuario?**
2. **¿Qué información sensible hay ahí?**

Impacto de un bug de seguridad

Desde un punto de vista de seguridad hay, al menos, dos preguntas que siempre hay que hacer:

1. **¿Qué controla el usuario?**
2. **¿Qué información sensible hay ahí?**

Ejemplo: Para un programa **correcto** deberíamos responder:

1. El contenido del buffer nombre.
2. Nada.

Ejemplo: Para un programa **incorrecto** deberíamos responder:

1. El contenido del buffer nombre y todas las posiciones de memoria siguientes.
2. Todos los datos guardados en la pila por las llamadas anteriores.

Ejemplo

Situación antes de llamar a la función gets:

hola.c

```
1 #include <stdio.h>
2
3 int main(int argc, char* argv[]) {
4     char nombre[80];
5
6     printf("Ingrese su nombre: ");
7     gets(nombre);
8     printf("Hola, %s!\n", nombre);
9
10    return 0;
11 }
```

Stack

addr	val
...	...
ebp-80	nombre
...	...
ebp-4	...
ebp	ebp _{anterior}
ebp+4	return address
ebp+8	argc
ebp+12	*argv[]
...	...

1. ¿Qué controla el usuario?
2. ¿Qué información sensible hay ahí?

El usuario controla:

- Toda la pila **después** de nombre (posiciones más altas).
- En particular, el return address de `main` (`main` es una función más).
- Controla qué se ejecutará cuando `main` termine con un `return`.

Impacto:

- **Funcionalidad oculta:** Permite al usuario ejecutar cualquier código que desee luego de `main`.

A veces el impacto puede ser diferente:

- **Escalado de privilegios:** ejecutar con un usuario de mayor privilegio.
- **Autenticación indebida:** ingresar a la sesión de un usuario que no nos corresponde (no necesariamente conociendo las credenciales).
- **Denial of Service:** Deshabilitar el uso de un servicio para terceros (ej: “se cayó el sistema”).
- **Obtención de datos privados:** base de datos de clientes, códigos de tarjetas de crédito, código fuente privado, etc.

Exploit

Un **exploit** es un fragmento de código que utiliza la funcionalidad oculta del programa vulnerable. Se dice que **explota la vulnerabilidad**.

Veamos ahora este programa:

hola.c

```
#include <stdio.h>

void saludo(void) {
    char nombre[80];

    printf("Ingrese su nombre completo: ");
    gets(nombre);
    printf("Hola, %s!\n", nombre);
}

int main(int argc, char* argv[]) {
    saludo();

    return 0;
}
```

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Necesitamos saber en qué posición **del buffer** están los datos sensibles:

saludo (desensamblado)

```
1 0804841b <saludo>:
2   804841b: push    ebp
3   804841c: mov     ebp,esp
4   804841e: sub     esp,0x58
5   8048421: sub     esp,0xc
6   8048424: push    0x8048500
7   8048429: call    80482e0 <printf@plt>
8   804842e: add     esp,0x10
9   8048431: sub     esp,0xc
10  8048434: lea     eax,[ebp-0x58]
11  8048437: push    eax
12  8048438: call    80482f0 <gets@plt>
13  804843d: add     esp,0x10
14  8048440: sub     esp,0x8
15  8048443: lea     eax,[ebp-0x58]
16  8048446: push    eax
```

Stack

addr	val
...	...
ebp-0x68	param0
ebp-0x64	param1
...	...
ebp-0x58	nombre
...	...
ebp-0x08	???
ebp-0x04	???
ebp	ebp _{anterior}
ebp+0x04	return address
...	...

La dirección de nombre dentro de la función es `ebp-0x58`.

Debemos pasar como entrada un buffer de la siguiente forma:

1. 80 bytes cualquiera para llenar el buffer.
2. 8 bytes que se escribirán sobre los ???.
3. 4 bytes que se escribirán sobre el valor de ebp. (abcd)
4. 4 bytes que se escribirán sobre el valor del return address.
(efgh)

Detalle:

- fgets deja de leer con un **fin de línea** o un **fin de archivo**.

Ejemplo de buffer

00000000	30 31 32 33 34 35 36 37	38 39 30 31 32 33 34 35	0123456789012345
00000010	36 37 38 39 30 31 32 33	34 35 36 37 38 39 30 31	6789012345678901
00000020	32 33 34 35 36 37 38 39	30 31 32 33 34 35 36 37	2345678901234567
00000030	38 39 30 31 32 33 34 35	36 37 38 39 30 31 32 33	8901234567890123
00000040	34 35 36 37 38 39 30 31	32 33 34 35 36 37 38 39	4567890123456789
00000050	30 31 32 33 34 35 36 37	61 62 63 64 65 66 67 68	01234567abcdefgh
00000060			

¿Qué sería deseable que ocurra si pasamos este buffer como entrada?

¿Qué sería deseable que ocurra si pasamos este buffer como entrada?

Segmentation fault¹

¹Es una de las pocas veces que el objetivo es que el resultado sea Segmentation fault

¿Qué sería deseable que ocurra si pasamos este buffer como entrada?

Segmentation fault¹

Demo

¹Es una de las pocas veces que el objetivo es que el resultado sea Segmentation fault

- Ya controlamos eip, por lo tanto, podemos saltar a donde queramos
- ¿y ahora? ¿A **dónde** saltamos?

- Ya controlamos eip, por lo tanto, podemos saltar a donde queramos
- ¿y ahora? ¿A **dónde** saltamos?

Opción 1: Saltar a nuestro propio buffer.

- Ya controlamos eip, por lo tanto, podemos saltar a donde queramos
- ¿y ahora? ¿A **dónde** saltamos?

Opción 1: Saltar a nuestro propio buffer.

Opción 2: Saltar a código del programa que haga lo que nosotros queremos (por ejemplo, "authenticate()").

- Ya controlamos eip, por lo tanto, podemos saltar a donde queramos
- ¿y ahora? ¿A **dónde** saltamos?

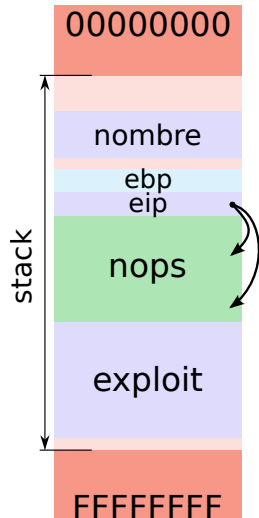
Opción 1: Saltar a nuestro propio buffer.

Opción 2: Saltar a código del programa que haga lo que nosotros queremos (por ejemplo, "authenticate()").

Opción 3: Saltar a código suelto del programa que ejecuta parte de lo que queremos y luego permite seguir controlando el flujo (realizamos varios saltos).

Opción 1:

- return-address una dirección dentro de nuestro buffer.
- Saltamos dentro de nuestro buffer.
- No es necesario saber la dirección **exacta** del exploit.



Demo

Vulnerabilidades Clásicas

- Buffer Overflow (ya la vimos)
- Integer Overflow
- Format String
- SQL Injection

...y cada día se descubren nuevas ideas.

Integer Overflow

- Ocurre cuando un valor entero se pasa del tamaño de la variable donde está almacenado.
- No es un problema de seguridad de por sí, pero puede ser usado en combinación con otros problemas.

Esta vulnerabilidad se basa en un mal diseño de la función `printf` y similares:

```
int printf(const char *format, ...);
```

1

El primer parámetro contiene información sobre cómo darle formato a los distintos parámetros:

- Hay opciones para mostrar un número en decimal (`%d`), hexa (`%x`), etc...
- Hay modificadores para indicar que el parámetro es de 1, 2, 4 u 8 bytes (`hh`, `h`, `l`);

```
long a = 1;
double pi = 3.14159265358979;
printf("a=%ld pi=%lf", a, pi);
```

1

2

3

- Hay opciones para mostrar el contenido de un puntero (string o char*) hasta la primer ocurrencia de un 0 (%s).
- Hay modificadores para elegir qué parámetro se desea mostrar en ese punto. Ejemplo:

```
printf("%2$d %1$s", "hola", 123);
```

1

- Hay opciones para mostrar el contenido de un puntero (string o char*) hasta la primer ocurrencia de un 0 (%s).
- Hay modificadores para elegir qué parámetro se desea mostrar en ese punto. Ejemplo:

```
printf("%2$d %1$s", "hola", 123);
```

1

- Hay una opción para **escribir** en el puntero pasado por parámetro cuántos caracteres fueron escritos hasta el momento en el stream.

```
printf("%d %n %d", 123, &a, 456);
```

1

Parece tener alguna limitada utilidad para realizar un parser (scanf también la tiene), pero se usa muy poco o nunca.

Format String

Justo antes de realizar el call `printf` la situación es así:

fs.c

```
1 #define MAX_BUF 2048
2
3 void echo(void) {
4     char echobuf[MAX_BUF];
5
6     fgets(echobuf, MAX_BUF, stdin);
7     printf(echobuf);
8 }
```

La dirección de `echobuf` dentro de la función es `ebp-0x808`.

Stack

addr	val
...	...
ebp-0x818	&echobuf
...	...
ebp-0x808	echobuf
...	...
ebp-0x008	???
ebp-0x004	???
ebp	ebp _{main}
ebp+0x004	ret addr _{main}
...	...

¿Qué controla el usuario?

¿Qué controla el usuario?

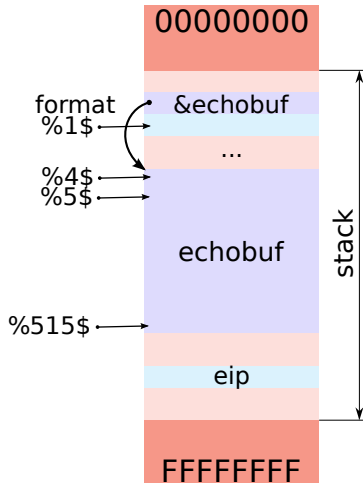
El contenido de echobuf y el **format string** de la llamada a printf.

¿Qué controla el usuario?

El contenido de `echobuf` y el **format string** de la llamada a `printf`.

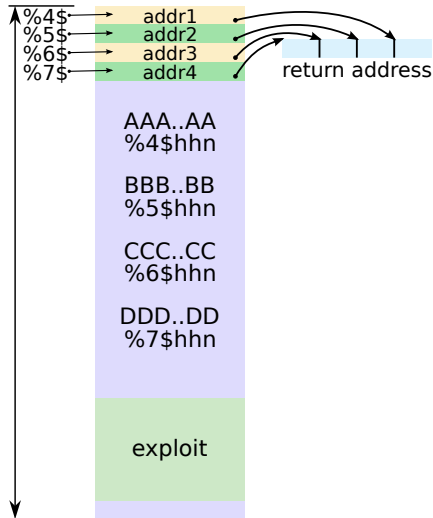
Estrategia:

- Enviar en `echobuf` caracteres `%n` para escribir en memoria
- Enviar en el mismo buffer direcciones de memoria que vamos a usar como parámetros (puntero).
- Usar la variante que nos permite elegir el número de parámetro a usar. Con esto ya podemos elegir dónde escribir.



Format String

- Para escribir un número de 32bits deberíamos enviar por stdout esa cantidad de caracteres (recoredemos %n guarda la cantidad de caracteres escritos). No entran en el buffer.
- Usamos la variante que escribe número de 1 byte (%hhn) y aprovechamos el integer overflow para escribir de a 1 byte.



1. Hacemos 4 escrituras de 1 byte.
2. Usamos este esquema de escritura para sobrescribir información sensible: return address.
3. Saltamos a otro código: ¿Cuál?
Ponemos al final del buffer nuestro exploit y saltamos a esa posición.

1. Hacemos 4 escrituras de 1 byte.
2. Usamos este esquema de escritura para sobrescribir información sensible: return address.
3. Saltamos a otro código: ¿Cuál?
Ponemos al final del buffer nuestro exploit y saltamos a esa posición.

El buffer quedaría así:

buffer

```
"addr1addr2addr3addr4 AA...AA%4$hhn BB...BB%5$hhn  
CC...CC%6$hhn DD...DD%7$hhn paa...aaading codigo"
```

Demo

- Ejemplo de consulta a una base de datos para la validación de un usuario:

```
SELECT *  
FROM usuarios  
WHERE login=' '$USER' ' AND pass=' '$PASS' '
```

1

2

3

donde \$USER y \$PASS son controlados por el usuario.

- Si hay una entrada en la base de datos con esas credenciales, el SELECT lo devuelve.
- **¿Seguro?**

- Ejemplo de consulta a una base de datos para la validación de un usuario:

```
SELECT *
FROM usuarios
WHERE login=' '$USER' ' AND pass=' '$PASS' '
```

1

2

3

donde \$USER y \$PASS son controlados por el usuario.

- Si hay una entrada en la base de datos con esas credenciales, el SELECT lo devuelve.
- **¿Seguro?**
- ¿Qué pasaría si el usuario ingresara lo siguiente?

USER: admin

PASS: nose'' OR ''1''=''1

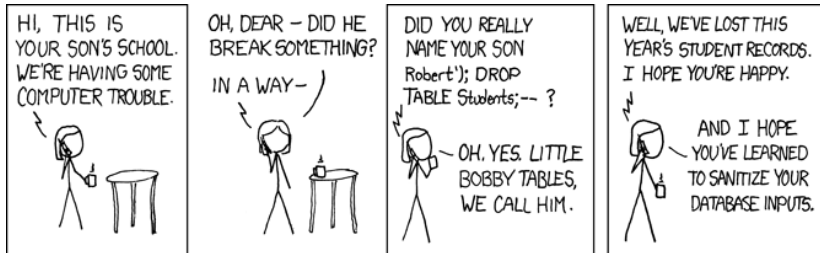
La consulta queda conformada de la siguiente manera:

```
SELECT *                                1
FROM    usuarios                        2
WHERE    login=' 'admin' ' AND pass=' 'nose' ' OR ' '1' '=' '1' ' 3
```

- El reemplazo se realiza en el programa y la consulta ya reemplazada se envía a la base de datos.
- El SELECT devuelve la entrada de la base de datos aún cuando el *password* ingresado no es el correcto.
- El programa asume que el *password* es correcto porque el SELECT devolvió la entrada.

El impacto puede ser diferente:

- Escalado de privilegio o login indebido.
- Destrucción de datos, por ejemplo, `PASS: "'; DROP TABLE usuarios; --"`
- Obtención de datos privados.



"Robert'); DROP TABLE Students; --" alias "Little Bobby Tables".

Algunos sistemas operativos implementan uno o más mecanismos para **protegerse** de posibles ataques. Los principales son:

- DEP: *Data Execution Prevention*
- ASLR: *Address Space Layout Randomization*
- Stack Canaries: También conocido como *Stack Guards* or *Stack Cookies*

Todos estos son mecanismos que se utilizan en conjunción para intentar mitigar diferentes clases de ataques.

- Ninguna región de memoria debería ser al mismo tiempo **escribible** y **ejecutable**
- Ejemplos básicos: Heap y Stack
- Se implementan con ayuda del hardware, por ejemplo, bit NX (en Intel)
- Impide ataques básicos (como los vistos hoy). Es decir, ya no se puede inyectar código.

- Ninguna región de memoria debería ser al mismo tiempo **escribible** y **ejecutable**
- Ejemplos básicos: Heap y Stack
- Se implementan con ayuda del hardware, por ejemplo, bit NX (en Intel)
- Impide ataques básicos (como los vistos hoy). Es decir, ya no se puede inyectar código.
- ¿Esto significa que ya no se puede explotar un programa vulnerable?

- Ninguna región de memoria debería ser al mismo tiempo **escribible** y **ejecutable**
- Ejemplos básicos: Heap y Stack
- Se implementan con ayuda del hardware, por ejemplo, bit NX (en Intel)
- Impide ataques básicos (como los vistos hoy). Es decir, ya no se puede inyectar código.
- ¿Esto significa que ya no se puede explotar un programa vulnerable?
- **No!** Hay técnicas para “bypassear” esta protección: ROP

- Modifica de manera aleatoria la dirección base de regiones importantes de memoria entre las diferentes ejecución de un proceso
- Por ejemplo: Heap, Stack, LibC, etc.
- Ver la salida del comando: `cat /proc/self/maps`
- Impide ataques que utilizan direcciones “hardcodeadas” (como los vistos hoy)
- No todo se “randomiza”. Por lo general, la sección de texto de un programa no lo cambia. Para que lo haga, se tiene que compilar especialmente para ser *Position Independent Code*.
- Sí está compilado con PIE el sistema operativo puede cambiar su dirección base entre sucesivas ejecuciones.
- Al igual que DEP, también es “bypassable” (aunque puede ser más difícil)

- Implementado a nivel del compilador
- Se coloca un valor en la pila luego de crear el *stack frame*
- Antes de retornar de la función se verifica que el valor sea el correcto.
- La idea es proteger el valor de retorno de la función de posibles *buffer overflows*
- Esta técnica, también es “bypassable”.

Todas estas técnicas pueden vencerse con menor o mayor esfuerzo individualmente, sin embargo, para vulnerar la seguridad de un sistema se deben vencer todas al mismo tiempo. Esto incrementa bastante la dificultad para lograrlo de manera exitosa.

¿Preguntas?