Resumen Seguridad

Tomás Felipe Melli

Noviembre 2024

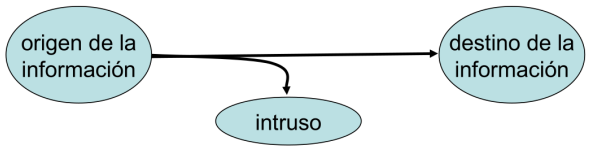
Índice

1 Introducción

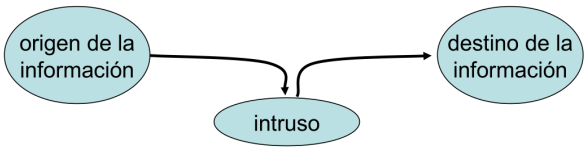
Cuando hablamos de **seguridad de la información** tenemos que hacer la distinción con **seguridad informática**, ya que esta última es un subconjunto de la primera. Cuando hablamos de seguridad de la información, *hablamos de proteger la información en cualquier formato : digital, físico y verbal* y acá vemos que el formato digital es sub-categoría. Pero no sólo eso, proteger la información en todos sus formatos tiene como objetivos principales garantizar :

1.1 Objetivos

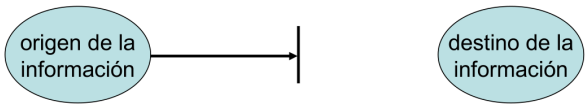
- **Confidencialidad** : la información sólo es accesible para aquellos sujetos que tienen los permisos para acceder a ella.
  Ejemplo : compartir fotos de una mina con la que estuviste en la que la mujer se encuentra desnuda (sin el consentimiento de la chica) es un *ataque a la confidencialidad* ya que los demás sujetos (los observadores) no debería ser accesible para ellos (ya que no cuentan con los permisos)



- **Integridad** : la información no ha sido alterada por un tercero sin permisos.
  Ejemplo : pensemos en un canal de comunicación como un email. Supongamos que un sujeto intercepta el mensaje y hace cambios en la información, eso es un *ataque a la integridad*.



- **Disponibilidad** : la información es accesible siempre para cualquiera de los usuarios autorizados.
  Ejemplo : vas al chino, querés comprar pan para hacer sanguchitos y a la hora de pagar querés usar tu banco para pagar con QR. Lamentablemente, el servidor del banco no está en línea. Sabés que tenés los recursos para pagar, pero no los tenés disponibles. Cualquier atacante que busque tirar un sistema como este, estaría *atacando la disponibilidad*.



1.2 Abstracciones

En el contexto de la *seguridad de la información* tendremos **sujetos**, **objetos** y **acciones**. Estos sujetos son los que realizan acciones sobre los objetos. Por ejemplo, un *usuario* es un sujeto del sistema operativo que ejecuta acciones y que a veces es *owner* de archivos, procesos, memoria, conexiones, puertos. Las acciones que ejecuta incluye : leer, escribir, hacerle *kill* a un proceso. Como sujeto de un sistema operativo tenemos también a los **grupos** que son colecciones de usuarios. Como son sujetos del sistema operativo también tienen permisos definidos y por tanto, son sujetos del sistema de permisos. Existe otra abstracción del sistema, los **roles**. Es decir, se puede definir un rol a un usuario : ejemplo usuario común / administrador.

1.3 Triple A

En el contexto de Seguridad de la Información existen *3 conceptos* comúnmente referidos como la *AAA (triple A)* como conjunto de principios fundamentales utilizados para gestionar y asegurar el acceso a recursos de sistemas y redes. Estos principios son :

- **Authentication** : es el proceso de verificar la identidad de un usuario, dispositivo o sistema. Sos quien decís ser ? Para responder a esta pregunta, el sistema implementa ciertos métodos de autenticación como : contraseñas, múltiples factores, datos biométricos ...
- **Authorization** : es el proceso de determinar *a qué recursos* puede acceder el usuario (que ya auntenticamos) y *qué acciones* puede realizar sobre los objetos del sistema.
- **Accounting** : es el proceso de monitoreo y registro de las acciones que ese sujeto hace en el sistema. La idea es poder rastrear el comportamiento y detectar actividades sospechosas.

1.4 Criptografía

En Seguridad se suelen implementar mecanismos **cripográficos** para **cifrar / descifrar** información en contextos de intercambio de mensajes para garantizar que nadie pueda leer el mensaje (en caso de Sniffing) ya que se encuentra **encriptado**, esto garantiza **confidencialidad**. Para lograr este cifrado, se utiliza una función llamada **función de hash** que lo que hace es tomar como parámetro el **texto claro** (el texto legible) y devuelve un texto ilegible llamado **texto encriptado o cifrado**. Esta función devuelve un valor único que es el **hash** o **firma digital**. Al ser único, es posible saber si ha habido alguna modificación en el mensaje, por tanto, esta garantiza **integridad** (esto se logra con las funciones de hash *one-way*. Ahora bien, dijimos que la función lo convierte en algo ilegible... y entonces cómo logra leerlo el que lo recibe ? Existen *tres algortimos* :

- **Algoritmos simétricos** : son aquellos algortimos de encriptación en los que se utiliza *la misma clave tanto para cifrar como para descifrar*. El tema es cómo le mandás al otro la clave sin que en el medio te la capturen digamos.

- **Algoritmos asimétricos**: son algoritmos de encriptación en los que se utilizan *dos claves diferentes*. Este algoritmo viene a resolver el problema de compartir la clave para descifrar. La idea es la siguiente: cada usuario tiene su propia **clave privada** que nunca se comparte y siempre está en su posesión. Pero también tiene una **clave pública** que está siempre visible a cualquier persona. ¿Cómo funciona esto? Existe una relación matemática entre ambas claves en la que **la clave privada es utilizada para descifrar un mensaje cifrado con la clave pública**. Ejemplo Algoritmo RSA. En este algortimo la relación matemática es: dos números primos grandes tales que

$$n = p \times q$$

Donde n es el módulo y es parte tanto de la clave privada como de la pública. Se utiliza la función de Euler $\varphi(n)$ para sacar la clave privada. Se elige el *exponente público* e tal que el máximo común divisor (MCD) de e y $\varphi(n)$ sea 1:

$$\text{MCD}(e, \varphi(n)) = 1$$

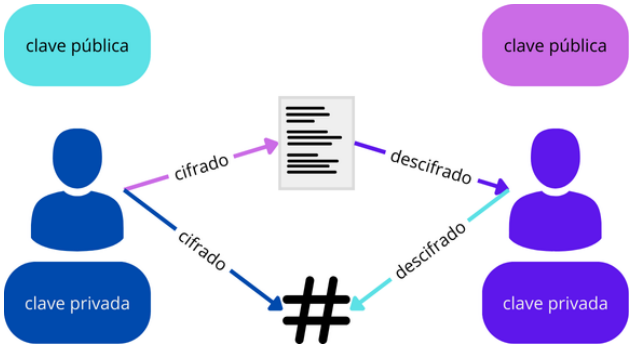
Se calcula el *exponente privado* d, que es el inverso multiplicativo de e módulo $\varphi(n)$, es decir, se debe encontrar d tal que:

$$e \times d \equiv 1 \pmod{\varphi(n)}$$

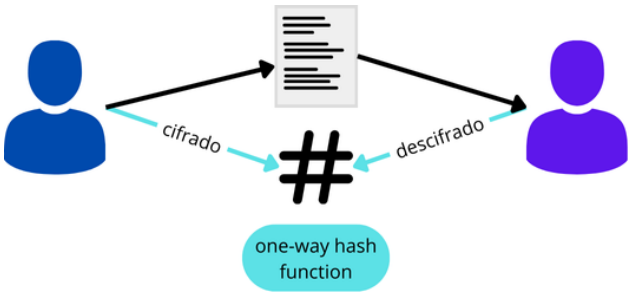
Dicho esto, el cifrado ocurre como sigue: **mensaje cifrado** $c = m^e \mod n$, y el descifrado sucede por:

$$(m^e)^d \equiv m \pmod{n}$$

** En resumen, es interesante la forma en que se garantiza la confidencialidad de este algoritmo con esta relación matemática que es computacionalmente difícil de hallar para valores grandes. Además se puede garantizar la integridad y autenticidad del mensaje si el *emisor* lo firma. Esta firma no forma parte del encriptado del mensaje, pero sí de información adicional que se envía con él. Para lograrlo, el emisor calcula el hash del mensaje con la clave pública del receptor (como ya vimos); luego, con su clave privada procede a calcular el hash de ese hash (es decir, una cosa es el cifrado del mensaje, y otra cosa es el cifrado de ese hash). El receptor ahora con su clave privada puede descifrar el mensaje y luego va descrifrar el segundo hash (el que se usa para la integridad). Una vez que tiene el mensaje descifrado y el hash con el que vino, debe capturar la clave pública del emisor para descifrar el hash utilizado para chequear integridad, si este descifrado le da que es el mismo hash con el que le llegó el mensajo, entonces nadie modificó el mensaje :



- **one-way** : son algortimos que hacen uso de funciones de hash que **son prácticamente irreversibles** porque son computacionalmente difíciles de revertir. Otra cualidad importante es que son muy resistentes a colisiones (es decir que es extremadamente raro que dos entradas diferentes coincidad en el hash). Como son determinísticas, para cierta entrada su resultado es siempre el mismo (son todos los resultados de tamaño fijo). SHA-256 es una función de hashing unidireccional muy conocida que produce un hash de 256 bits como resultado y hasta el momento es (computacionalmente imposible) de revertirlo y obtener el mensaje original. Normalmente este tipo de funciones se usan para *verificar la autenticidad* como por ejemplo *HMAC (Hash-based Message Authentication Code)* o al hacer un *checksum* en el que el servidor además de cierto mensaje te pasa un checksum del archivo (un hash) y vos luego de descargarlo le calculás el hash y chequeas si coinciden.



1.5 Replay Attack : las funciones de hash no lo impiden

Un *Replay Attack* ocurre cuando un atacante **intercepta y reenvía** un mensaje o transacción válida, con la intención de que el sistema o el destinatario crea que es una solicitud nueva y legítima. Es importante destacar que el atacante **no altera el mensaje, solo lo repite**. El ejemplo simple es : yo escucho la comunicación entre cierto usuario y su banco. Si yo logro capturar los paquetes que le envía el usuario en el que transfiere a cierta cuenta, y esos paquetes no los toco, los dejo tal cuál están, y al rato los mando de nuevo, el banco va a pensar que todo es legítimo, que el usuario necesita volver a transferir el dinero. Para prevenir este tipo de ataques se implementan *timestamps* o *nonces* (números aleatorios generados por el servidor o el cliente y que debe ser incluido en el mensaje (no se repiten)).

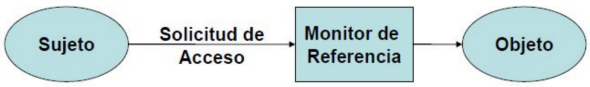
2 Control de Acceso

En la introducción comentamos que uno de los objetivos que tenemos cuando hablamos de seguridad de la información es la *confidencialidad*. Mencionamos que cuando los sujetos tienen *permisos* pueden acceder a cierta información, pero cómo se implementa esto ?

2.1 Monitor de Referencia

Este es una abstracción de seguridad que controla el acceso de un proceso a los recursos del sistema, asegurándose de que todos los accesos estén autorizados según las *políticas de seguridad* establecidad. Este componente tiene las siguientes responsabilidades :

- **Controla el acceso** que los sujetos tienen al momento de solicitar acceso a cierto recurso. Es decir, verifica, según la política de acceso establecida si dicho sujeto tiene los permisos adecuados.
- **Previene el acceso no-autorizado** de forma que, al constatar que cierto sujeto no tiene permisos, rechaza la solicitud de acceso.



2.2 Matriz de Control de Accesos

La forma de representar los permisos que tiene un sujeto sobre cierto objeto es matricialmente. Esto es una matriz de Sujetos x Objetos, en donde $\mathrm{M}[\mathrm{S}_i, O_j] = \{r_x, \ldots, r_y\}$, el conjunto de permisos $\{r_x, \ldots, r_y\}$

$$\begin{bmatrix} S_1 & O_1 & O_2 & \cdots & O_n \\ S_2 & O_1 & O_2 & \cdots & O_n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ S_m & O_1 & O_2 & \cdots & O_n \end{bmatrix}$$

2.3 DAC : Discretionary Access Control

El **Control de Acceso Discrecional** es un modelo de control de acceso. Es decir, define *cómo* se gestionan los permisos de acceso a los recursos del sistema. Sus características principales son :

- **Permiso basado en el propietario** : es decir, el **owner** del recurso tiene **autoridad para definir quién tiene acceso y de qué tipo** (lectura, escritura, ejecución,...)
- **Discrecional** : esto quiere decir que a discreción del propietario, los permisos de acceso pueden ser modificados dinámicamente (otorgando o revocando) sin necesidad de un control centralizado.
- **Herencia** : los permisos son heredados por los objetos que existan dentro de él.

Dicho esto, al existir tanta flexibilidad se vuelve difícil de auditar (qué sujetos tienen acceso a qué recursos) y en particular, los propietarios podrían accidentalmente dar permisos incorrectos.

2.4 MAC : Mandatory Access Control

Es otro modelo de control de accesos. En este modelo tenemos las siguientes características :

- **Permisos centralizados** : es decir, existe una **autoridad central** o también llamada **política de seguridad** que es la que controla los permisos (en contraposición al propietario como vimos antes).
- **Basado en etiquetas de seguridad** : todos los sujetos y objetos del sistema tienen etiquetas que definen el nivel de acceso permitido. Todos los accesos se controlan con estas. Un sujeto sólo puede acceder a objetos de gardo menor o igual al que figure en esta etiqueta.
- **No-modificable por el sujeto** : los permisos son **obligatorios** y los sujetos deben seguir las reglas predefinidas.

Este modelo suele ser más estricto en términos de seguridad al existir mayor control. Esto conlleva una administración más compleja de configurar y mantener.

2.5 Otros modelos

Existen otra variedad de modelos como el **RBAC : Role-Based Acces Control**, **OBAC : Obejct-Based Access Control**, **Bell-LaPadula**, entre otros.

3 Permisos en UNIX

En UNIX el modelo de control de acceso utilizado en DAC. Para cada Objeto (archivo) en el sistema existen 3 Sujetos (**Owner, Group, Others**). La forma en la que se representan es como sigue :

EL **Type** se refiere al tipo de archivo : como ya vimos cuando estudiamos File Systems, los tipos de archivos que tenemos son :

- **-** : archivos regulares
- **d** : directorios
- **l** : enlaces simbólicos (los hard-links no tiene distinción)
- **b / c** : block-device o char-device (representa los devices)

- **p** : pipe con nombre
- **s** : socket

En la terminal si corremos ls -l (long listing format) :

```
drwxr-xr-x  4 tonius tonius  4096 Nov  5 08:14 .
drwxr-x--- 34 tonius tonius  4096 Nov  6 10:17 ..
-rw-rw-r--  1 tonius tonius 12407 Oct 31 18:15 clase.txt
drwxrwxr-x 12 tonius tonius  4096 Nov  5 09:01 Compu
drwxrwxr-x  6 tonius tonius  4096 Oct  5 22:43 picoctf
```

Existen otros tipos especiales de permisos en UNIX:

- El **bit SETUID (user) / SETGID (group)** se puede setear en cierto archivo y esto le permite al archivo ejecutarse con permiso **s** los del **owner**. Esto es útil porque el acceso no es de **superusuario**, entonces tiene acceso para hacer ciertas modificaciones al sistema.
- **SuperUser DO** permite a un usuario autorizado a ejecutar un programa con privilegios elevados. En lugar de dar acceso completo a una cuenta de root, sudo permite que se ejecuten comandos con privilegios elevados de forma controlada y registrada.
- **StickyBit** es un permiso especial que se utiliza en directorios en sistemas Unix y Linux. Su principal objetivo es proteger los archivos dentro de un directorio, permitiendo que solo el propietario de un archivo o el root puedan eliminar o renombrar esos archivos, incluso si otros usuarios tienen permisos de escritura en el directorio.
- **chattr** (Change File Attributes) es un control adicional sobre cómo pueden ser modificados los archivos. Se usa como sigue :
  - *sudo chattr +i /...PATH* : el archivo no puede ser modificado, eliminado o renombrado, ni siquiera por el root.
  - *sudo chattr +a /...PATH* : sólo pueden tener datos añadidos, no modificados ni eliminados.
  - *sudo chattr +s /...PATH* : los datos del archivo se sobrescriben de manera que se vuelven irrecuperables, dificultando la recuperación de datos borrados.

4 Seguridad del Software : vulnerabilidades

Los momentos en los que hay que preocuparse por la seguridad del software son :

- **Arquitectura / Diseño** : cuando estamos pensando la aplicación.
- **Implementación** : cuando estamos escribiendo el código
- **Operación** : cuando ya está en producción

Los errores comunes suceden cuando **hacemos suposiciones sobre el ambiente del programa**, es decir, que el usuario va seguir las "reglas" que definimos en el aplicativo. Por ejemplo, que ingresará una cantidad acotada de caracteres. También es importante recordar que la **entrada** puede provenir de **variables de entorno** (PATH), **red local o externa** u otras fuentes.

Estos **bugs** se llaman **vulnerabilidades** y si esos bugs pueden ser aprovechados por atacantes para ejecutar código malicioso, le llamamos **exploitable bug**. Como dijimos que el error puede aparecer durante producción, supongamos que el atacante se entera de la vulnerabilidad **antes de que el proveedor haya lanzado una corrección**, a esa vulnerabilidad se la llama **Zero-Day** ya que el proveedor tiene *cero días para arreglarla* ;).

Existe un concepto llamado **Proof-of-Concept** que es una demostración de que una vulnerabilidad existe y que puede ser explotada existosamente. La finalidad es que a partir de esta prueba se puedan tomar decisiones para corregirla. Como dijimos antes, en el Zero-Day, el proveedor tendría que haber chequeado antes, en el 0day ya es tarde. Como contraparte existe el **exploit** que es un código diseñado para aprovecharse de la vulnerabilidad negativamente.

Como detalle de color : existen ataques altamente sofisticados y dirigidos en los que el atacante permanece mucho tiempo comprometiendo al sistema y lo hace silenciosamente con la motivación de monitorear, exfiltrar datos sin ser detectado. Este tipo de ataque se denomina **APT : Advanced Persistent Threat**. Si se quiere explayar en casos reales, buscar : APT28 (Fancy Bear, intrusión en el Comité Nacional Demócrata de EE.UU. en 2016, durante las elecciones presidenciales por parte de la inteligencia militar rusa), APT29 (Cozy Bear, intrusión a las agencias gubernamentales de EE.UU, también vinculado a Rusia), Stuxnet (un malware desarrollado por los gobiernos de EE.UU. e Israel para atacar las instalaciones nucleares de Irán).

4.1 Buffer Overflow

La situación es la siguiente, nosotros escribimos nuestro programa y como dijimos antes, suponemos que el usuario va a poner, por ejemplo, una contraseña de tamaño razonable :

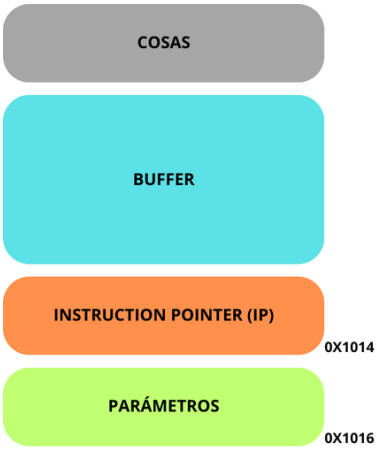
```
void login_ok() {
    printf("Login granted.\n");
    system("/bin/sh");
}

void login_fail() {
    printf("Login failed, password was not valid.\n");
}
```

```
bool validate_password() {
    char password[32];

    printf("Insert your password: ");
    scanf("%s", password);

    printf("Password is: ");
    printf(password);
    printf("\n");

    return strcmp(password, "atomicint") == 0;
}
```

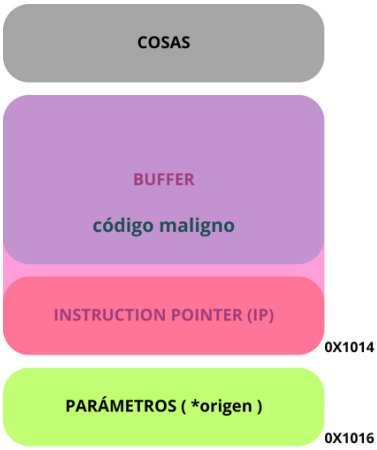
```
int main(int argc, char const *argv[]) {
    if (setuid(0) == -1) {
        perror("setUID ERROR\n");
    }

    if (validate_password()) {
        login_ok();
    } else {
        login_fail();
    }
    return 0;
}
```

Qué pasa si no? Volvamos a Orga-2 por un segundo... qué pasa cuando se invoca a una función en C ?

```
; Pasamos la dire de passw para testearla
mov eax, pass
; Llamamos a testpassw.C
call testpassw
```



El tema es el siguiente, el tamaño reservado para almacenar la contraseña es de 100 bytes. El problema viene cuando a la función **scanf** se le pasa la password, ya que si esta supera los 100 bytes, hay desbordamiento y en el caso de esta función, no hay control sobre qué cantidad de datos se copian en *password*, generando esto :



Y como resultado, **la dirección de retorno** se ve afectada. Y el chiste cuál es ? Que si el atacante sabe la dirección de memoria a la que salta el programa cuando se le da acceso, puede escribirla allí :

```
; Llamamos a testpassw.C
call testpassw
; Chequeamos respuesta de testpassw (true/false)
cpm eax, 1
je login_ok
; si no, login_fail
login_fail:

; login_ok est  en 0x5550 por ejemplo
login_ok:
```

La idea es que, conociendo dónde salta, pueda pasar cierta info para llenar el buffer con basura y los valores de **IP** ponerles esa dire, *0x5550*, y entonces cuando retorne, se irá directo dentro del sistema.

Una solución posible es reemplazar la función **scanf()** por **fgets()** (en este contexto) ya que esta última toma un **sizeof(password)**. Hay muchas otras que limitan la cantidad de caracteres (es una buena práctica para evitar **buffer overflow**) como *strncat, snprintf, strncopy.*

Otra solución es considerar que **cualquiar dato que proviene de una fuente no confiable al 100 porciento se considera TAINTED**. Por tanto, se procede a implementar mecanismos de **validación**

como limitar el tamño, verificar que coincida con el formato esperado, entre otras cosas.

4.2 SQL Inyection

Es una vulnerabilidad de seguridad en SQL en la que el atacante "inyecta" código malicioso SQL en una consulta SQL a través de entradas proporcionadas por formularios web, URL, parámetros de consulta, entre otras. Si la aplicación usa esos datos sin validarlos y los incorpora directamente a la base de datos, el atacante puede alterar la consulta y obtener resultados inesperados. Supongamos un formulario que pide que se ingrese el número de LU para aprobar a los alumnos. EL programa hace lo siguiente :

```
UPDATE alumnos SET aprobado=true WHERE lu='
```

Y el usuario, bicho-bicho, le pasa

```
307/08'; DROP alumnos; SELECT';
```

Logrando borrar toda la tabla de alumnos de la base de datos.

4.3 Malware

Se denomina **Malware** (malicious software) a todo software diseñado para llevar a cabo acciones no deseadas y sin el consentimiento explícito del usuario. Hay una gran variedad :

- **virus** es un tipo de malware que se adjunta a archivos o programas legítimos. Se replica y se propaga, infectando otros archivos cuando se ejecutan. Puede causar daños a sistemas, archivos o datos. Ejemplo : Un archivo de programa que, al abrirse, destruye archivos del sistema operativo.

- **troyanos** es un malware que se disfraza de software legítimo. El usuario lo ejecuta pensando que es seguro, pero en realidad, permite que un atacante tenga acceso remoto al sistema. Ejemplo : Un programa aparentemente inofensivo que da acceso a un atacante para robar información o controlar el sistema.

- **gusanos** Un gusano es un tipo de malware que se propaga por sí mismo a través de una red. No necesita un archivo o programa anfitrión para funcionar, solo se propaga de una máquina a otra, a menudo explotando vulnerabilidades. Ejemplo : Un gusano que se propaga automáticamente a través de una red local aprovechando una vulnerabilidad en un sistema operativo.

- **bots** Un bot es un software malicioso que controla un dispositivo de manera remota. A menudo se utiliza en una red de bots (botnet) para ejecutar ataques distribuidos, como DDoS** (Denial of Service). Ejemplo : Un bot que convierte una computadora en parte de una red de bots para enviar spam o realizar un ataque DDoS. ** un tipo de ataque cibernético cuyo objetivo es hacer que un servicio o sistema en línea sea indisponible al sobrecargarlo con una cantidad masiva de tráfico, impidiendo que los usuarios legítimos accedan a él.

- **adware** El adware es software que muestra anuncios no deseados. A menudo se instala junto con otro software gratuito, y su objetivo principal es generar ingresos mediante la publicidad. Ejemplo: Un programa que muestra ventanas emergentes o banners con anuncios mientras navegas por Internet.

- **keyloggers** Un keylogger es un tipo de malware que registra las teclas presionadas en el teclado del usuario. Esto permite a los atacantes obtener información sensible, como contraseñas y datos bancarios. Ejemplo: Un keylogger que graba las pulsaciones de teclas y las envía al atacante.

- **dialers** Un dialer es un tipo de malware que modifica la configuración de conexión de un dispositivo para realizar llamadas de pago a números de tarifa premium sin el conocimiento del usuario. Ejemplo: Un software que cambia la configuración de un módem para llamar a números de alta tarifa, generando cargos inesperados.

- **rootkits** Un rootkit es un conjunto de herramientas que permite a un atacante obtener y mantener el acceso privilegiado a un sistema sin ser detectado. A menudo oculta su presencia modificando el sistema operativo. Ejemplo: Un rootkit que permite que un atacante administre un sistema de forma remota sin que el usuario lo sepa.

- **ransomware** : El ransomware es un malware que bloquea el acceso a los archivos o sistemas de la víctima y exige un pago (ransom) para restaurar el acceso. Generalmente cifra archivos y pide un rescate. Ejemplo: Un programa que cifra archivos en tu computadora y pide dinero (generalmente en criptomonedas) a cambio de la clave para desencriptarlos.

- **rogueware** El rogueware es un software malicioso que se presenta como una herramienta legítima de seguridad (como un antivirus), pero en realidad es inútil o dañino. A menudo engaña a los usuarios para que paguen por una "licencia" para eliminar falsas amenazas. Ejemplo: Un falso antivirus que te muestra alertas de amenazas inexistentes y luego te pide dinero para eliminar esos "virus".

4.4 Race Condition

Las condiciones de carrera también pueden ocasionar comportamiento anómalo debido a una dependencia crítica en el timing de los eventos. Entonces presenta un potencial problema de seguridad ya que un atacante podría manipular el orden de ejecución para aprovechar el desfase temporal y obtener acceso no autorizado.

- Si un proceso intenta acceder a un recurso compartido, como un archivo, que puede ser modificado por otro proceso, un atacante podría aprovechar la race condition para modificar ese archivo o recurso antes de que el sistema verifique sus permisos, logrando así **escalar privilegios**.

- Si un sistema de banco tiene una race condition en el proceso de transferencia de dinero, un atacante podría explotar la condición de carrera para transferir más dinero del que tiene en su cuenta simplemente al aprovechar el tiempo entre dos verificaciones de saldo, logrando **manipular datos**.

- Las race conditions pueden ser utilizadas para eludir mecanismos de seguridad, como firewalls, sistemas de detección de intrusos (IDS), o mecanismos de autenticación. Un atacante podría explotar un momento específico en que el sistema no está validando correctamente una operación para realizar acciones no autorizadas o eludir la detección. Logrando **evadir medidas de seguridad**

- Si un atacante puede manipular la condición de carrera en un servicio crítico, puede hacer que el sistema se bloquee o falle, causando un **Denial of Service (DoS)** al impedir que los procesos legítimos accedan a los recursos necesarios.

5 Aislamiento

5.1 Sandbox

Un **Sandbox** es un entorno aislado en el que se ejecutan programas de manera controlada y segura. La idea es limitar el acceso a recursos del SO, en caso de que alguno de los procesos afecte negativamente al sistema principal. Se suelen utilizar estos entornos para hacer pruebas de software y análisis de malware. Existen diferentes tipos :

- **Sandbox de aplicaciones** son entornos específicos creados para aislar aplicaciones dentro de un sistema. Por ejemplo, los navegadores web como Google Chrome o Mozilla Firefox usan un modelo de sandboxing para ejecutar contenido web (como JavaScript) de forma aislada, evitando que los scripts maliciosos accedan al sistema del usuario.

- **Virtualización** como VMware o VirtualBox crean máquinas virtuales completamente aisladas del sistema principal, proporcionando un sandbox a nivel de hardware.

- **Contenedores** como Docker crean entornos de ejecución aislados a nivel de sistema operativo. A diferencia de las máquinas virtuales, los contenedores comparten el núcleo del sistema operativo, pero se mantienen separados del resto del sistema. Esto es más eficiente, pero sigue proporcionando aislamiento.
- **Análisis de Malware** como Cuckoo Sandbox permiten ejecutar malware en un entorno controlado y observar su comportamiento, incluyendo acciones como la creación de archivos, la manipulación del registro o las conexiones de red. Esto ayuda a los analistas a comprender cómo funciona el malware y qué acciones realiza, sin poner en peligro sistemas de producción.

Aunque parecería que estos entornos son infalibles, no enteramente. Los atacantes sofisticados pueden intentar esquivar o evadir el sandboxing mediante técnicas como la detección de sandbox. Algunas veces, el malware detecta si está corriendo en un entorno aislado y se comporta de manera diferente para evitar ser detectado.

5.2 chroot() y jail()

Son técnicas de aislamiento en UNIX que ayudan a restringir el acceso a ciertas partes del sistema de archivos o incluso limitar el entorno en el que se ejecutan ciertos procesos.

- **chroot()** cambia el directorio raíz del sistema de archivos para un proceso específico. Después de ejecutar chroot(), el proceso solo puede acceder a los archivos dentro de ese nuevo directorio raíz, lo que limita su capacidad de interactuar con otras partes del sistema. Cuando un proceso invoca la función chroot(), este proceso y sus descendientes quedan "atrapados" dentro de un subdirectorio, es decir, cualquier intento de acceder a archivos fuera de ese subdirectorio será denegado, ya que el nuevo directorio raíz actúa como la raíz del sistema para ese proceso. De todos modos no ofrece un aislamiento fuerte, ya que un proceso con privilegios elevados puede salir. Sólo limita el acceso a archivos, pero la CPU, la red o la memoria no la limita.
- **jail()** es una técnica que se utiliza principalmente en FreeBSD (y otros sistemas basados en BSD) para crear un "entorno de prisión" o jaula para un proceso. La diferencia con chroot() es que jail es un mecanismo más robusto y seguro. Aunque jail() también cambia el entorno del sistema de archivos y restringe el acceso de un proceso a ciertas áreas, proporciona más control sobre otros recursos del sistema, como las redes y la capacidad de interactuar con otros procesos. ** FreeBSD es un sistema operativo de tipo UNIX.

6 Buenas prácticas de seguridad

- Mınimo privilegio
- Simplicidad
- Validar todos los accesos a datos
- Separación de privilegios
- Minimizar la cantidad de mecanismos compartidos
- Seguridad multicapa.
- Facilidad de uso de las medidas de seguridad

7 Exploits de la práctica

7.1 Format String

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int BUFFERSIZE = 512;

int main(int argc, char **argv)
{
    char command[BUFFERSIZE + 1];

    if (setuid(0) == -1)
    {
        perror("setUID ERROR");
        exit(1);
    }

    snprintf(command, BUFFERSIZE, "ping -c 4 %s", argv[1]);

    printf("Executing: '%s'\n", command);
    system(command);

    return 0;
}
```

En este código se le cambia el UID del proceso a 0 con *setuid(0)*, por ello, si tiene permisos para hacerlo, adquirirá *privilegios de root*. La función *snprintf*, formateará el string que en simples términos es el comando que se ejecutará. *system* ejecutará aquello contenido en *command*.
Dónde está el tema ? En **argv[1]**, si no se controla adecuadamente, podés hacer algo como :

```
# el programa se ejecuta con argv[1] = "127.0.0.1; rm -rf /"
./programa "127.0.0.1; rm -rf /"
```

```
# y termina siendo ejecutado :
ping -c 4 127.0.0.1; rm -rf /
```

Para controlar este tipo de vulnerabilidad, es tan simple como chequear el input antes de ejecutarlo. Esto se puede hacer con un **allowlist** (es decir, validar el formato) o **blocklist** (es decir, saneamiento de caracteres peligrosos o inválidos) :

```c
// con BLOCK LIST
int is_safe_input(const char* input) {
    // Verifica si el input contiene caracteres peligrosos
    const char* invalid_chars = ";&|()";
    while (*input) {
        if (strchr(invalid_chars, *input)) {
            return 0;  // Entrada con caracteres peligrosos
        }
        input++;
    }
    return 1;  // Entrada segura
}

int main(int argc, char **argv) {
    /...
    // Validamos la entrada antes de usarla
    if (!is_safe_input(argv[1])) {
        printf("Entrada inv lida: contiene caracteres peligrosos.\n");
        return 1;
    }
    /...
    return 0;
}
```

Si el atacante escribiese esto:

```
$ ./programa - ping '127.0.0.1; /bin/sh'
Executing: 'ping -c 4 8.8.8.8; /bin/sh'
PING 127.0.0.1 (127.0.0.1) 56(84) bytes of data.
64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.071 ms
...
# whoami
root
```

Tendríamos todavía un problema de seguridad. Si usamos una validación de formato con *allow list* donde listamos las ip permitidas o con otro **Block list**, no deberíamos tener este problema :

```
// con otro BLOCK LIST
int is_valid_ip(const char* ip) {
    // Verifica que solo contenga n meros y puntos
    while (*ip) {
        if (!isdigit(*ip) && *ip != '.') {
            return 0;  // Entrada inv lida
        }
        ip++;
    }
    return 1;  // Entrada v lida
}

int main(int argc, char **argv) {
    /...
    // Validamos la IP antes de usarla
    if (!is_valid_ip(argv[1])) {
        printf("Invalid input: only numbers and dots are allowed.\n");
        return 1;
    }
    /...
    return 0;
}
```

7.2 Variables de entorno

```
int BUFFERSIZE = 512;

int main(int argc, char **argv) {
    char command[BUFFERSIZE + 1];
    char *ipaddr_sanitized = strtok(argv[1], " ;&|()");

    snprintf(command,
             BUFFERSIZE,
             "ping -c 4 %s",
             ipaddr_sanitized);

    if (setuid(0) == -1) {
        perror("setUID ERROR");
    }

    printf("Executing: '%s'\n", command);
    system(command);

    return 0;
}
```

Para este caso, se utiliza la función **strtok** para sanitizar lo que ingresa el usuario de algunos de los caracteres peligrosos, pero `;` , `/` no fueron sanitizados. Por tanto, un posible exploit de esto es :

```
$ echo -e '#!/bin/sh\n/bin/sh' > /tmp/ping
$ chmod +x /tmp/ping
$ export PATH="/tmp:$PATH"
$ ./ping '8.8.8.8'
Executing: 'ping -c 4 8.8.8.8'
```

```
# whoami
root
```

En donde, el atacante crea el archivo malicioso en */tmp/ping* con el **script** que cuando se ejecute, abrirá un shell. Ese script debe ser ejecutable, por ello el **chmod (change mode)**** se usa para darle permiso de ejecución. El atacante modifica la **variable de entorno PATH** de forma que el **el sistema buscará en el directorio /tmp cuando ejecute comandos**. Como el archivo /tmp/ping tiene el mismo nombre que el programa que intenta ejecutar (el que intenta está en /bin y el que el atacante usa está en /tmp, ahí está la joda). Esto concluye en que se inicia un shell con privilegios de root. Otro ejemplo de **escalado de privilegios** a causa de una vulnerabilidad de tipo **inyección de comandos o búsqueda de ejecutables** en directorios no seguros.
Para resolver esto, se debe **especificar el PATH absoluto**, ya que no importa lo que tenga PATH, siempre se ejecutará el programa correcto :

```
snprintf(command, BUFFERSIZE, "/bin/ping -c 4 %s", ipaddr_sanitized);
```

7.3 Buffer Overflow

```
void login_ok() {
    printf("Login granted.\n");
    system("/bin/sh");
}
void login_fail() {
    printf("Login failed, password was not valid.\n");
}
struct login_data_t {
    char password[100];
    bool valid;
} login_data;
void validate_password() {
    login_data.valid = false;
    printf("Insert your password: ");
    scanf("%s", login_data.password);
    printf("Password is: ");
    printf(login_data.password);
    printf("\n");
    if (strcmp(login_data.password, "porfis") == 0) {
        login_data.valid = 1;
    }
}
int main(int argc, char const *argv[]) {
    if (setuid(0) == -1) {
        perror("setUID ERROR\n");
    }
    if (validate_password()) {
        login_ok();
    } else {
        login_fail();
    }
    return 0;
}
```

Para este ejemplo, miramos el **struct**. Si prestamos atención, tiene un **bit de validez**, si logramos desbordar el buffer (100 bytes) y poner un 1 allí, todo listo. Para resolver esta vulnerabilidad, reemplazamos el **scanf** por:

```
fgets(login_data.password, sizeof(login_data.password), stdin);
```

7.4 Integer Overflow

Un Integer Overflow ocurre cuando se resta de un valor entero, y el valor resultante excede el rango del tipo de datos del entero, provocando que se ”desborde” hacia valores muy grandes o negativos.

```
static int grub_username_get(char buf[], unsigned buf_size) {
    unsigned cur_len = 0;
    int key;

    while (1) {
        key = grub_getkey();
        if (key == '\n' || key == '\r')
            break;

        if (key == '\b') { //  No  se verifica el underflow!
            cur_len--; //  Integer  Underflow!
            grub_printf("\b");
            continue;
        }
    }

    // Escritura fuera de los l mites
    grub_memset(buf + cur_len, 0, buf_size - cur_len);
    grub_xputs("\n");
    grub_refresh();

    return (key != '\e');
}
```

En este caso específico, lo que ocurre es que cuando la tecla de retroceso (/b) es presionada, el contador cur_len se decrementa sin comprobar que no se haya llegado a un valor menor que 0, lo que provoca un underflow. Esto es peligroso porque puede llevar a una escritura fuera de los límites (out-of-bounds write) más adelante en el código.

El código muestra que, si no se controla correctamente el valor de cur_len, puede llegar a ser negativo y eso puede causar que la función grub_memset escriba más allá de la memoria reservada para el búfer buf. Este tipo de vulnerabilidad es una potencial puerta de entrada para ataques, como la ejecución de código arbitrario (buffer overflow).

La solución es :

```
if (key == '\b') { // Verificar si cur_len es mayor que 0
    if (cur_len > 0) { // Asegurar que cur_len no sea negativo
        cur_len--;  // Decrementar solo si cur_len > 0
        grub_printf("\b");
    }
    continue;
}
```

Wrapper

```
#!/usr/bin/sudo python3
import os
import sys

FORBIDDEN = [";", "/", "(", ")", ">", "<", "&", "|"]

if len(sys.argv) <= 1:
    print("Use: ping IP")
    exit()

hostname = sys.argv[1]
for c in FORBIDDEN:
    if c in hostname:
        print("Wrong hostname!!")
        exit()

os.system("/sbin/ping -c 1 " + hostname)
```

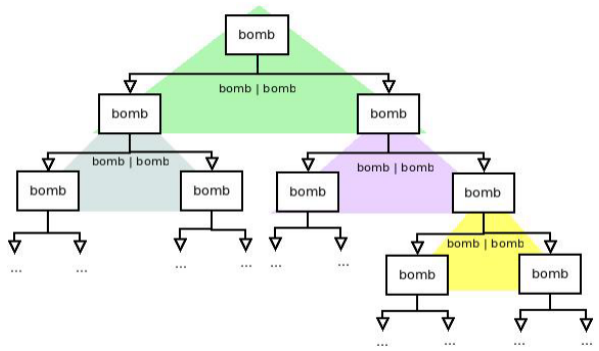
El script verifica que el usuario le haya pasado una *dirección IP o nombre de host*. En caso de *no pasar argumentos o pasarle alguno con caracteres peligrosos* se termina la ejecución. Si la entrada es *válida*, se ejecuta *ping* con el argumento provisto. El problema está en :

```
#!/usr/bin/sudo python3
```

Ya que se está ejecutando con privilegios elevados, es decir que un usuario podría escalar privilegios. El problema entonces es que **los permisos están mal**. La solución es implementar el **principio del mínimo privilegio**.

7.5 Denial of Service (DoS)

Hablamos un poco de esto ya. Es un ataque a la **disponibilidad** en donde el atacante busca **saturar los recursos** y que los usuarios no puedan acceder a los servicios que proporciona dicho sistema. Un ejemplo de esto es el **Fork Bomb** en donde el atacante crea un número masivo de procesos para agotar los recursos.



Fork Bomb en BASH

```
#!/usr/bin/env bash
:(){ :|: & };:
```

Fork Bomb en C

```
int main () {
    while (1) fork ();
    return 0;
}
```

Fork Bomb en ASM

```
_start:
    mov eax, 2      ; System call for forking
    int 0x80        ; Call kernel
    jmp _start
```

La solución a esto es **limitar la cantidad de procesos**

```
ulimit -S -u 5000
```

Este comando limita la cantidad de procesos que un usuario puede crear a 5000.

8 Mecanismos de protección del SO

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

- **DEP (Data Execution Prevention)** es una medida de seguridad que impide que ciertas áreas de la memoria (como la pila y el heap) se ejecuten como código. Su objetivo principal es prevenir que el código malicioso inyectado (por ejemplo, un exploit de buffer overflow) se ejecute en regiones de memoria donde no debería. Para lograrlo marca las regiones de la memoria como no ejecutables (NX - No eXecute). Si un atacante intenta ejecutar código desde esas regiones, el sistema lo bloquea y lanza

una excepción. El SO es el responsable implementando políticas para evitar que los datos almacenados en áreas de memoria no ejecutables se ejecuten como código.

- **ASLR (Address Space Layout Randomization)** es una técnica que aleatoriza las direcciones de memoria donde se cargan las bibliotecas, la pila, el heap y el código ejecutable de un programa. Esto hace que sea mucho más difícil para un atacante adivinar las ubicaciones de los datos y las direcciones de retorno en la memoria, lo que dificulta la explotación de vulnerabilidades de desbordamiento de búfer o de la ejecución de código. Para lograrlo, cada vez que un programa o servicio se ejecuta, las direcciones de memoria en las que se cargan sus componentes se randomizan, dificultando la creación de exploits predictivos. Lo gestiona el SO y podemos ver si está activado si el valor es 2 :

  ```
  cat /proc/sys/kernel/randomize_va_space
  ```

- **Stack Canaries** también conocido como **Stack Guards** o **Stack Cookies** son una técnica para detectar y mitigar ataques de desbordamiento de pila. El sistema coloca un valor especial (un "canario") en la pila, justo antes de la dirección de retorno de una función (IP). Si un atacante intenta sobrescribir la dirección de retorno mediante un desbordamiento de búfer, también sobrescribe el canario. Al regresar de la función, el valor del canario se verifica. Si el canario ha sido alterado, significa que un desbordamiento de búfer ha ocurrido y el programa se detiene inmediatamente para evitar la ejecución de código malicioso. La inserción y verificación del canario la lleva a cabo el **compilador**

  ```
  gcc -fstack-protector-all -o programa programa.c
  ```