Sistemas Operativos

Práctica 7: Protección y seguridad

Notas preliminares

- Los ejercicios marcados con el símbolo ★ constituyen un subconjunto mínimo de ejercitación. Sin embargo, aconsejamos fuertemente hacer todos los ejercicios.

Ejercicio 1

En muchos sistemas las contraseñas de los usuarios no se almacenan, sino que se almacena solo el resultado de una función de *hash*. Suponiendo que la función de *hash* utilizada entrega valores de 64 *bits* y que los resultados se encuentran *bien distribuidos*, responder:

a) ¿Cómo puede verificar el sistema si un usuario ingresó su contraseña correctamente, si no la tiene almacenada en el sistema?
b) Si bien existen varias contraseñas que entregan el mismo valor de *hash* (colisión), ¿qué probabilidad hay de acertar el valor de *hash* almacenado?
c) ¿En cuántos años se puede tener un 50 % de probabilidad de haber acertado al valor de *hash* almacenado, dado que pueden probarse mil millones de contraseñas por segundo?
d) Conocer el valor de *hash* de una contraseña no implica conocer una contraseña que genere ese valor. Suponer que se pueden probar contraseñas a la misma velocidad que en el ítem anterior, pero nos informan que la contraseña es de, a lo sumo, 6 caracteres, y que estos solo son letras minúsculas o dígitos. ¿Cuánto tiempo hay que esperar para averiguar la clave?

Ejercicio 2

Los sistemas de autenticación remota, donde el usuario se debe autenticar a través de un canal inseguro (que puede ser visto por terceros), como por ejemplo POP3, a menudo no transmiten la contraseña en el proceso de autenticación para evitar que ésta sea interceptada.

a) ¿Sería seguro autenticar al usuario enviando por el canal (desde el usuario hasta el sistema) el *hash* de la contraseña? ¿A qué ataque es vulnerable este esquema?
b) Un esquema *Challenge-Response* basado en una función de *hash* opera de la siguiente manera:

   - El sistema envía una cadena *seed* elegida al azar.
   - El usuario responde con el resultado de *hash(seed + contraseña)*.
   - El sistema hace la misma cuenta de su lado y verifica el resultado.

   Si un atacante captura toda esta conversación, ¿sería capaz de realizar un ataque de fuerza bruta sobre la contraseña sin realizar ninguna interacción con el servidor (ataque *offline*)?

Ejercicio 3 ★

Si el usuario tiene control sobre la entrada estándar, el siguiente código tiene problemas de seguridad.

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

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

Se considera como un problema de seguridad que un usuario atacante pueda realizar operaciones que no le eran permitidas, ya sea tanto acceder a datos privados, ejecutar código propio en otro programa o inhabilitar un servicio. Determinar:

a) ¿Dónde se introduce el problema de seguridad?
b) ¿Qué datos del programa (registros, variables, direcciones de memoria) pueden ser controladas por el usuario?
c) ¿Es posible sobreescribir la dirección de retorno a la que vuelve la llamada de alguna de las funciones `printf` o `gets`?
d) ¿Se soluciona el problema de seguridad si se elimina el segundo `printf`?

Ejercicio 4 ★

El siguiente es un sistema de *login* que valida los datos contra una base de datos.

```
struct credential {
    char name[32];
    char pass[32];
}
bool login(void) {
    char realpass[32];
    struct credential user;
    // Pregunta el usuario
    printf("User: ");
    fgets(user.name, sizeof(user), stdin);
    // Obtiene la contraseña real desde la base de datos y lo guarda en realpass
    db_get_pass_for_user(user.name, realpass, sizeof(realpass));
    // Pregunta la contraseña
    printf("Pass: ");
    fgets(user.pass, sizeof(user), stdin);

    return strncmp(user.pass, realpass, sizeof(realpass)-1) == 0;
    // True si user.pass == realpass
}
```

Suponiendo que la función `db_get_pass_for_user()` es totalmente correcta y no escribe fuera de `realpass()`:

a) Hacer un diagrama de cómo quedan ubicados los datos en la pila, indicando claramente en qué sentido crece la pila y las direcciones de memoria. Explicar, sobre este diagrama, de qué datos (posiciones de memoria, buffers, etc.) tiene control el usuario a través de la función `fgets()`.
b) Indicar un valor de la entrada, que pueda colocar el usuario, para loguearse como `admin` sin conocer la contraseña de este.

Ejercicio 5

La siguiente función se utiliza para verificar la contraseña del usuario actual `user` en el programa que desbloquea la sesión luego de activarse el protector de pantalla del sistema. El usuario ingresa la misma por teclado.

```
1  bool check_pass(const char* user) {
2      char pass[128], realpass[128], salt[2];
3      // Carga la contraseña (encriptada) desde /etc/shadow
4      load_pass_from(realpass, sizeof(realpass), salt, user, "/etc/shadow");
5
6      // Pregunta al usuario por la contraseña.
7      printf("Password: ");
8      gets(pass);
9
10     // Demora de un segundo para evitar abuso de fuerza bruta
11     sleep(1);
12
13     // Encripta la contraseña y compara con la almacenada
14     return strcmp(crypt(pass, salt), realpass) == 0;
15 }
```

- `void load_pass_from(buf, buf_size, salt, user, file)` lee del archivo `file` la contraseña encriptada del usuario `user` y la almacena en el buffer `buf` escribiendo a lo sumo `buf_size-1` caracteres y un `"\0"` al final de estos; guarda además en `salt` el valor de los dos caracteres que se usaron para encriptar la contraseña guardada en `file`. Si el usuario no se encuentra, se almacena en `buf` un valor de contraseña inválido, no generable por la función `crypt`.
- `char* crypt(pass, salt)` devuelve la contraseña `pass` encriptada usando el valor `salt`.
- `/etc/shadow` almacena información sensible del sistema y un usuario común no tiene acceso a este archivo, tiene permisos `r-- --- --- root root /etc/shadow`.

a) La línea 4 del código hace un llamado a función que debe leer el archivo `/etc/shadow`, protegido para los usuarios sin privilegio. Explicar qué mecanismo permite lanzar este programa a un usuario normal y hace que éste tenga acceso a `/etc/shadow` sin alterar los permisos del archivo.

b) Explicar por qué esta función tiene problemas de seguridad. ¿Qué datos sensibles del programa controla el usuario?

c) Si solo los usuarios que ya ingresaron al sistema, que de por sí pueden ejecutar cualquier programa desde la terminal, son los que pueden utilizar el protector de pantalla y por tanto esta función, ¿este problema de seguridad en el protector de pantalla del sistema compromete la integridad del sistema completo? Justificar.

Ejercicio 6 ★

Considerando que el usuario controla el valor del parámetro `f`, analizar el siguiente código de la función `signo`.

```
#define NEGATIVO 1
#define CERO 2
#define POSITIVO 3
int signo(float f) {
    if (f <  0.0) return NEGATIVO;
    if (f == 0.0) return CERO;
    if (f >  0.0) return POSITIVO;

    assert(false && "Por aca no paso nunca");
    return 0; // Si no pongo esto el compilador se queja =(
}
```

a) ¿El usuario tiene alguna forma de que se ejecute el `assert()`? **Pista**: Piense en el formato IEEE-754.
b) En las versiones "*release*" los `assert` suelen ignorarse (opción de compilación). ¿Sería *seguro* utilizar la función `signo` sobre un dato del usuario y esperar como resultado alguno de los valores 1, 2 o 3?

Ejercicio 7

Un esquema de protección implementado por algunos sistemas operativos consiste en colocar el *stack* del proceso en una posición de memoria al azar al iniciar (*stack randomization*). Indique cuáles de las siguientes estrategias de ataque a una función con problemas de seguridad se ven fuertemente afectadas por esta protección, y explique por qué:

a) Escribir el valor de retorno de una función utilizando un *buffer overflow* sobre un buffer en *stack* dentro de dicha función.
b) Utilizar el control del valor de retorno de una función para saltar a código externo introducido en un *buffer* en *stack* controlado por el usuario.
c) Utilizar el control del valor de retorno de una función para ejecutar una *syscall* particular (por ejemplo `read`) que fue usada en otra parte del programa.

Ejercicio 8 ★

Suponiendo que el usuario controla el parámetro `dir`, el siguiente código, que corre con mayor nivel de privilegio, intenta mostrar en pantalla el directorio `dir` usando el comando `ls`. Sin embargo, tiene problemas de seguridad.

```
#define BUF_SIZE 1024
void wrapper_ls(const char * dir) {
    char cmd[BUF_SIZE];
    snprintf(cmd, BUF_SIZE-1, "ls %s", dir);
    system(cmd);
}
```

a) Muestre un valor de la cadena `dir` que además de listar el directorio actual muestre el contenido del archivo `/etc/passwd`.
b) Posteriormente se reemplazó esta por la función `secure_wrapper_ls` donde el tercer parámetro de `snprintf` en vez de ser `"ls %s"` se reemplaza por `"ls \"%s\""`. Muestre que la modificación no soluciona el problema de seguridad.
c) Posteriormente se agregó una verificación de la cadena `dir`: no puede contener ningún carácter "`;`". Muestre que esta modificación tampoco soluciona el problema.
d) Proponga una versión de esta función que no tenga problemas de seguridad.

Ejercicio 9

La siguiente es una función que determina si una posición de memoria `addr` se encuentra dentro de una página de memoria `page_addr` considerando que las páginas están alineadas a direcciones que terminan en `bits_per_page` bits en 0 y tienen un espacio de direcciones de esa misma cantidad de *bits*. Por ejemplo, la llamada `addr_in_page(0x12345000, 12, 0x12345ABC)` devuelve verdadero pues la dirección `0x12345ABC` se encuentra en la página de 4 KB (12 *bits* de direcciones) que comienza en `0x12345000`, lo cual es un esquema típico en la arquitectura x86. Notar que del primer parámetro se ignoran los *bits* menos significativos, usualmente utilizados para atributos.

```
bool addr_in_page(int page_addr, int bits_per_page, int addr) {
    int mask = (-1) << bits_per_page;
    return ((page_addr & mask) == (addr & mask));
}
```

Es claro que, si una dirección de memoria se encuentra dentro de una página alineada a `b` *bits*, entonces también se encuentra dentro de una página de dirección equivalente pero de más de `b` *bits* de alineación.

Sin embargo, dadas tres variables `int paddr, b, addr;` controladas por el usuario, mostrar valores para estas variables que hagan verdadera la siguiente evaluación:

```
(addr_in_page(paddr, b, addr) && !addr_in_page(paddr, b+1, addr))
```

Pista: Pensar en los rangos posibles de las variables y en aritmética circular.

Curiosidades

Los siguientes ejercicios explotan algún tecnisismo o son de una dificultad mayor a la que se pedirá en el contexto de la materia. De todos modos, complementan los conceptos de esta práctica.

Ejercicio 10

Suponiendo que el usuario controla la entrada estándar, el siguiente código tiene problemas de seguridad.

```
#define BUF_SIZE 1024
int suma_indirecta(void) {
    int buf[BUF_SIZE];
    int i, v;
    memset(buf, 0, sizeof(buf));
    while (cin >> i >> v) { // Leo el índice y el valor
        if (i == -1) break; // Un índice -1 significa que tengo que terminar.
        if (i < BUF_SIZE) buf[i] = v; // Guardo el valor en el buffer
    }

    // Calculo la suma de los valores
    v = 0
    for (i=0; i < BUF_SIZE; i++)
        v += buf[i];

    return v;
}
```

a) El código verifica que el valor de `i` no se pase del tamaño del *buffer* (`BUF_SIZE`). ¿Es suficiente esta verificación para garantizar que no se escribe fuera de `buf`?

b) Considerando que la dirección de retorno de esta función (`suma_indirecta`) se encuentra en una posición de memoria más alta (mayor) que `buf`, ¿existe algún valor de `i` que permita sobreescribirla al ejecutar el cuerpo del `while`? Justifique. **Pista**: Pensar en la aritmética de punteros que se realiza dentro del cuerpo del `while`.

c) Si el compilador protegiera el *stack* colocando un *canario*[a] de valor desconocido (incluso *random*), una posición de memoria antes (una posición menor) del *return address* de cada función, ¿aún es posible modificar el retorno de la función `suma_indirecta` y retornar de la misma satisfactoriamente?

[a] Conocido como *stack-protector*.

Ejercicio 11

Suponiendo que el usuario controla la entrada estándar, el siguiente código tiene problemas de seguridad.

```
#define MAX_BUF 4096
void saludo(void) {
    char nombre[MAX_BUF];
    printf("Ingrese su nombre: ");
    fgets(nombre, MAX_BUF, stdin);
    printf(nombre);
}
```

a) ¿Dónde se introduce el problema de seguridad?
b) ¿Qué datos del programa (registros, variables, direcciones de memoria) pueden ser controladas por el usuario? (**Ayuda**: lea con mucho detalle `man printf` y permita sorprenderse)
c) ¿Es posible sobreescribir la dirección de retorno a la que vuelve alguna de las llamadas a la función `printf`?
d) ¿Se soluciona el problema de seguridad si, luego del segundo `printf`, se coloca una llamada a `exit(0)`?

Ejercicio 12

Teniendo en cuenta el ejercicio anterior, considerar el siguiente código:

```
void saludo(void) {
    char nombre[80];
    printf("Ingrese su nombre: ");
    gets(nombre);
    printf(nombre);
    exit(0);
}
```

Dado que la función `saludo` nunca retorna y, por ende, no interesa el valor de su *return address*, ¿es segura la segunda llamada a `printf`?

Ejercicio 13

Versiones de la popular biblioteca de encriptación OpenSSL anteriores a la 0.9.8d tenían un problema de seguridad en la función `SSL_get_shared_ciphers`. A continuación se encuentra una versión simplificada de la función en cuestión. La idea de la función es concatenar todos los nombres que vienen en `ciphers[]` colocando un carácter "`:`" entre ellos.

```
char *SSL_get_shared_ciphers(
    char* ciphers[],
    int cant_chipers,
    char *buf,
    int len
) {
    char *p;
    const char *cp;
    int i;

    if (len < 2)
        return(NULL);
```

```
    p=buf;

    for (i=0; i<cant_ciphers; i++) {
        /* Decrement for either the ':' or a '\0' */
        len--;

        for (cp=ciphers[i]; *cp; ) {
            if (len-- == 0) {
                *p='\0';
                return(buf);
            }
            else
                *(p++)= *(cp++);
        }
        *(p++)=':';
    }
    p[-1]='\0';
    return(buf);
}
```

Dado que el usuario controla el contenido de `ciphers` y `cant_ciphers`, siendo el primero un *array* válido de `cant_ciphers` posiciones definidas donde cada una apunta a una cadena válida de a lo sumo 100 caracteres y *al menos* un carácter, y que `buf` es un buffer de tamaño `len` que no es controlado por el usuario:

a) ¿Es posible escribir información del usuario más allá del límite de tamaño de `buf`, a pesar de todos los chequeos de longitud? Explicar cómo

b) En la versión 0.9.8d corrigieron este problema cambiando la línea "`if (len-- == 0)`" por "`if (len-- <= 0)`". Sin embargo, dos versiones después encontraron que esta correción aún permitía sobreescribir un valor 0 un *byte* después de `buf`.

   Si bien poder escribir un *byte* en 0 luego de un buffer no parece permitir tomar control del programa, también es considerado un problema de seguridad. ¿Qué riesgos podría correr un servidor que utiliza esta biblioteca para conexiones encriptadas por Internet?

Ejercicio 14

En algunas combinaciones de sistema operativo y compilador, el siguiente código permite al usuario tomar control de la ejecución del programa:

```
void leo_y_salgo(void) {
    char leo[80];
    gets(leo)
    exit(1);
}
```

Dado que al regresar de la función `gets` el programa termina la ejecución ignorando el valor de retorno de la función `leo_y_salgo`, para tomar control del programa se debe evitar volver de esta función.

Sabiendo que en estos sistemas al inicio del *stack* se almacena la dirección de los distintos *handlers* de excepciones del proceso (división por cero, error de punto flotante, etc.), explique cómo puede tomar control de la ejecución sin regresar de la función `gets`.