UNIVERSIDAD DE MÁLAGA

PROGRAMACIÓN SEGURA

Práctica 3: Buffer overflow estático

CIBERSEGURIDAD E INTELIGENCIA ARTIFICIAL

Pablo Jesús Delgado Muñoz

Octubre de 2024

ÍNDICE GENERAL

CAPÍTULO I: CONOCIMIENTOS PREVIOS

1.1 Sockets en C

Un socket es un **punto final** de comunicación entre dos dispositivos, nos permite que dos programas que pueden estar en diferentes equipos intercambien información. La principal diferencia es que en el socket tenemos una comunicación **activa-activa**. En una conexión normal cliente-servidor en la que el cliente hace por ejemplo, un http get al servidor para cargar una web, simplemente este servidor responderá y enviará al cliente la información necesaria para que este haga cierta acción.
El problema viene cuando nosotros necesitamos que esa comunicación se actualice constantemente, por lo tanto, para no estar constantemente enviando solicitudes get, se establece esta comunicación ACTIVA-ACTIVA, en la que el servidor va a ser capaz de enviar notificaciones a la máquina cliente poder interactuar con ello.
Esto permite ser notificados en tiempo real cuando algo sucede.
Los sockets se usan mucho en aplicaciones de red y operan bajo un modelo cliente-servidor.

- **Servidor escucha** un puerto específico y espera la conexión.
- **Cliente** inicia la comunicación **solicitando** conectarse a ese puerto del servidor.

Tenemos diferentes tipos de sockets, entre ellos:

- **Sockets de Internet**
- **Sockets UNIX**

Además en C existe una biblioteca de sockets que perite crear y manejar sockets con las siguientes funciones:

- **socket()**: Crea sockets.
- **bind()**: Asocia socket a puerto.
- **listen()**: Permite poner al socket en modo de escucha.
- **accept()**: Acepta conexión entrante.
- **connect()**: Conecta socket cliente al socket servidor.

Sabemos que inetd es un daemon, programa que se encuentra siempre funcionando en segundo plano y que gestiona muchos servicios de red. En lugar de que cada servicio esté ejecutándose y esperando conexciones, inetd escucha en una serie de puertos asociados a servicios como fingerd, telnet o ftp, y cuando detecta una conexión entrante, inicia el proceso correspondiente a ese servicio y se lo delega. Esto reduce el número de procesos en ejecución y lo hace todo mucho más eficiente. En esta práctica vamos a ver el protocolo fingerd.

1.1.1 Servicio fingerd

El daemon fingerd es un protocolo simple que proporciona una interfaz para el comando finger en varios sitios de red. Finger devuelve un informe de estado sobre un usuario conectado a un sistema remoto. El dameon fingerd se instala en el servidor y lo que hace es escuchar las solicitudes TCP en el puerto 79.
El cliente no tendrá instalado este daemon pero ejecutará el comando finger que enviará una solicitud al servidor preguntando por información de cierto usuario. Finger se conecta al servidor en el puerto 79 haciendo uso de TCP y el servidor recibe la solicitud, procesándola a través de fingerd, enviando la respuesta de vuelta al cliente.
La versión que vamos a tratar en esta práctica es fingerd de UNIX 4.3 BSD, 1986. En esta actualización de fingerd se corregían algunos errores y se limitaban viertos detalles de los usuarios para mejorar la seguridad aunque seguía siendo vulnerable a ataques de buffer overflow que ahora trataremos.

1.1.2 Entrada y salida en C (stdio)

Se hace uso de la biblioteca stdio que ofrece funciones que permiten la entrada y salida de datos, incluyendo:

- **stdin**: entrada estándar (teclado).
- **stdout**: salida estándar (pantalla o consola).
- **stderr**: error estándar (mensajes de error).

También se dispone la función gets(), que lee una línea completa con stdin pero no pone un límite de caracteres a leer, lo cual puede llegar a ser peligroso.

1.1.3 BSD4.3

Fue una de las versiones de UNIX más influyentes, incorporando avances en redes y sistemas de archivos. Fue tan importante, que muchos sistemas operativos como FreeBSD, OpenBSD y macOS, derivan de BSD.

CAPÍTULO II: DETECCIÓN DEL PROBLEMA

En este capítulo vamos a jugar con el código y buscar su problema, así como tratar de pensar qué puede causar dicho fallo o riesgos que conlleva.

2.1 Herramientas utilizadas

Para abrir, analizar y leer código, podemos utilizar una gran variedad de editores de texto y combinar estos con diferentes herramientas que facilitan el trabajo de desarrollo y depuración. Destacamos **Vim** , **SublimeText** o **Kate** como editores de texto.
Para facilitar la depuración, emplearemos **Valgrind**, un conjunto de herramientas muy útil para encontrar errores de administración de memoria y subprocesamiento en código.

Cuando recibimos código para revisar en busca de errores, muchas veces podemos encontrar que este no se encuentra correctamente indentado u organizado. Por lo tanto, para evitar cometer errores organizando el código por nosotros mismos, emplearemos **Indent**.

2.2 Análisis código original

Al tratar de compilar ejecutar el código original nos devuelve este eror:
./fingerd_original: getpeername: **Socket operation on non-socket**

Por lo tanto, vamos a comentar la parte del código donde se hace esta verificación:

```
//if (getpeername(0, &sin, &i) < 0)
//   fatal(argv[0], "getpeername");
```

Ahora vamos a compilar y ejecutar de nuevo:
Vemos que al compilar se nos avisa del uso de una función insegura gets() - **aviso: the 'gets' function is dangerous and should not be used.** Hasta ahora hemos encontrado esta vulnerabilidad, pero en la siguiente sección analizaremos con GDB y Valgrind para encontrar el resto de problemas.

Una vez ejecutamos el programa nos permite leer de teclado: Cuando un cliente se conecta, el código utiliza la función `gets` para leer los datos enviados por el cliente. Estos datos deberían normalmente tener el nombre de usuario del que se quiere conseguir información. Después el programa debería de invocar a finger con la entrada que le ha pasado el usuario. Después la salida debería ser recogida y reenviada al cliente.
Claro que todo esto sería en el caso de una conexión cliente servidor, pero aquí estamos tratando de replicar la vulnerabilidad en nuestro propio ordenador por lo tanto vamos a sustituir cliente por stdin y servidor por stdout.

2.3 Proceso de detección del problema con GDB

Primero vamos a usar GDB y leer la memoria, pero antes vamos a obviamente compilar fingerd con el flag de debug para poder usarlo.
Ejecutamos con GDB y ponemos un breakpoint antes de leer, para poder ver el bloque de memoria antes de hacer el buffer overflow con el gets(). (gdb) next

gets(line); //leemos de stdio, enganchado al socket porque fingerd lo hemos lanzado con inetd

Memoria antes de escribir en gets(): 0x7fffffffdc20: 0x00000000 0x00000000 0x00000000 0x00000000

0x7fffffffdc30: 0x6d6f682f 0x73702f65 0x636f442f 0x6e656d75

0x7fffffffdc40: 0x73702f65 0x636f442f 0x6e656d75 0x502f7374

0x7fffffffdc50: 0x67756265 0x676e6967 0x45485300 0x2f3d4c4c

0x7fffffffdc60: 0x5f004554 0x62696c5f 0x61655f63 0x5f796c72

0x7fffffffdc70: 0x00000060 0x00000000 0x00000000 0x00000000

0x7fffffffdc80: 0x0000ff00 0x000000ff 0x00000000 0x00ff0000

0x7fffffffdc90: 0x2f2f2f2f 0x2f2f2f2f 0x2f2f2f2f 0x2f2f2f2f

(gdb) next

bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb...
Memoria después de escribir en gets:
0x7fffffffdc20: 0x00000000 0x00000000 0x00000000 0x00000000

0x7fffffffdc30: 0x6d6f682f 0x73702f65 0x636f442f 0x6e656d75

0x7fffffffdc40: 0x62626262 0x62626262 0x62626262 0x62626262
0x7fffffffdc50: 0x62626262 0x62626262 0x62626262 0x62626262

0x7fffffffdc60: 0x62626262 0x62626262 0x62626262 0x62626262

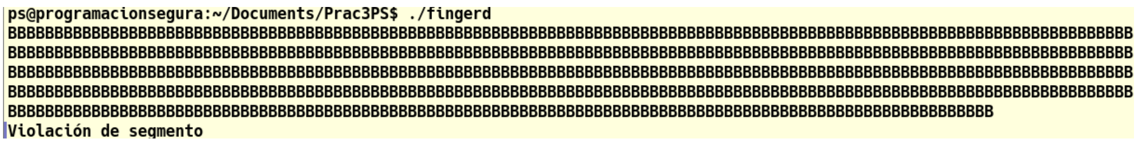
0x7fffffffdc70: 0x62626262 0x62626262 0x62626262 0x62626262

0x7fffffffdc80: 0x62626262 0x62626262 0x62626262 0x62626262

0x7fffffffdc90: 0x62626262 0x62626262 0x62626262 0x62626262

0x7fffffffdca0: 0x62626262 0x62626262 0x62626262 0x62626262

0x7fffffffdcb0: 0x62626262 0x62626262 0x62626262 0x62626262
Podemos claramente ver cómo se ha escrito en memoria sin controlar el tamaño. Esto llegará a tal punto que se toque el return address, hasta que pete el programa y nos de violación de segmento:



También tenemos que tener en cuenta que este programa lanzará el comando fingerd que se guarda en av, pero si se sobreescribe esta dirección, habrá comportamiento errático.
Error en execv:
av[0] = ”finger”;
execv(/usr/ucb/finger”, av);

2.4 Análisis con Valgrind

Ejecutaremos valgrind con los siguientes flags para un análisis más a fondo:
valgrind –leak-check=full –tool=memcheck –track-origins=yes–show-reachable=yes –num-callers=10 ./fingerd_debugging

Pasaremos una entrada grande para que se sobreescriba la memoria...

==4217== LEAK SUMMARY:

==4217== definitely lost: 0 bytes in 0 blocks

==4217== indirectly lost: 0 bytes in 0 blocks

==4217== possibly lost: 0 bytes in 0 blocks

==4217== still reachable: 1,024 bytes in 1 blocks

==4217== suppressed: 0 bytes in 0 blocks

1,024 bytes in 1 blocks are still reachable in loss record 1 of 1

==4217== by 0x48ED0FC: gets (iogets.c:38)

==4217== HEAP SUMMARY:

==4217== in use at exit: 1,024 bytes in 1 blocks

==4217== total heap usage: 3 allocs, 2 frees, 5,592 bytes allocated

Jump to the invalid address stated on the next line

==4217== at 0x4242424242424242: ???

==4217== by 0x4242424242424241: ???

==4217== by 0x4242424242424241: ???

==4217== by 0x4242424242424241: ???

Bad permissions for mapped region at address 0x4242424242424242

==4217== at 0x4242424242424242: ???

==4217== by 0x4242424242424241: ???

==4217== by 0x4242424242424241: ???

CAPÍTULO III: SOLUCIONES

3.1 Cambiar gets() por fgets()

Lo que haremos para que no haya errores de buffer overflow será cambiar la función gets() por fgets() y además validar la entrada para que si se manda una entrada de gran longitud, ésta sea rechazada.

Cambiamos el código de tal forma que queda:

fgets(line, sizeof(line), stdin);

Ojo, se pone sizeof(line) y no se resta uno porque ya se tiene en cuenta el caracter nulo.
Si ahora tratamos de realizar un buffer overflow o escribir más de la cuenta veremos que no es posible, solo se escribirá de nuestra entrada lo que si debe entrar y el resto se ignorará.
Ahora ejecutaremos de nuevo con valgrind e introduciremos una entrada mayor a 512 caracteres y veremos que no se produce ningún error.

==2736== Command: ./fingerd_fixed

BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB...

==2738== HEAP SUMMARY:

==2738== in use at exit: 0 bytes in 0 blocks

==2738== total heap usage: 1 allocs, 1 frees, 1,024 bytes allocated

==2738== All heap blocks were freed – no leaks are possible

==2738== For lists of detected and suppressed errors, rerun with: -s

==2738== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)

==2736== HEAP SUMMARY:

==2736== in use at exit: 0 bytes in 0 blocks

==2736== total heap usage: 3 allocs, 3 frees, 5,592 bytes allocated

==2736== All heap blocks were freed – no leaks are possible

==2736== For lists of detected and suppressed errors, rerun with: -s

==2736== ERROR SUMMARY: **0 errors from 0 contexts (suppressed: 0 from 0)**

Finalmente, el código queda así:

```
#include <stdio.h>
#include <ctype.h>

main(argc, argv)
        char *argv[]; //puntero a un puntero.
{
        register char *sp; //puntero a char.
        char line[512]; //line
        struct sockaddr_in sin; //estructura llamada sin
        int i, p[2], pid, status; //entero i, p, pid, status...
        FILE *fp; //fp
        char *av[4]; //4 punteros a av en cada una de sus posiciones
        i = sizeof (sin);
//        if (getpeername(0, &sin, &i) < 0)
//                fatal(argv[0], "getpeername"); //le pide a inetd el nombre
        line[0] = '\0';
        fgets(line, sizeof(line), stdin);//leemos de stdio enganchado socket
        sp = line;
        av[0] = "finger"; //metemos en el buffer la constante finger
        i = 1;
        while (1) {
                while (isspace(*sp))
                        sp++;
                if (!*sp)
                        break;
                if (*sp == '/' && (sp[1] == 'W' || sp[1] == 'w')) {
                        sp += 2;
                        av[i++] = "-l";
                }
                if (*sp && !isspace(*sp)) {
                        av[i++] = sp;
                        while (*sp && !isspace(*sp))
                                sp++;
                        *sp = '\0';
                }
        }
        av[i] = 0;
```

```
        if (pipe(p) < 0)
                fatal(argv[0], "pipe");
        if ((pid = fork()) == 0) {
                close(p[0]);
                if (p[1] != 1) {
                        dup2(p[1], 1);
                        close(p[1]);
                }
                execv("/usr/ucb/finger", av); //ejecutamos lo de stdio (entrada)
                _exit(1);
        }
        if (pid == -1)
                fatal(argv[0], "fork");
        close(p[1]);
        if ((fp = fdopen(p[0], "r")) == NULL)
                fatal(argv[0], "fdopen");
        while ((i = getc(fp)) != EOF) {
                if (i == '\n')
                        putchar('\r');
                putchar(i);
        }
        fclose(fp);
        while ((i = wait(&status)) != pid && i != -1)
                ;
        return(0);
}
fatal(prog, s)
        char *prog, *s;
{
        fprintf(stderr, "%s: ", prog);
        perror(s);
        exit(1);
}
```