Unidad 6
Desarrollo seguro

Procesos de desarrollo
Separación de tareas

Separación física y lógica de ambientes
Despersonalización de datos

Control de versiones

- Gestión de cambios al código fuente
- Registro histórico de acciones realizadas
- Responsables
- Ej: CVS, Subversion, SourceSafe, TFS, GIT

Ciclo de desarrollo de software

- ***Arquitectura/Diseño***
  - Cuando estamos pensando la aplicación
- ***Implementación***
  - Cuando estamos escribiendo el código de la aplicación.
- ***Operaciones***
  - Cuando la aplicación se encuentra en producción.

Arquitectura de seguridad

- **Definición:**

  Conjunto de principios de diseño de alto nivel y de decisiones que permiten que un programador diga “sí” con confianza y “no” con certeza.

  – Ej. saber que al agregar un componente no se comprometerá la seguridad del sistema

- **Propiedades:**
  – Sirve como marco para el diseño seguro
  – Debe ajustarse a un requerimiento de seguridad dado
  – Se corresponde con un conjunto de documentos de diseño.

Contenidos del documento de arquitectura

- Organización de la aplicación
- Estrategia de cambios
- Decisiones sobre qué se compra y qué se desarrolla.
- Estructuras de datos principales
- Algoritmos claves
- Objetos principales
- Funcionalidad genérica
- Métodos para procesar los errores (corrección o detección)
- Tolerancia a fallas
- ...

Principios de diseño

- **En septiembre de 1975 Saltzer y Schroeder describen ocho principios para el diseño y la implementación de mecanismos de seguridad.**

  Los principios se basan en las ideas de simplicidad y de restricción.

J. Saltzer and M. Schroeder , "The Protection of Information in Computer Systems," Proceedings of the IEEE 63 (9), pp. 12781308 (Sep. 1975).

Principios de diseño

- **Simplicidad (cuando no es posible complejidad mínima)**
  - Los diseños y los mecanismos son mas fáciles de entender.
  - Menos cosas pueden ir mal (menos interacción).
  - Menor posibilidad de inconsistencias entre una política y otras.
- **Restricciones**
  - Minimizar los accesos (solo a lo que necesito).
  - Minimizar las comunicaciones (solo cuando es necesario).

Menor Privilegio

- **Se deben asignar a un sujeto solo aquellos privilegios necesarios para realizar su tarea.**
  - Los privilegios se asignan en base a la función, no a la identidad.
  - Los derechos se agregan cuando se necesitan y se descartan luego de usarlos.
  - Dominio de protección mínimo.

Defaults a prueba de fallo

- **A menos que a un sujeto se le haya otorgado explícitamente el acceso a un objeto, se le debe negar el acceso al mismo.**
  - La acción por defecto es negar el acceso.
  - Si una acción falla, el sistema es tan seguro como cuando la acción comenzó.

Economía de mecanismo

- **Los mecanismos de seguridad deben ser tan simples como sea posible.**
  - Principio KISS (“Keep It Simple, Silly”)
  - Simple significa que hay menos cosas que pueden ir mal
    - Cuando los errores ocurren son más fáciles de entender y arreglar
  - Interfaces e interacciones
    - ¡ Problemas al asumir cosas !

Mediación completa

- **Todos los accesos a los objetos deben ser chequeados para asegurar que los mismos se encuentran permitidos.**
  - Chequear todos los accesos
  - Por lo general se hace solo la primera vez.
  - Si mas tarde los permisos cambian, es posible obtener un acceso no autorizado.

Mediación completa

- **Ej. acceso a archivos en UNIX**

  Cuando en UNIX un proceso trata de leer un archivo, el SO determina si el proceso puede realizar la operación.

  Si es asi, le asigna un descriptor que incluye el acceso permitido, cada vez que el proceso quiere leer el archivo, presenta el descriptor al kernel y este permite el acceso.

  Si el dueño del archivo deshabilita el permiso de lectura luego de que el descriptor fue creado, el kernel al recibir ese descriptor va a permitir el acceso.

  Este esquema viola el principio de mediación completa ya que el segundo acceso no es chequeado, se utiliza un valor cacheado, y por lo tanto la denegación de acceso por parte del dueño del archivo no tiene efecto.

Diseño abierto

- **La seguridad de un mecanismo no debe depender del secreto de su diseño o implementación.**
  - ¿ Es cierto que todo el código fuente debe ser público ?
  - ≠ Seguridad por oscuridad
  - No se aplica a información tal como contraseñas y claves criptográficas.


SUBE
PRESIDENCIA
BANCO DE LA NACION ARGENTINA

Separación de privilegios

- **Un sistema no debe conceder un permiso basándose en una sola condición.**
  - Separación de responsabilidades (ej. Firma de cheques)
  - Defensa en profundidad (varias condiciones se deben cumplir)

Minimizar mecanismos comunes

- **El mecanismo para acceder a los recursos no debe ser compartido.**
  - La información que fluye por canales compartidos es vulnerable.

Aceptabilidad psicológica

- **Los mecanismos de seguridad no deben agregar dificultad al acceso a un recurso.**
  - Se debe ocultar la complejidad que el mecanismo agrega.
  - Facilidad de instalación, configuración y uso
  - Aquí el factor humano es crítico
- **En la práctica se sabe que un mecanismo de seguridad va a agregar una carga extra, pero que la misma debería ser mínima y razonable.**

Puntos claves

- **Los principios de diseño seguro son la base de todos los mecanismos relacionados con seguridad.**
- **Se requiere:**
  - Buena comprensión del objetivo de un mecanismo y del entorno en el que se va a utilizar.
  - Cuidadoso análisis y diseño.
  - Cuidadosa implementación.

Ataques y defensas

- **Los ataques se pueden dividir en tres categorías de acuerdo a la etapa del desarrollo con la que la vulnerabilidad se encuentra relacionada:**
  - Arquitectura/Diseño
    - Cuando se pensó la aplicación.
  - Implementación
    - Cuando se codificó la aplicación.
  - Operación
    - Luego de que la aplicación se encuentra en producción.
    - Los ataques por lo general tienen lugar cuando la aplicación se está ejecutando.

- **Ganar u otorgar confianza, pero nunca darla por sentada**
- **Utilizar un mecanismo de autenticación que no pueda ser eludido ni alterado**
- **Autorizar después de autenticar**
- **Separar estrictamente datos e instrucciones de control**
- **Validar todos los datos explícitamente**
- **Utilizar Criptografía correctamente**
- **Identificar datos sensibles, y como se los debería gestionar**
- **Considerar siempre a los usuarios**
- **La integración de componentes cambia la superficie de ataque**
- **Considerar cambios futuros en objetos y actores**

Fuente: http://cybersecurity.ieee.org/center-for-secure-design/

Ejemplos de problemas en cada etapa

Arquitectura/Diseño

- **Como regla general los problemas más difíciles de solucionar son aquellos que resultan de malas decisiones de diseño.**

- **Fallas importantes en el diseño de protocolos o software**
  - No puede ser resuelto sin afectar versiones anteriores
  - Manteniendo el bug se conserva la compatibilidad hacia atrás

- **Casos típicos**
  - Autenticación en texto claro en protocolos (telnet, ftp)
    - Las credenciales pueden ser sniffeadas por un intruso
  - Dependencia de dirección IP para la autenticación
    - El atacante puede falsificar direcciones IP

TCP/IP

- **TCP/IP tiene problemas de diseño, debido a que no se lo pensó para un ambiente hostil como la Internet Pública de hoy en día.**

- **Problemas:**
  - Sniffing
  - Syn Flooding
  - MiTM
  - Replay
  - Session Hijack
  - Session Termination

Caso de ejemplo: redes wireless

- **Requerimiento de seguridad para IEEE 802.11:**
- **Proveer seguridad “equivalente” a la de un red cableada, de tal manera que solo los dispositivos y usuarios autorizados puedan enviar y recibir paquetes de datos.**
- **Para ello el standard define los mecanismos de cifrado y autenticación WEP (wired equivalence protocol).**
- **WEP es un ejemplo como no diseñar seguridad.**

Problemas

- **WEP se encuentra especificado como opcional y por lo tanto los fabricantes pueden vender access points con WEP deshabilitado por defecto.**

- **Existen muchos AP con todo tipo de cifrado deshabilitado, para simplificar el acceso a los usuarios.**

Problemas

- **WEP tiene errores de diseño relativos al uso de cifrado**
- **Paquete 802.11 no cifrado**

  [header] [data body]
- **Paquete 802.11 cifrado**

  [header] [24 bit IV] [encrypted body] [encrypted 32 bit CRC]

  Clave de cifrado: 104 bits + [24 bit IV] = 128 bits

  Algoritmo de cifrado: RC4
- **Autenticación**

  AP --- R (128 bits) ---> cliente

  AP <--- WEP(R) ------ cliente

Problemas

- **Generación de claves débil**
  - Se conocen 24 bits de la clave de cifrado RC4, los atacantes pueden recuperar los 128 bis luego de observar entre 5 y 6 millones de paquetes cifrados.
- **IV es muy pequeño**
  - Es una forma de one-time-pad, que probablemente sería seguro si cada pad se utilizara una solo vez. Dado que solo se utilizan 24 bits cada pad puede ser reutilizado frecuentemente.
- **CRC-32**
  - La función de integridad, CRC-32, no es criptográficamente fuerte. Un atacante puede cambiar bits del cuerpo cifrado del mensaje sin alterar el CRC-32.
- **La autenticación permite obtener información al atacante**
  - Si un atacante observa una autenticación exitosa de un cliente puede obtener un par texto en claro, texto cifrado.

Implementación

- **En general son debilidades más fáciles de entender y solucionar que los errores de diseño.**

- **Error común:**
  - Hacer suposiciones sobre el ambiente del programa.
    - Ej. Las cadenas de caracteres de entrada serán cortas y en ASCII imprimible
  - Pero usuarios maliciosos pueden introducir lo que ellos deseen.

- **La entrada puede venir desde:**
  - Variables de ambiente.
  - Entrada del programa (local o en red).
  - Otras fuentes.

Problemas de validación de datos de entrada

Se produce cuando las aplicaciones aceptan entradas sin chequear de manera correcta los datos de entrada.

Ej.

```
char buf[1024];
snprintf(buf, "system lpr -P %s", user_input, sizeof(buf)-1);
system(buf);
```

Entrada: **FRED; xterm&**

Mal uso de privilegios


```
[plaguez@plaguez plaguez]$ ls -al /etc/shadow
-rw----   1 root     bin 1039 Aug 21 20:12 /etc/shadow

 [plaguez@plaguez bin]$ ID
uid=502(plaguez) gid=500(users) groups=500(users)

[plaguez@plaguez plaguez]$ cd /usr/X11R6/bin
[plaguez@plaguez bin]$ ./XF86_SVGA -config /etc/shadow
Unrecognized option: root:qEXaUxSeQ45ls:10171:-1:-1:-1:-1:-1:-1
use: X [:<display>] [option]
-a #                   mouse acceleration (pixels)
-ac                    disable access control restrictions
-audit int            set audit trail level
-auth file            select authorization file
bc                     enable bug compatibility
-bs                    disable any backing store support
-c                     turns off key-click
```

The Apple goto fail vulnerability: lessons learned

David A. Wheeler

2017-01-27 (original 2014-11-23)

This paper identifies lessons that we *should* learn from the Apple "goto fail" vulnerability. It first starts with some background, discusses the misplaced blame on the goto statement, focuses on identifying what could have countered this, briefly discusses the Heartbleed countermeasures from my separate paper on Heartbleed, and ends with conclusions. Some of these points have been made elsewhere, but this tries to merge them together in one place. Others have not been noted elsewhere, to my knowledge. For example, gcc quietly dropped its support for detecting dead code that can lead to this vulnerability, and detecting duplicate lines in source code might also be an additional useful countermeasure. This paper is part of the essay suite Learning from Disaster.

1. Background

On 2014-02-21 Apple released a security update for its implementation of SSL/TLS in many versions of its operating system. The vulnerability is formally named CVE-2014-1266, but informally it's often called the Apple "goto fail" vulnerability (or "goto fail goto fail" vulnerability).

The essence of the problem is straightforward. The code included these lines [Apple2014] in function SSLVerifySignedServerKeyExchange:

```
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
  goto fail;
  goto fail;
... other checks ...
fail:
  ... buffer frees (cleanups) ...
  return err;
```

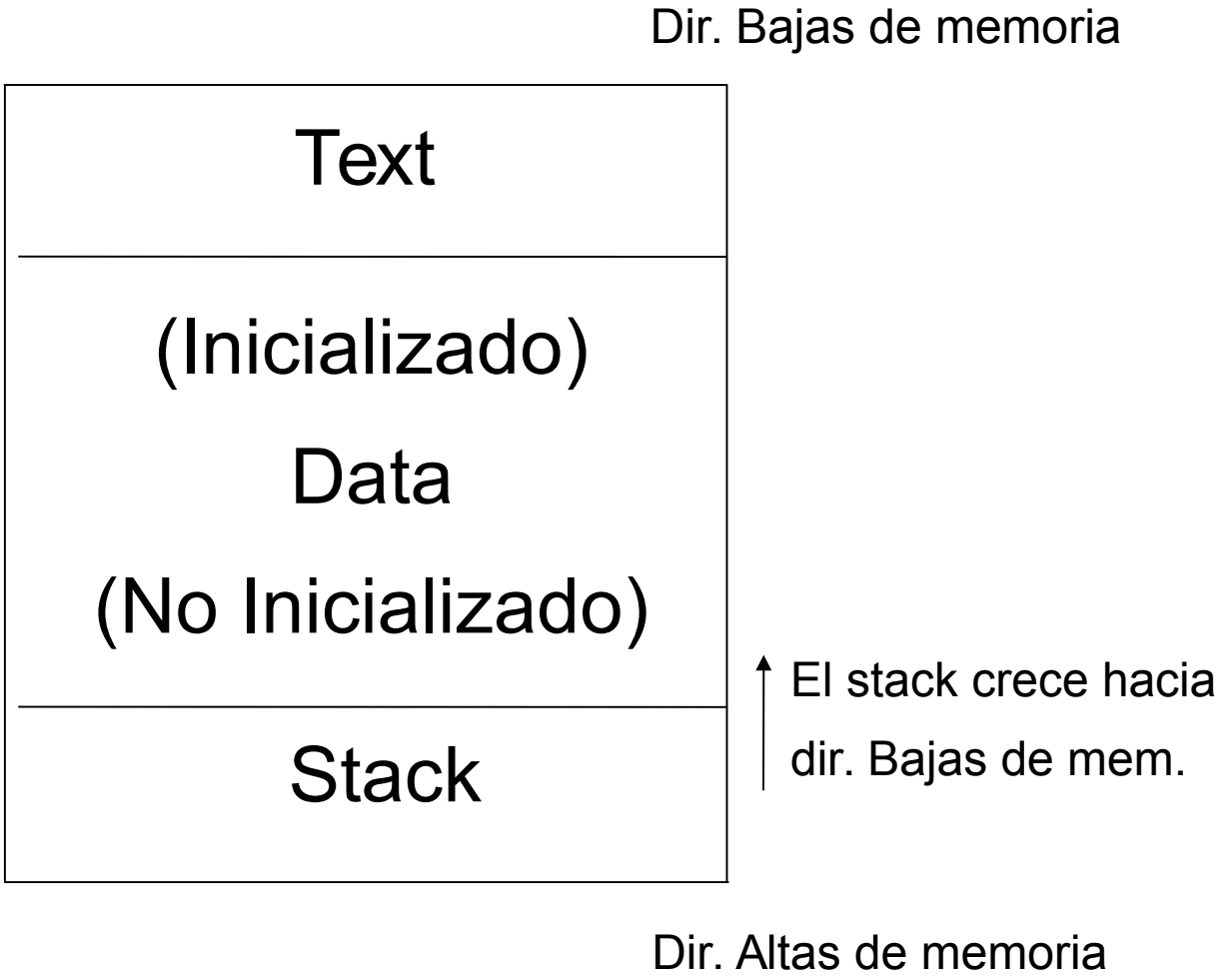
The problem was the second (duplicate) "goto fail". The indentation here is misleading; since there are no curly braces after the "if" statement, the second "goto fail" is *always* executed. In context, that meant that vital signature checking code was skipped, so both bad and good signatures would be accepted. The extraneous "goto" caused the function to return 0 ("no error") when the rest of the checking was skipped; as a result, invalid certificates were quietly accepted as valid.

Buffer overflow

- **Error: Asumir que un buffer de tamaño fijo es lo suficientemente grande como para almacenar toda la entrada**
  - El lenguaje C no tiene incorporado verificaciones de límite de buffer.
- **Pregunta**
  - ¿Qué tal si hay mas datos que espacio en el buffer?
- **Respuesta**
  - Los datos extra son escritos en memoria, después del buffer

- **Ejemplo: Stack smashing**

Buffer overflow



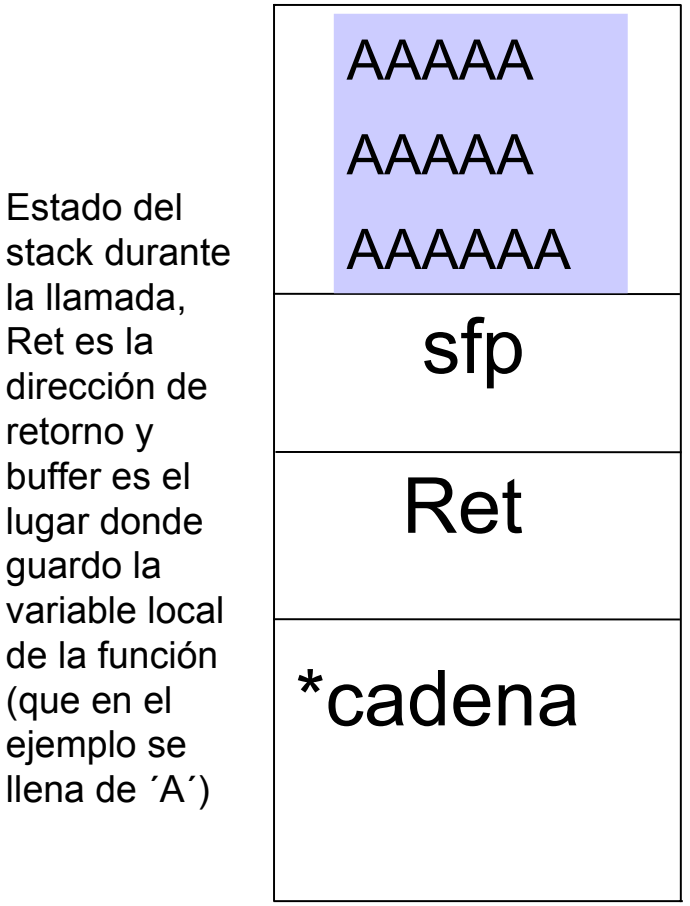
Buffer overflow

```
Void function(char *str) {
  char buffer[16];
  strcpy(buffer,str);
}

Void main () {
  char cadena[16];
  int i;

  for (i=0; i < 16; i++)
   cadena[i] = ´A´;
  function(cadena);
}
```

Dir. Bajas de memoria



Estado del stack durante la llamada, Ret es la dirección de retorno y buffer es el lugar donde guardo la variable local de la función (que en el ejemplo se llena de ´A´)

sfp = stack frame pointer

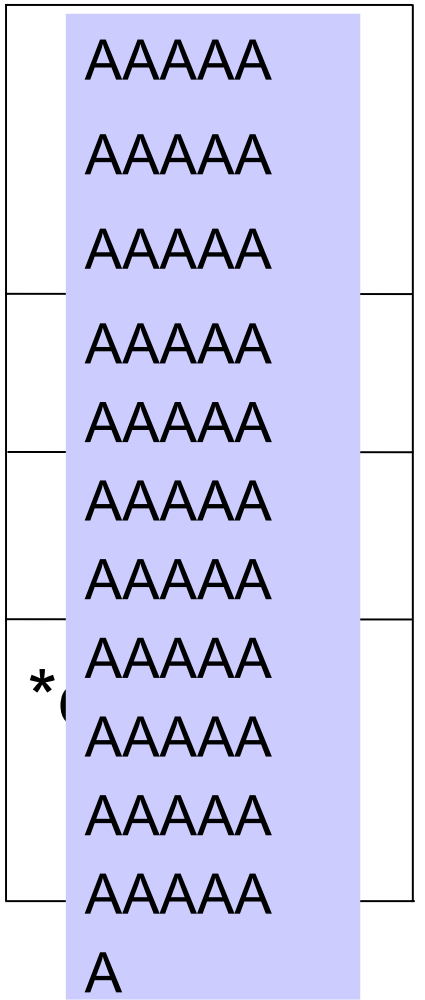
Buffer overflow

```
Void function(char *str) {
  char buffer[16];
  strcpy(buffer,str);
}

Void main () {
  char cadena[256];
  int i;

  for (i=0; i < 256; i++)
   cadena[i] = ´A´;
  function(cadena);
}
```

Idem Anterior, pero se produce un buffer overflow. Se escribe mas allá del tamaño de buffer, pudiendo cambiar la dirección de retorno.



Shellcode

- **Un Shellcode es un pequeño código usado como "payload" en la explotación de una vulnerabilidad de software.**

- **Se llama así porque en muchos casos ejecuta un intérprete de comandos desde el cual el atacante puede controlar el equipo comprometido.**

- **Hoy en día, pueden realizan acciones varias como crear usuarios, descargar un binario y ejecutarlo, crear canales inversos de comunicación, etc.**

Ejemplo Shellcode


```
#include <stdio.h>
#include <stdlib.h>
void main() {

char *name[2];
name[0] = "/bin/sh";
    name[1] = NULL;
    execve(name[0], name,
    NULL);
    exit(0);
}
```

```
char shellcode[] =
    "\xeb\x2a\x5e\x89\x76\x08\xc
    6\x46\x07\x00\xc7\x46\x0c\x0
    0\x00\x00"
    "\x00\xb8\x0b\x00\x00\x00\x8
    9\xf3\x8d\x4e\x08\x8d\x56\x0c
    \xcd\x80"
    "\xb8\x01\x00\x00\x00\xbb\x0
    0\x00\x00\x00\xcd\x80\xe8\xd
    1\xff\xff"
    "\xff\x2f\x62\x69\x6e\x2f\x73\x
    68\x00\x89\xec\x5d\xc3";
```

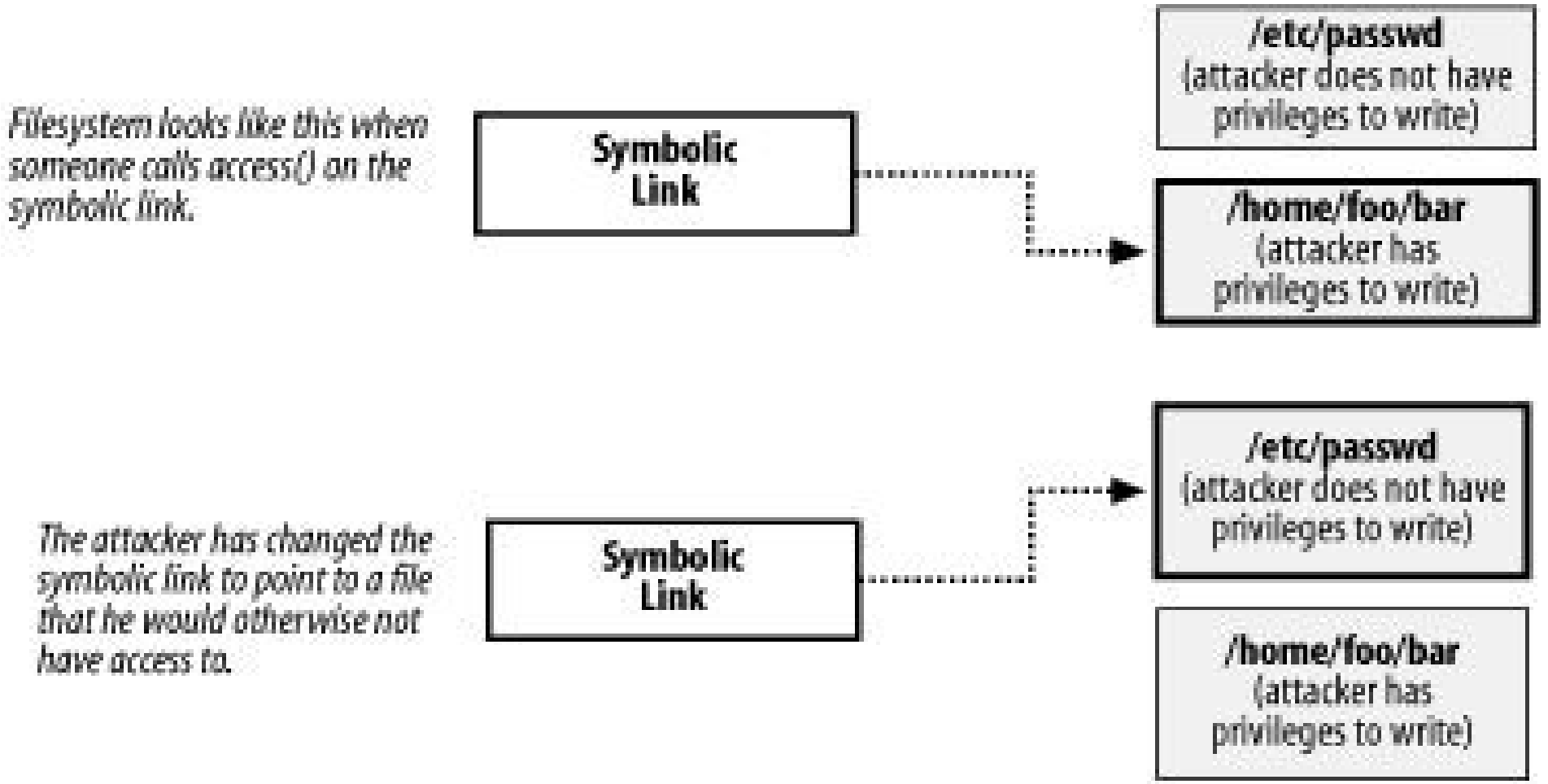
Race Conditions

- **Ejemplo de Race Condition: Time of Check, time of use (TOCTTOU)**

```
if (access(file, R_OK) != 0)
{ exit(1); }
fd = open(file, O_RDONLY);
// do something with fd...
```

- **Supongamos que este es un programa con setUID.**
- **La función access() chequea si el usuario que ejecuta el programa setuid tiene acceso al archivo (chequea el real userid, no el effective userid)**

Race Conditions



Format string

- **No hay problema si el formato de cadena se especifica en el código**
  - ej. printf(“%s = %d\n”, variable, value);
- **Problema si el formato de cadena puede ser especificado por el intruso**
  - ej. printf(error);
  - ¿Qué tal si el error = “<código malicioso>%x%x.. %n..%n..%n.. %n” ?
- **Varias funciones utilizan formatos de cadena, p.e.**
  - printf, fprintf, sprintf, snprintf, vprintf, syslog, setproctitle, …
  - Error común: syslog(mensaje) donde el mensaje viene de la entrada del usuario.

Format string (ejemplo)


```
#include <stdio.h>
int main(int argc, char* argv[]) {
   if(argc > 1)
   printf(argv[1]);
   return 0;
}
```

parameter	output	passed as
%d	decimal (int)	value
%u	unsigned decimal (unsigned int)	value
%x	hexadecimal (unsigned int)	value
%s	string ((const) (unsigned) char *)	reference
%n	number of bytes written so far, (* int)	reference

```
/formatstr  "%x %x"
12ffc0 4011e5
```

```
unsigned int bytes;
printf("%s%n\n", argv[1], &bytes);
printf("Your input was %d characters long\n, bytes");
```

```
/formatstr2  "Some random input"
Some random input
Your input was 17 characters long
```

Format string

DEPARTAMENTO DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

- **Los mismos pasos que los buffer overflow**
  - Poner código en la pila (binario hostil).
  - Moverse a través de la pila (%x muestra variables, o la pila).

```
printf ("%08x.%08x.%08x.%08x.%08x\n");
40012980.080628c4.bffff7a4.00000005.08059c04
printf ("\x10\x01\x48\x08_%08x.%08x.%08x.%08x.%08x.|%s|");
Muestra un string desde la dirección 0x08480110 hasta
encontrar un NULL
```

  - Sobrescribir la dirección de retorno (%n establece una variable, o byte en la pila).

```
"\xc0\xc8\xff\xbf_%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%n"
Re-escribe 4 bytes en la dirección 0xbfffc8c0
```

Format string

- **Como funciona el ataque**
  - El atacante utiliza varios %x para moverse a la dirección de retorno.
  - Entonces el atacante sobrescribe la dirección de retorno con %n
    - Escribe n bytes seguidos por %n para escribir el byte 1 de la nueva dirección de retorno.
  - El resto del buffer es llenado con el código del atacante.
  - En el regreso, la ejecución salta al código del atacante.
- **Puede sobrescribir otras direcciones, ej. process user id**

Back door

El atacante (programador) incluye funcionalidad adicional en la aplicación que hace que mas tarde sea posible saltear los mecanismos de control de acceso, por ejemplo utilizando un nombre usuario especial.

- **Posibles soluciones:**

  Adoptar procedimientos de Quality Assurance para chequear que el código no tenga back doors.

Operación

- **Estos ataques ocurren como resultado de decisiones que se toman luego del desarrollo de la aplicación, cuando la misma es puesta en producción.**

- **Problemas:**
  - Cuentas con claves por defecto
  - Instalación por defecto
  - DoS
  - Política pobre de actualizaciones de seguridad