Resumen Teórica 14 : Desarrollo seguro

Tomás F. Melli

September 2025

Índice

1 Introducción **2**
1.1 Separación de tareas . . . 2
1.1.1 Ambiente de Diseño y Programación . . . 2
1.1.2 Ambiente de Testing . . . 2
1.1.3 Ambiente de Producción . . . 3
1.1.4 Separación física y lógica de ambientes . . . 3
1.1.5 Despersonalización de datos . . . 3
1.2 Control de Versiones y Procesos de Desarrollo Seguro . . . 3

2 Ciclo de desarrollo de software **4**
2.1 Arquitectura de seguridad . . . 4
2.1.1 Contenidos del documento de arquitectura . . . 4

3 Principios de Diseño Seguro **5**
3.1 Simplicidad y Restricciones . . . 5
3.2 Principio del Menor Privilegio . . . 5
3.3 Defaults a Prueba de Fallo . . . 5
3.4 Economía de Mecanismo . . . 5
3.5 Mediación Completa . . . 5
3.6 Diseño Abierto . . . 6
3.7 Separación de Privilegios . . . 6
3.8 Minimización de Mecanismos Comunes . . . 6
3.9 Aceptabilidad Psicológica . . . 6

4 Puntos Claves **7**
4.1 Ataques y Defensas . . . 7
4.1.1 Avoiding the Top 10 Software Security Design Flaws . . . 7

5 Ejemplos de Problemas en Cada Etapa del Desarrollo Seguro **7**
5.1 Arquitectura / Diseño . . . 7
5.1.1 Problemas de TCP/IP . . . 8
5.1.2 Caso de Ejemplo: Redes Wireless (IEEE 802.11) . . . 8
5.2 Implementación . . . 8
5.2.1 Problemas de validación de datos de entrada . . . 9
5.2.2 Mal uso de privilegios . . . 9
5.2.3 Caso de Estudio: Apple `goto fail` Vulnerability . . . 10
5.2.4 Buffer Overflow . . . 11
5.2.5 Shellcode . . . 12
5.2.6 Race Conditions — TOCTTOU (Time Of Check, Time Of Use) . . . 14
5.2.7 Vulnerabilidades por Format String . . . 16
5.2.8 Back Door . . . 18
5.3 Operación . . . 18

1 Introducción

Un proceso de **desarrollo seguro** establece un conjunto ordenado de actividades que aseguran la **calidad y seguridad del software** en todas sus etapas. Incluye políticas, estándares y controles que permiten:

- Incorporar la seguridad desde el diseño.
- Verificar código y dependencias.
- Registrar y auditar cambios.
- Asegurar la trazabilidad de las decisiones y versiones.

El objetivo es reducir vulnerabilidades y evitar errores que comprometan la seguridad del sistema o la información.

1.1 Separación de tareas

La **separación de tareas** (Separation of Duties) busca evitar que una sola persona concentre funciones críticas dentro del proceso de desarrollo. Cada rol tiene responsabilidades acotadas:

- Los desarrolladores crean y modifican código.
- Los testers verifican funcionalidad y seguridad.
- Los administradores gestionan la infraestructura y despliegues.
- El área de seguridad revisa y aprueba cambios sensibles.

Este principio minimiza el riesgo de errores no detectados o acciones malintencionadas.

1.1.1 Ambiente de Diseño y Programación

Es el entorno donde los desarrolladores **diseñan, programan y realizan pruebas unitarias**. Debe:

- Estar aislado del entorno de producción.
- Usar datos ficticios o despersonalizados.
- Incluir herramientas de control de versiones y análisis de seguridad.
- Permitir revisiones y pruebas sin afectar sistemas reales.

Acá se aplica la mayor parte de las prácticas de codificación segura.

1.1.2 Ambiente de Testing

El ambiente de testing replica las condiciones de producción para **evaluar el comportamiento del sistema antes de su liberación**. Sus características principales son:

- Contener versiones casi finales del software.
- Ejecutar pruebas funcionales, de integración y de seguridad.
- Utilizar datos anonimizados.
- Estar bajo control de un equipo independiente del desarrollo.

Su finalidad es detectar defectos antes del paso a producción.

1.1.3 Ambiente de Producción

Es el entorno real donde la aplicación está en operación y acceden los usuarios finales. Debe contar con:

- Controles estrictos de acceso y cambio.
- Monitoreo continuo y registro de incidentes.
- Procedimientos formales de despliegue y rollback.
- Aprobación previa de todas las modificaciones.

La estabilidad y disponibilidad son prioritarias, por lo que se prohíben desarrollos o pruebas directas en este ambiente.

1.1.4 Separación física y lógica de ambientes

La separación entre entornos (desarrollo, testing y producción) debe ser **técnica y organizativa**:

- **Separación física:** servidores o máquinas virtuales distintas, redes independientes y accesos diferenciados.
- **Separación lógica:** credenciales, bases de datos y configuraciones específicas para cada ambiente.

Esta separación impide que un error o intrusión en desarrollo comprometa los datos o sistemas de producción.

1.1.5 Despersonalización de datos

La **despersonalización de datos** consiste en reemplazar información sensible de personas por valores ficticios o anonimizados. Se utiliza en ambientes de desarrollo y testing para proteger la privacidad y cumplir con leyes de protección de datos.

Ejemplos:

- Sustituir nombres reales por alias.
- Eliminar identificadores únicos (DNI, correo).
- Modificar datos sensibles manteniendo el formato.

Esto permite realizar pruebas realistas sin exponer información confidencial.

1.2 Control de Versiones y Procesos de Desarrollo Seguro

El control de versiones es un elemento esencial dentro de los procesos de desarrollo seguro de software. Permite gestionar los cambios al código fuente, registrar el historial de modificaciones, identificar responsables y garantizar la trazabilidad de las acciones realizadas durante todo el ciclo de vida del software.

El uso de herramientas de control de versiones facilita la separación de entornos (diseño, programación, testing y producción), la revisión de código, y la implementación de estrategias seguras de despliegue. Además, contribuye a la **integridad del código**, asegurando que los cambios sean auditables y reversibles en caso de incidentes o fallos.

Ejemplos de herramientas

- **CVS (Concurrent Versions System):** uno de los primeros sistemas de control de versiones, basado en archivos de texto y orientado al trabajo colaborativo en proyectos centralizados.
- **Subversion (SVN):** mejora sobre CVS que introduce commits atómicos y una gestión más robusta de ramas y fusiones.
- **SourceSafe:** herramienta de Microsoft orientada a entornos empresariales Windows, con control centralizado del repositorio.
- **TFS (Team Foundation Server):** solución integral de Microsoft que combina control de versiones, seguimiento de tareas y automatización de procesos de desarrollo.
- **GIT:** sistema distribuido ampliamente adoptado, que permite desarrollo en paralelo, gestión de ramas eficiente, auditoría completa y una gran flexibilidad para la integración continua y la colaboración.

2 Ciclo de desarrollo de software

El ciclo de desarrollo puede dividirse en tres grandes fases:

- **Arquitectura / Diseño:** etapa donde se define la estructura general de la aplicación, sus componentes y las políticas de seguridad que regirán su desarrollo.
- **Implementación:** momento en que se escribe el código. Se aplican prácticas de codificación segura, revisión de código y análisis de vulnerabilidades.
- **Operaciones:** etapa donde la aplicación está en producción. Se monitorea su comportamiento, se aplican actualizaciones seguras y se gestionan incidentes.

2.1 Arquitectura de seguridad

Definición: conjunto de principios y decisiones de diseño que aseguran que las aplicaciones puedan integrar nuevos componentes sin comprometer la seguridad global. Permite que los programadores puedan decir "sí" con confianza y "no" con certeza.

Propiedades:

- Proporciona un marco de referencia para el diseño seguro.
- Se ajusta a los requerimientos de seguridad establecidos.
- Se traduce en documentos técnicos de diseño y decisión.

2.1.1 Contenidos del documento de arquitectura

El documento de arquitectura de seguridad debe incluir, al menos, los siguientes elementos. Cada uno contribuye a garantizar que el diseño del sistema sea seguro, consistente y mantenible a lo largo de su ciclo de vida:

- **Organización de la aplicación:** Describe la estructura general del sistema, los módulos principales y sus interrelaciones. Define cómo se distribuyen las responsabilidades entre componentes.
- **Estrategia de cambios y mantenimiento:** Establece cómo se gestionarán las modificaciones futuras en el sistema, incluyendo procedimientos de actualización, control de versiones y revisión de seguridad.
- **Decisiones sobre qué se compra y qué se desarrolla internamente:** Define los componentes o servicios que se adquirirán de terceros y los que serán desarrollados por el equipo interno, considerando implicancias de seguridad y soporte.
- **Estructuras de datos principales:** Documenta las estructuras de información críticas, su formato, relaciones y mecanismos de protección de la integridad y confidencialidad de los datos.
- **Algoritmos clave:** Especifica los algoritmos fundamentales del sistema, en particular los relacionados con seguridad, cifrado, autenticación, validación o control de acceso.
- **Objetos y componentes principales:** Detalla las clases, módulos o servicios esenciales que conforman la aplicación y cómo interactúan entre sí para cumplir los requisitos funcionales y de seguridad.
- **Funcionalidades genéricas del sistema:** Incluye aquellas funciones transversales, como auditoría, registro de eventos, gestión de usuarios, o comunicación entre módulos, que deben seguir políticas de seguridad consistentes.
- **Mecanismos de manejo y detección de errores:** Describe cómo el sistema responde ante fallos, excepciones o condiciones inesperadas, asegurando que dichos mecanismos no revelen información sensible ni comprometan la disponibilidad.
- **Estrategias de tolerancia a fallas:** Define los métodos para mantener la continuidad del servicio ante errores de hardware, software o ataques, como redundancia, recuperación automática o replicación de datos.
- **Criterios de seguridad y control de acceso:** Especifica las políticas de autenticación, autorización y trazabilidad, así como los niveles de acceso permitidos para cada rol o componente dentro del sistema.

3 Principios de Diseño Seguro

En septiembre de 1975, Jerome Saltzer y Michael Schroeder presentaron en su trabajo *"The Protection of Information in Computer Systems"* una serie de principios fundamentales para el diseño e implementación de mecanismos de seguridad en sistemas informáticos. Estos principios se basan en dos ideas clave: la **simplicidad** y la **restricción**. Su propósito es servir como guía general para el diseño de sistemas que sean seguros por construcción, reduciendo tanto la posibilidad de errores como el impacto de vulnerabilidades inevitables.

3.1 Simplicidad y Restricciones

- **Simplicidad:** Cuando no es posible lograr una complejidad mínima, se debe procurar que los mecanismos y estructuras de seguridad sean lo más simples posible. Los diseños simples son más fáciles de entender, analizar y mantener. Además, la simplicidad reduce las interacciones no deseadas y las inconsistencias entre políticas de seguridad.
- **Restricciones:** Este principio establece que se deben minimizar los accesos y las comunicaciones al mínimo necesario. Es decir, los procesos, usuarios o componentes del sistema solo deben acceder a los recursos estrictamente requeridos y comunicarse únicamente cuando sea imprescindible. Esta limitación reduce la superficie de ataque y mejora la contención de fallos o intrusiones.

3.2 Principio del Menor Privilegio

A cada sujeto (usuario, proceso o componente) se le deben asignar únicamente los privilegios estrictamente necesarios para realizar su tarea. Los privilegios deben basarse en la *función* desempeñada y no en la identidad del sujeto. Además, deben otorgarse dinámicamente cuando sean requeridos y revocarse una vez finalizada la operación.

Esto se conoce como **dominio de protección mínimo**, y su correcta aplicación limita el impacto de errores o compromisos de seguridad, evitando que un fallo local escale a todo el sistema.

3.3 Defaults a Prueba de Fallo

Por defecto, el acceso a los recursos debe ser **denegado**, a menos que exista una autorización explícita. Este principio asegura que el sistema se mantenga en un estado seguro incluso ante fallos o errores de configuración. En otras palabras, si una acción falla, el sistema debe permanecer tan seguro como lo estaba antes de que la acción comenzara.

> *"Fail-safe defaults": lo que no está explícitamente permitido, está prohibido.*

3.4 Economía de Mecanismo

Los mecanismos de seguridad deben ser tan simples como sea posible, siguiendo el principio KISS (*Keep It Simple, Silly*). La simplicidad implica que hay menos elementos que pueden fallar, y cuando ocurren errores, estos son más fáciles de detectar y corregir.

Asimismo, las interfaces y las interacciones entre módulos deben ser claras y consistentes. Los problemas suelen surgir cuando se asumen comportamientos implícitos o no documentados, lo que puede derivar en vulnerabilidades.

3.5 Mediación Completa

Todos los accesos a los objetos deben ser verificados para asegurar que están permitidos. Esto significa que no basta con validar un acceso la primera vez: cada intento de acceso debe ser comprobado en función de las políticas de control actuales. De lo contrario, pueden surgir accesos no autorizados si las condiciones cambian.

- En muchos sistemas, por motivos de eficiencia, el control de acceso se realiza solo una vez y el resultado se almacena en caché.
- Si luego cambian los permisos del recurso, el acceso puede seguir siendo permitido indebidamente.

Ejemplo en UNIX: Cuando un proceso intenta leer un archivo, el sistema operativo determina si tiene permiso y, en caso afirmativo, le asigna un descriptor de archivo que incluye los derechos de acceso. Cada vez que el proceso quiere leer, presenta ese descriptor al kernel. Si el propietario del archivo revoca el permiso de lectura después de que el descriptor fue creado, el kernel seguirá permitiendo el acceso porque se basa en la información cacheada. Este comportamiento **viola el principio de mediación completa**, ya que los accesos posteriores no son revalidados.

3.6 Diseño Abierto

La seguridad de un mecanismo no debe depender del secreto de su diseño o implementación. Este principio sostiene que la fortaleza de un sistema debe basarse en la solidez de sus algoritmos, políticas y controles, no en el hecho de mantener oculto su funcionamiento interno.

- Un sistema seguro debe poder resistir ataques incluso si sus detalles de diseño son públicos.
- Este principio no implica que toda la implementación o código fuente deba ser necesariamente abierto, sino que la seguridad no debe basarse exclusivamente en su ocultamiento.
- Se diferencia del concepto de **"seguridad por oscuridad"**, que es una práctica insegura y poco confiable.
- No aplica a información sensible como contraseñas, claves criptográficas o datos de autenticación, que sí deben mantenerse secretas.

Ejemplo: Los algoritmos criptográficos modernos (como AES o RSA) son públicos y ampliamente analizados, pero su seguridad depende de la dificultad matemática del problema subyacente, no del secreto del algoritmo.

3.7 Separación de Privilegios

Un sistema no debe conceder un permiso basándose en una sola condición. Este principio busca dividir la autoridad o la toma de decisiones en múltiples factores o roles, reduciendo el riesgo de abuso o error.

- **Separación de responsabilidades:** una acción sensible debe requerir la participación o autorización de más de una entidad. Por ejemplo, en una organización financiera, la emisión de un pago puede requerir tanto la creación como la aprobación de la orden por personas diferentes.
- **Defensa en profundidad:** el acceso o la autorización deben depender del cumplimiento de varias condiciones o controles, creando capas sucesivas de protección.

Ejemplo: Para acceder a un sistema crítico se puede requerir autenticación multifactor: una contraseña, un token físico y la validación de un rol autorizado. Solo cumpliendo todas las condiciones se concede el acceso.

3.8 Minimización de Mecanismos Comunes

El mecanismo utilizado para acceder a los recursos no debe ser compartido por múltiples componentes o usuarios de manera indiscriminada. El uso de mecanismos comunes puede crear canales de comunicación no controlados y aumentar la exposición del sistema.

- Los canales compartidos pueden permitir filtrado o fuga de información entre procesos o usuarios con distintos niveles de privilegio.
- Cada mecanismo de acceso debe ser tan específico y aislado como sea posible, reduciendo las dependencias entre componentes.

Ejemplo: Compartir un mismo archivo temporal o una cola de mensajes entre procesos con diferentes permisos puede abrir un canal encubierto por el cual se filtra información sensible.

3.9 Aceptabilidad Psicológica

Los mecanismos de seguridad deben ser razonablemente fáciles de usar, configurar e instalar. Si las medidas de seguridad resultan excesivamente complejas o interfieren con las tareas cotidianas de los usuarios, estos tenderán a evitarlas o desactivarlas, comprometiendo la efectividad global del sistema.

- Se debe ocultar la complejidad técnica siempre que sea posible, proporcionando interfaces simples y comprensibles.
- El factor humano es crítico: un sistema seguro pero impráctico no será utilizado correctamente.
- La carga adicional introducida por los controles de seguridad debe ser mínima y justificada.

Ejemplo: Un sistema de autenticación que obliga a cambiar contraseñas diariamente con requisitos excesivos puede llevar a que los usuarios las anoten o reutilicen, debilitando la seguridad en lugar de fortalecerla.

4 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.

4.1 Ataques y Defensas

Los ataques pueden clasificarse según la etapa del desarrollo en la que la vulnerabilidad se origina:

- **Arquitectura/Diseño:** vulnerabilidades introducidas al concebir la aplicación, por ejemplo, decisiones estructurales inseguras o ausencia de mecanismos de validación.
- **Implementación:** errores cometidos durante la codificación, como buffer overflows, inyecciones o manejo inseguro de datos.
- **Operación:** problemas que surgen una vez que la aplicación está en producción, como configuraciones inseguras, contraseñas débiles o falta de actualización de parches.

En general, los ataques ocurren mientras la aplicación se encuentra en ejecución, aprovechando debilidades de cualquiera de las etapas anteriores.

4.1.1 Avoiding the Top 10 Software Security Design Flaws

Según el *IEEE Center for Secure Design (2014)*, existen diez errores comunes en el diseño de software seguro que deben evitarse:

1. Ganar u otorgar confianza, pero nunca darla por sentada.
2. Utilizar un mecanismo de autenticación que no pueda ser eludido ni alterado.
3. Autorizar después de autenticar.
4. Separar estrictamente los datos de las instrucciones de control.
5. Validar todos los datos de manera explícita.
6. Utilizar la criptografía correctamente.
7. Identificar los datos sensibles y definir cómo deben ser gestionados.
8. Considerar siempre a los usuarios en el diseño (usabilidad y experiencia segura).
9. Tener en cuenta que la integración de componentes modifica la superficie de ataque.
10. Considerar los cambios futuros en objetos y actores del sistema.

5 Ejemplos de Problemas en Cada Etapa del Desarrollo Seguro

Los problemas de seguridad pueden aparecer en diferentes etapas del ciclo de vida del software y los sistemas, siendo generalmente más difíciles de solucionar aquellos originados en la **etapa de arquitectura y diseño**.

5.1 Arquitectura / Diseño

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

- Fallas importantes en el diseño de protocolos o software:
  - No pueden ser resueltas sin afectar versiones anteriores.
  - Mantener el bug puede conservar compatibilidad hacia atrás.
- Casos típicos:
  - Autenticación en texto claro en protocolos como Telnet o FTP, lo que permite que un intruso capture las credenciales (*sniffing*).
  - Dependencia de la dirección IP para la autenticación, lo que permite que un atacante falsifique direcciones IP.

5.1.1 Problemas de TCP/IP

El conjunto de protocolos TCP/IP presenta vulnerabilidades de diseño, dado que no se concibió para un entorno hostil como la Internet pública actual. Algunos de los problemas conocidos son:

- Sniffing (captura de tráfico en texto claro).
- Syn Flooding (ataque de denegación de servicio).
- Man-in-the-Middle (interceptación de comunicaciones).
- Replay (repetición de mensajes válidos).
- Session Hijacking (secuestro de sesiones activas).
- Session Termination (finalización indebida de sesiones).

5.1.2 Caso de Ejemplo: Redes Wireless (IEEE 802.11)

El estándar IEEE 802.11 busca proporcionar seguridad equivalente a la de una red cableada, de manera que solo dispositivos y usuarios autorizados puedan enviar y recibir paquetes de datos. Para ello, definió mecanismos de cifrado y autenticación conocidos como WEP (*Wired Equivalent Protocol*).

Problemas de WEP:

- WEP es especificado como opcional; muchos fabricantes venden puntos de acceso (*Access Points*) con WEP deshabilitado por defecto.
- Muchos AP permiten todo tipo de cifrado deshabilitado para simplificar el acceso a usuarios.
- 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 bits de IV = 128 bits, usando el algoritmo RC4.
  - Autenticación: `AP --- R (128 bits) ---> cliente AP <--- WEP(R) ------ cliente`

Problemas específicos de seguridad en WEP:

- **Generación de claves débil:** los 24 bits del IV permiten a un atacante recuperar la clave de 128 bits observando entre 5 y 6 millones de paquetes cifrados.
- **IV muy pequeño:** cada pad de cifrado (one-time pad) se reutiliza frecuentemente, debilitando la seguridad.
- **CRC-32:** la función de integridad CRC-32 no es criptográficamente segura; un atacante puede alterar bits del cuerpo cifrado sin modificar el CRC.
- **Autenticación insegura:** un atacante que observa una autenticación exitosa puede obtener pares de texto en claro y texto cifrado, facilitando la recuperación de información sensible.

5.2 Implementación

En general, las debilidades introducidas durante la **implementación** son más fáciles de comprender y corregir que los errores de diseño, pero siguen siendo una fuente importante de vulnerabilidades explotables en sistemas reales.

Errores comunes

- **Hacer suposiciones sobre el ambiente del programa.** Por ejemplo asumir que las cadenas de entrada serán cortas y compuestas sólo por caracteres ASCII imprimibles. Usuarios maliciosos pueden enviar cualquier secuencia de bytes — incluyendo bytes no imprimibles, secuencias largas o payloads especialmente construidos.
- **Asumir propiedades del sistema:** formato de rutas, configuración de locale, permisos por defecto, existencia/valor de variables de entorno, o que librerías externas siempre devolverán resultados “correctos”.

Fuentes de entrada La entrada no confiable puede provenir de múltiples canales, por ejemplo:

- Variables de entorno.
- Entrada del programa (stdin), argumentos de línea de comandos o formularios web.
- Tráfico de red (sockets, RPC, APIs).
- Ficheros de configuración, colas de mensajes, o incluso archivos temporales compartidos.

5.2.1 Problemas de validación de datos de entrada

Ocurren cuando las aplicaciones aceptan entradas sin realizar la validación y saneamiento adecuados. Un patrón inseguro habitual es componer comandos del sistema con datos del usuario y pasarlos a funciones como `system()`, lo cual puede derivar en *command injection*.

Ejemplo (comando vulnerable en C):

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

En este ejemplo:

- `user_input` no se valida ni se escapa.
- Si el atacante envía: `FRED; xterm&` se ejecutará `xterm` además del comando esperado (o cualquier otro payload).
- Uso de `system()` es especialmente peligroso porque invoca el intérprete de comandos del sistema.

Mitigaciones recomendadas:

- Evitar `system()`; usar llamadas seguras con vectores de argumentos (`execve`, `posix_spawn`) o APIs de biblioteca que no invocan el shell.
- Validar estrictamente la entrada (lista blanca) y/o escapar/normalizar datos antes de su uso.
- Limitar el tamaño de buffers y comprobar longitudes (usar funciones seguras con control de tamaño).
- Emplear tests (unitarios, fuzzing) que forcen entradas atípicas.

5.2.2 Mal uso de privilegios

Otra fuente frecuente de vulnerabilidades es el mal manejo de privilegios: procesos que ejecutan operaciones con más derechos de los necesarios, o que mantienen derechos elevados al invocar componentes inseguros.

Ejemplo ilustrativo (salida de consola ficticia):

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

[plaguez@plaguez plaguez]$ 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]
...
```

En el ejemplo se ilustra cómo una aplicación con opciones de configuración mal diseñadas (o utilidades con privilegios elevados) puede leer o revelar ficheros sensibles (`/etc/shadow`) si se le pasan rutas arbitrarias como parámetros. Esto puede ocurrir por:

- Binaries con bits `setuid` o servicios que ejecutan código con privilegios de root.
- Validación insuficiente de argumentos o archivos de configuración.
- Funciones que imprimen o procesan contenidos sensibles sin restricciones.

Mitigaciones recomendadas:

- Aplicar el principio de **menor privilegio**: ejecutar procesos con los permisos mínimos necesarios.
- Evitar el uso de programas `setuid` siempre que sea posible; si se usan, revisar cuidadosamente su entrada y realizar un control estricto de las acciones permitidas.
- Separar responsabilidades: componentes que manejan datos sensibles deben estar aislados y tener controles de acceso estrictos.
- Validar y canonicalizar rutas y parámetros antes de abrir ficheros.
- Revisiones de código y auditorías de binarios con privilegios elevados.

5.2.3 Caso de Estudio: Apple `goto fail` Vulnerability

En febrero de 2014 se descubrió un bug crítico en la implementación de SSL/TLS de Apple (`SecureTransport`) que hacía que la verificación de la firma en el intercambio de claves del servidor **no se ejecutara** correctamente. Esto permitía que clientes iOS y OS X aceptaran certificados TLS/SSL falsos, facilitando ataques *man-in-the-middle (MITM)*. Este bug recibió el identificador CVE-2014-1266.

Causa técnica

El bug se debía a un `goto fail;` duplicado en el código de verificación de firmas. Debido a la falta de llaves en los bloques `if`, el segundo `goto` se ejecutaba siempre, saltándose la verificación real y devolviendo éxito por defecto.

Pseudocódigo vulnerable:

```
if ((err = some_check(...)) != 0)
    goto fail;
if ((err = other_check(...)) != 0)
    goto fail;
goto fail;   // <-- este segundo "goto fail" hace saltar la verificación real
/* código de verificación nunca alcanzado */
fail:
   /* cleanup; return err; */
```

Pseudocódigo corregido:

```
if ((err = some_check(...)) != 0)
    goto fail;
if ((err = other_check(...)) != 0)
    goto fail;
/* seguir con la verificación correctamente */
```

Impacto

- Afectó versiones de iOS anteriores a 7.0.6 y 6.1.6, y OS X 10.9.x sin parche.
- Cualquier aplicación que usara `SecureTransport` podía confiar en certificados no verificados.
- Permitía interceptar y modificar tráfico HTTPS en redes no seguras.

Lecciones prácticas

1. **Mediación completa:** toda verificación crítica debe ejecutarse siempre y no puede ser saltada por rutas de control inesperadas.
2. **Evitar constructos frágiles:** 'goto' sin llaves o 'if' sin delimitadores claros facilitan errores. Prefiera funciones pequeñas y manejo consistente de errores.
3. **Tests unitarios y cobertura:** extraer la lógica de verificación a funciones probadas unitariamente ayuda a detectar saltos de ejecución no deseados.
4. **Análisis estático y revisiones de código:** SAST y code reviews son fundamentales en código criptográfico.
5. **Actualizaciones rápidas:** mantener procesos de parcheo y comunicación con usuarios para corregir vulnerabilidades críticas.

Mitigaciones para desarrolladores

- Aplicar parches del sistema operativo si se utiliza la pila SSL/TLS de Apple.
- Encapsular la lógica de verificación y añadir tests de regresión para certificados inválidos y rutas de fallo.
- Revisiones de código y análisis estático en componentes críticos.
- Evitar construir verificaciones criptográficas propias; usar APIs probadas.

5.2.4 Buffer Overflow

Un **buffer overflow** ocurre cuando un programa escribe más datos en un búfer (área de memoria de tamaño fijo) de los que el búfer puede contener. En lenguajes de bajo nivel como `C` esto es muy común porque el lenguaje *no* realiza comprobaciones automáticas de límites sobre accesos a arrays o cadenas.

Error típico: asumir que un buffer de tamaño fijo es siempre lo suficientemente grande para almacenar toda la entrada.

- **Pregunta:** ¿qué pasa si llega más datos que espacio disponible en el buffer?
- **Respuesta:** los datos extra se escriben en memoria contigua, *después* del buffer, sobrescribiendo variables locales, punteros, el frame pointer y la dirección de retorno (stack).

Código vulnerable:

```
void funcion(char *str) {
    char buffer[16];
    strcpy(buffer, str);   // copia sin comprobar longitud
}

int main() {
    char cadena[256];
    int i;
    for (i = 0; i < 256; i++) cadena[i] = 'A';
    cadena[255] = '\0';
    funcion(cadena);
    return 0;
}
```

En este ejemplo la función `funcion` reserva un buffer local de 16 bytes en el *stack*, pero se le pasa una cadena de 256 bytes. `strcpy` copia sin límite, lo que sobrescribe memoria más allá de `buffer`.

Estado conceptual del stack (direcciones desde altas a bajas):

```
Dir. Altas de memoria  <-- (arriba)
...
[ argumentos de main ]
[ cadena (en heap o data) ]
...
[ retorno a main  ]  <-- dirección de retorno
[ saved frame pointer (sfp) ]
[ buffer[16] ]        <-- espacio reservado
Dir. Bajas de memoria  <-- (abajo)
```

Si se escriben más de 16 bytes en `buffer`, los bytes adicionales sobrescriben primero el `sfp` y luego la `dirección de retorno` (`ret`). Al corromper la dirección de retorno, cuando la función haga `return`, la ejecución saltará a la dirección controlada por el atacante.

Cómo se explota ?

1. El atacante suministra una entrada cuidadosamente construida: `[padding]` + `[nueva_ret]` + `[payload]`.
2. El `padding` ocupa exactamente hasta el lugar donde se sobrescribe la dirección de retorno.
3. `nueva_ret` contiene una dirección a la que se quiere transferir ejecución (por ejemplo, al principio del `payload` que también se colocó en el stack).

4. Cuando la función retorna, salta a `nueva_ret` y el `payload` (por ejemplo, shellcode) se ejecuta con los privilegios del proceso vulnerable.

Técnicas relacionadas: *stack smashing*, inyección de código (shellcode), *return-to-libc* o *return-oriented programming (ROP)* cuando las protecciones impiden ejecutar el stack.

Consecuencias

- Ejecución arbitraria de código con los privilegios del proceso vulnerable.
- Escalada de privilegios si el binario es `setuid` o corre con privilegios elevados.
- Denegación de servicio o corrupción de datos.
- Compromiso completo del sistema en entornos no mitigados.

Medidas de mitigación y buenas prácticas

A nivel de código

- Evitar funciones inseguras: `strcpy`, `strcat`, `gets`, etc.
- Usar versiones acotadas y seguras: `strncpy`, `snprintf`, `strlcpy` (si está disponible), `fgets` para lecturas desde stdin.
- Validar y normalizar la entrada (lista blanca cuando sea posible).
- Comprobar siempre longitudes antes de copiar o concatenar.
- Uso de tipos y estructuras que incluyan el tamaño y operaciones que verifiquen límites.

A nivel de compilador / sistema

- **Stack canaries / Stack cookies:** valores colocados entre buffers locales y el saved frame pointer/return address que se verifican al retornar; si han cambiado, el proceso aborta. (ej.: `-fstack-protector` en gcc/clang).
- **NX / DEP (Non-Executable Stack):** marcar el stack como no ejecutable para impedir que el shellcode colocado en el stack sea ejecutado.
- **ASLR (Address Space Layout Randomization):** aleatoriza direcciones de memoria (stack, heap, librerías) para dificultar predecir valores de retorno y gadgets.
- **Relocation and RELRO / PIE:** enlaces más seguros y binarios recompilables con posiciones aleatorias.
- **Fortify Source:** `-D_FORTIFY_SOURCE` (con optimizaciones) para que ciertas funciones detecten desbordes en tiempo de ejecución.

A nivel de arquitectura

- Minimizar privilegios de procesos (principio de menor privilegio).
- Separar componentes: procesos con funciones sensibles no deben compartir espacio con procesos expuestos.
- Revisiones de código, análisis estático (SAST) y pruebas dinámicas (DAST, fuzzing) enfocadas en entradas limites.

Alternativas seguras a C para evitar estos errores

- Lenguajes con comprobación de límites en tiempo de ejecución: `Rust`, `Go`, `Java`, `C#`, etc.
- Si se debe usar C/C++, aplicar rigurosamente patrones seguros, encapsular en interfaces bien testeadas y revisar las APIs que manipulan memoria.

5.2.5 Shellcode

Un **shellcode** es un pequeño fragmento de código máquina (binario) usado como *payload* en la explotación de una vulnerabilidad de software (por ejemplo, un *buffer overflow*). Históricamente se denomina así porque muchos payloads arrancaban un *shell* (interprete de comandos, p. ej. `/bin/sh`) para dar control interactivo al atacante. En la práctica actual un shellcode puede realizar múltiples acciones: crear usuarios, descargar y ejecutar binarios, abrir canales inversos de comunicación, cargar stagers, etc.

Características típicas

- Tamaño reducido (típicamente unos pocos bytes a kilobytes).
- Posicion-independiente (PIC): no depende de direcciones absolutas.
- Evita bytes problemáticos (por ejemplo `0x00`) que puedan truncar la explotación.
- Diseñado para invocar *syscalls* o instrucciones mínimas para lograr el objetivo sin depender de librerías de alto nivel.
- Ejecuta con los privilegios del proceso vulnerable.

Tipos y técnicas

- **Stageless:** el shellcode contiene todo el payload en un único bloque.
- **Staged:** un primer stage pequeño (stager) descarga o recibe un segundo stage mayor que ejecuta la carga útil completa.
- **Return-to-libc / ROP:** alternativas cuando la pila no es ejecutable; reutilizan código existente o gadgets.
- **Egg-hunters:** pequeños buscadores que localizan el payload en memoria cuando el espacio disponible es limitado.
- **Encoders/Obfuscators:** codifican el shellcode para evitar bytes nulos o firmas, e incluyen un decoder stub en ejecución.

Restricciones prácticas

- **Bytes nulos y caracteres especiales:** muchas explotaciones tratan la entrada como cadena C; un `0x00` puede truncar el payload.
- **Tamaño limitado del espacio de inyección:** obliga a optimizar y a utilizar técnicas compactas.
- **Arquitectura y ABI:** instrucciones y convenciones difieren entre x86, x86_64, ARM, MIPS, etc.
- **Mitigaciones del sistema:** NX/DEP, ASLR, canarios de pila, RELRO, CFI, que afectan la factibilidad de ejecución directa del shellcode.

Ejemplo

A continuación se muestran dos fragmentos : (1) una versión en C que invoca `execve("/bin/sh", ...)` y (2) la representación típica de un shellcode binario en un array de bytes.

Ejemplo en C (función que llama a `execve`):

```
#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);
}
```

Representación binaria (ejemplo de shellcode):

```
char shellcode[] =
"\xeb\x2a\x5e\x89\x76\x08\xc6\x46\x07\x00"
"\xc7\x46\x0c\x00\x00\x00\x00\xb8\x0b\x00"
"\x00\x00\x89\xf3\x8d\x4e\x08\x8d\x56\x0c"
"\xcd\x80\xb8\x01\x00\x00\x00\xbb\x00\x00"
"\x00\x00\xcd\x80\xe8\xd1\xff\xff\xff\x2f"
"\x62\x69\x6e\x2f\x73\x68\x00\x89\xec\x5d"
"\xc3";
```

Este tipo de array contiene la secuencia de bytes de un shellcode clásico que pretende construir la cadena `"/bin/sh"` en memoria y llamar a la syscall correspondiente.

Variantes modernas

- En x86_64 y arquitecturas RISC las convenciones y números de syscall cambian, por lo que el shellcode debe adaptarse.
- Cuando el stack es no ejecutable, se usan *return-to-libc* o *ROP* para encadenar llamadas legítimas y lograr la ejecución de comportamientos maliciosos.
- Los payloads modernos incluyen funcionalidades avanzadas de *command and control* (C2), persistencia y evasión.

Detección y defensa

Prevención:

- Eliminar la vulnerabilidad raíz: validación y saneamiento de entradas, evitar funciones inseguras (`strcpy`, `gets`, etc.).
- Aplicar el principio de menor privilegio: procesos expuestos con los permisos mínimos necesarios.
- Evitar binarios `setuid` innecesarios.

Mitigaciones a nivel sistema:

- **NX/DEP**: marcar la pila/heap como no ejecutable.
- **ASLR**: aleatorizar el layout de memoria para complicar saltos fiables.
- **Stack canaries**: detectar corrupción del stack antes del `return`.
- **RELRO / PIE**: proteger GOT/PLT y permitir relocación aleatoria de binarios.
- **Control Flow Integrity (CFI)** y mecanismos avanzados de protección.

Detección:

- Firmas y heurísticas en AV/IDS/IPS (patrones de instrucciones, `int 0x80`, secuencias raras en memoria).
- Monitoreo del comportamiento: creación de procesos inusuales, conexiones salientes anómalas, cambios en ficheros críticos.
- Emulación/sandboxing para análisis de muestras y uso de reglas YARA/Suricata/Zeek para correlación.

5.2.6 Race Conditions — TOCTTOU (Time Of Check, Time Of Use)

Una **race condition** ocurre cuando el comportamiento correcto de un programa depende del orden o la sincronización de eventos externos y un atacante puede alterar ese orden para inducir un estado inseguro. Un caso clásico en sistemas de ficheros es *TOCTTOU* (Time Of Check — Time Of Use), en el que se comprueba una condición sobre un recurso y, antes de usarlo, el recurso es cambiado por un atacante.

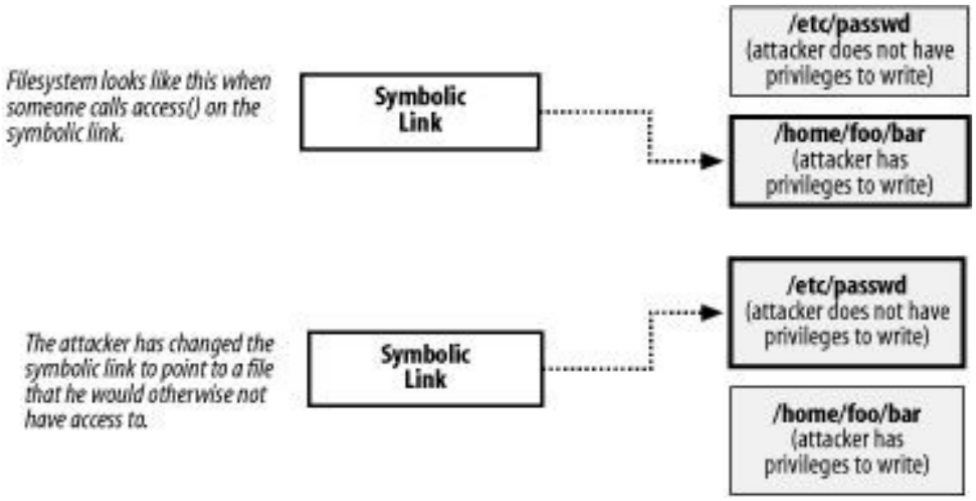
Ejemplo (TOCTTOU)

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

Supongamos que este programa es `setuid` (ejecutado con un *effective uid* distinto del *real uid*). La llamada `access()` verifica los permisos del *real uid* (el usuario que ejecutó el programa originalmente), mientras que `open()` se realiza bajo el *effective uid* (el privilegio elevado del programa). Esto abre una ventana de tiempo entre la comprobación (`access`) y el uso (`open`) en la que un atacante puede cambiar el contenido o la identidad del fichero (por ejemplo, reemplazar un fichero por un symlink hacia `/etc/shadow`).

Cómo se explota ?

1. El atacante prepara un archivo `file` al que él tiene acceso.
2. El programa invoca `access(file, R_OK)` y la comprobación pasa (porque el atacante posee el archivo).
3. Antes de que el programa ejecute `open(file, ...)`, el atacante reemplaza `file` por un `symlink` que apunta a un recurso sensible (p. ej. `/etc/passwd`).



4. Cuando el programa ejecuta `open` con el *effective uid* privilegiado, abre el recurso sensible y realiza operaciones con privilegios elevados, lo que puede permitir lectura/escritura no autorizada.

Por qué esto es crítico en programas `setuid`

- **Diferencia real vs effective UID:** muchas funciones (como `access()`) usan el *real* UID para comprobaciones de acceso; `open()` se evalúa con el *effective* UID. Esta discrepancia crea una ventana de explotación.
- **Ventana temporal:** entre la verificación y el uso hay tiempo suficiente para que un atacante cambie el recurso (si existe acceso concurrente).
- **Privilegios elevados:** si el programa tiene privilegios (root) el attacker puede inducir lecturas o escrituras sobre ficheros protegidos.

Mitigaciones y buenas prácticas

- **Evitar la comprobación-uso separada:** no hacer `access()` seguido de `open()`. En su lugar, abrir directamente el fichero y comprobar el descriptor resultante.
  - Por ejemplo, llamar `fd = open(file, O_RDONLY | O_NOFOLLOW);` y verificar `fd >= 0`. Evitar operaciones basadas en rutas después de abrir.
- **Usar banderas atómicas de `open()`:**
  - `O_NOFOLLOW` evita seguir symlinks.
  - `O_CREAT | O_EXCL` permite crear ficheros de forma atómica si se requiere crear un recurso seguro.
  - En Linux moderno, `openat()` y `fstatat()` permiten operaciones relativas a descriptores de directorio, reduciendo carreras con directorios.
- **Operar sobre descriptores de fichero:** después de `open()`, usar `fstat(fd, ...)` para verificar identidad, permisos y que no sea un dispositivo o enlace peligroso; realizar todas las operaciones usando `fd` (no volver a usar la ruta).
- **Comprobar propietario/inode:** usar `stat()` o `fstat()` para comprobar que el fichero no cambió (comparar `st_ino`, `st_dev`) si se usa un patrón check-then-use ineludible.
- **Reducir la ventana temporal:** minimizar la cantidad de tiempo entre comprobación y uso; pero esto *no* elimina la posibilidad de TOCTTOU.

- **Deshabilitar privilegios cuando no sean necesarios:** en programas `setuid`, usar `seteuid()`, `setuid()` para bajar privilegios antes de manipular recursos controlables por usuarios no confiables; elevar solo cuando sea imprescindible y de manera controlada.
- **Bloqueo de directorio/archivo:** usar mecanismos de locking (por ejemplo `flock()` o fcntl locks) cuando proceda, aunque los locks no siempre previenen todos los vectores de TOCTTOU relacionados con symlinks.
- **Operaciones atómicas en directorios:** usar `openat()`, `rename()` atómico y otras operaciones específicas del sistema de ficheros que sean atómicas para evitar sustituciones de recurso.
- **Separación de responsabilidades y diseño seguro:** evitar que binarios con privilegios realicen operaciones sobre archivos controlados por usuarios donde no sea estrictamente necesario.
- **Revisiones y pruebas:** incluir tests concurrentes y fuzzing para detectar condiciones de carrera; auditar código `setuid`.

5.2.7 Vulnerabilidades por Format String

Una vulnerabilidad por **format string** surge cuando una función que interpreta cadenas de formato (por ejemplo `printf`) recibe control total del formato desde una fuente no confiable (p. ej. entrada del usuario). Si el formato lo controla un atacante, éste puede leer o incluso escribir en memoria usando especificadores como `%x` (leer palabra de la pila) o `%n` (escribir el número de bytes impresos en una dirección indicada).

Seguro: el formato está fijo en el código:

```
printf("%s = %d\n", variable, value);
```

Inseguro: el formato proviene del atacante:

```
printf(error_message);   // peligroso si error_message es controlado por usuario
```

Funciones implicadas

Muchas funciones en C y en APIs de sistemas usan formatos de cadena:

- `printf`, `fprintf`, `sprintf`, `snprintf`, `vprintf`, ...
- `syslog` (peligroso si se llama `syslog(user_msg)` con `user_msg` sin formato seguro).
- `setproctitle` u otras utilidades que aceptan formato.

Ejemplo vulnerable

```
#include <stdio.h>

int main(int argc, char* argv[]) {
    if (argc > 1)
        printf(argv[1]);   // inseguro: el primer argumento controla el formato
    return 0;
}
```

Si se ejecuta con argumentos controlados por un atacante, éste puede:

- Usar `%x` repetidos para leer palabras arbitrarias de la pila (filtrado de memoria).
- Usar `%s` para intentar imprimir cadenas desde direcciones en memoria (si se logra apuntar correctamente).
- Usar `%n` para escribir el número de bytes impresos en una dirección apuntada desde la pila, permitiendo escritura arbitraria en memoria bajo ciertas condiciones.

Cómo funciona el ataque ?

1. **Lectura de memoria:** mediante especificadores `%x`, `%s`, un atacante recorre la pila y obtiene valores internos del programa (punteros, valores, direcciones de retorno, etc.). Esto permite información útil para construir etapas posteriores del exploit.

2. **Escritura en memoria (`%n`):** `%n` hace que la función de formato escriba en la dirección representada por el correspondiente argumento de la pila el número de bytes impresos hasta ese punto. Si el atacante controla también las palabras en la pila (por ejemplo mediante la propia string de formato) puede dirigir esa escritura a direcciones de memoria arbitrarias y así modificar valores críticos (variables, flags, direcciones de retorno en la pila).

3. **Construcción de exploit:** utilizando una combinación de lecturas (para ubicar offsets y direcciones) y escrituras parciales (por ejemplo escribir byte a byte), un atacante puede redirigir ejecución o manipular credenciales/privilegios.

Importante: la explotación práctica requiere conocer el layout de la pila para esa invocación, evitar bytes nulos y otras restricciones del canal de entrada, y lidiar con mitigaciones como ASLR, canarios de pila y NX/DEP.

Ejemplo ilustrativo

Usando muchos `%x` en el formato puede desplegarse contenido de la pila (valores hexadecimales), p. ej.:

```
printf("%08x.%08x.%08x\n");
// salida ilustrativa (ejemplo): 40012980.080628c4.bffff7a4
```

Esto muestra cómo se pueden filtrar direcciones y punteros desde el contexto del proceso.

Consecuencias

- **Divulgación de información sensible:** keys, punteros, datos internos pueden filtrarse.
- **Ejecución arbitraria / escalada:** mediante `%n` y escritura en memoria se pueden alterar direcciones de retorno o variables críticas, llevando a ejecución de código (o a redirecciones hacia gadgets en ROP).
- **Modificación de estado:** cambiar valores como UID, flags de autenticación, permisos, etc.

Mitigaciones y buenas prácticas

- **Nunca pasar directamente strings controladas por el usuario como formato:** usar `printf("%s", user_input)` en lugar de `printf(user_input)`.
- **Usar funciones seguras y con tope:** emplear `snprintf` con tamaño máximo y comprobar siempre el valor de retorno (por ejemplo, `ret = snprintf(buf, sizeof(buf), "%s", input);` y verificar `ret`).
- **Evitar APIs que interpreten formato si el mensaje viene del exterior:** por ejemplo, usar `syslog(LOG_ERR, "%s", user_msg);` en lugar de `syslog(LOG_ERR, user_msg);` .
- **Compilador y herramientas:**
  - Habilitar warnings del compilador (gcc/clang advierten de `printf(argv[1])` sin formato).
  - Usar `-Wformat` y `-Wformat-security`.
  - Aplicar análisis estático (SAST) que detecte llamadas peligrosas.
- **Mitigaciones a nivel de sistema:** ASLR, NX/DEP, stack canaries y RELRO reducen la probabilidad de explotación con éxito.
- **Validación y saneamiento:** restringir y filtrar la entrada cuando sea posible; evitar permitir caracteres que faciliten manipulaciones del formato.
- **Revisiones y tests:** code review enfocado en APIs de formato, pruebas dinámicas y fuzzing para detectar usos inseguros.

Detección y respuesta

- **Detección estática:** linters y SAST señalan usos inseguros de `printf`-like cuando el primer argumento no es literal.
- **Monitoreo en runtime:** IDS/EDR que detecte prints anómalos, filtrado de memoria o comportamiento extraño tras entradas formateadas.
- **Pruebas de seguridad:** fuzzers que envíen formatos con `%x`, `%s`, `%n` y correlacionen respuestas para detectar fugas o escrituras.

5.2.8 Back Door

Una **back door** es funcionalidad adicional incluida deliberadamente (o inadvertidamente) en una aplicación que permite, en un momento posterior, eludir los mecanismos normales de control de acceso. Suele ser introducida por un desarrollador malintencionado o por una modificación no revisada del código, y puede tomar formas diversas: cuentas ocultas, credenciales "maestras", rutas de administración no documentadas, flags de depuración que deshabilitan controles, comandos ocultos, etc.

Ejemplos

- Un nombre de usuario/contraseña especial que siempre autentica correctamente.
- Un parámetro HTTP secreto que activa un modo administrativo sin autenticación.
- Un módulo o función que restaura permisos o desactiva logging bajo ciertas condiciones.
- Inclusión de un servicio remoto que permite ejecución de comandos remotos (backdoor remoto).

Riesgos

- El acceso no autorizado y prolongado al sistema.
- Dificultad para detectar intrusiones porque la back door puede simular comportamiento legítimo.
- Escalada de privilegios y exfiltración de datos.
- Pérdida de integridad del software y reputación de la organización.

Detección y mitigaciones

- **Procedimientos de Quality Assurance (QA):** revisar cambios críticos mediante procesos de code review obligatorios y checklist de seguridad antes de aceptar merges.
- **Análisis estático y dinámico:** emplear SAST para detectar código sospechoso o porciones no referenciadas, y DAST/fuzzing en endpoints administrativos.
- **Política de Pull Requests y revisiones múltiples:** exigir revisiones por pares y aprobaciones de miembros de seguridad para cambios sensibles.
- **Auditoría de dependencias y terceros:** revisar bibliotecas y módulos adquiridos que puedan contener puertas traseras.
- **Revisión de logs y alertas:** detectar accesos inusuales, patrones anómalos o cuentas que usan privilegios fuera de lo esperado.
- **Pruebas de integridad:** firmas de binarios, checksums y validaciones que aseguren que el artefacto desplegado coincide con la versión revisada.
- **Separación de entornos y control de despliegue:** impedir despliegues directos a producción sin pasar por pipelines controlados y auditables (CI/CD con firmas).
- **Rotación de credenciales y gestión de secretos:** evitar credenciales embebidas en código; utilizar vaults y rotación periódica.

5.3 Operación

La fase de **operación** cubre los ataques y problemas que surgen una vez que la aplicación está en producción. Muchas vulnerabilidades explotables no son fallos del código per se, sino decisiones operativas o de configuración.

Problemas típicos en operación

- **Cuentas con claves por defecto:** dejar credenciales de fábrica sin cambio permite acceso trivial a servicios y dispositivos.
- **Instalación por defecto:** configuraciones por defecto a menudo priorizan facilidad sobre seguridad (puertos abiertos, servicios innecesarios).
- **Manuales de uso inadecuados:** documentación insuficiente puede llevar a configuraciones inseguras por parte de administradores.
- **DoS (Denegación de Servicio):** falta de controles de rate limiting, límites de recursos o arquitectura resiliente permite interrupciones de servicio.
- **Política pobre de actualizaciones de seguridad:** retrasar parches o no aplicar hotfixes deja sistemas expuestos a vulnerabilidades conocidas.

Buenas prácticas operacionales

- **Hardenización por defecto:** publicar imágenes y configuraciones seguras por defecto (principio de mínima exposición).
- **Gestión de parches:** mantener un proceso formal de evaluación, testing y despliegue de parches priorizando riesgos y ventanas de mantenimiento.
- **Gestión de cuentas y accesos:** inventario de cuentas, eliminación de cuentas inactivas, control de privilegios y políticas de contraseñas/2FA.
- **Monitoreo y respuesta a incidentes:** logs centralizados, correlación, detección de anomalías y planes de respuesta a incidentes.
- **Backups y planes de recuperación:** estrategias de backup y procedimientos de recuperación probados periódicamente.
- **Pruebas de resiliencia:** ejercicios de stress, test de DoS, pruebas de failover y simulacros de incidentes.
- **Documentación y formación:** guías claras y formación para operadores sobre configuraciones seguras y procedimiento ante incidentes.