

UNIVERSIDAD NACIONAL DE INGENIERÍA

Sistemas Operativos

Seguridad

Autenticación, control de acceso y criptografía

Gerson Garrido

Semana 16

Hoy

1. Metas de seguridad, modelo de amenazas y el monitor de referencia
2. Autenticación: contraseñas con hash y salt, multifactor
3. Control de acceso: ACLs, capabilities y los permisos de Unix
4. Explotación y endurecimiento: overflows, defensas, TOCTOU
5. Criptografía y seguridad en red: firmas, certificados, TLS
6. Cierre del curso

> **Lectura**
>
> OSTEP cap. 52–55 (sesión A). La sesión B es explotación y endurecimiento; criptografía y seguridad distribuida (cap. 56–57) quedan de lectura autónoma.

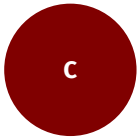
SECCIÓN 01

Metas de seguridad

01

¿Por qué le toca al sistema operativo?

- El kernel controla todo: memoria, archivos, red, dispositivos. Cualquier protección de las aplicaciones descansa sobre él.
- Llevamos 15 semanas construyendo *aislamiento*: modo usuario/kernel, tablas de páginas, locks. Hoy lo miramos como seguridad.



Confidencialidad
solo quien debe puede leer

Integridad
solo quien debe puede modificar



Disponibilidad
(availability)
el servicio responde

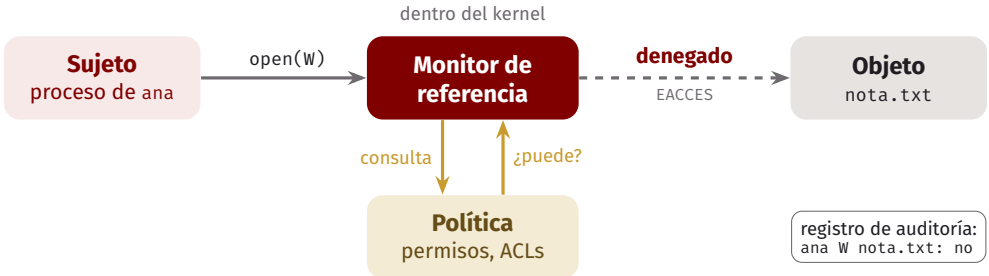
Modelo de amenazas

Pregunta	Ejemplo: una computadora compartida del laboratorio
¿Qué se protege?	Los archivos de cada alumno, las contraseñas, que la máquina funcione.
¿Quién ataca?	Otro alumno con cuenta propia; un programa descargado.
¿Qué puede hacer?	Ejecutar código como usuario normal, adivinar contraseñas.
¿Qué *no* puede?	Sacar el disco. (Si pudiera, habría que cifrarlo.)

Política y mecanismo, otra vez

La **política** dice qué se permite (“solo Ana lee sus notas”); el **mecanismo** lo hace cumplir. Un mecanismo perfecto con una política equivocada no protege nada.

El monitor de referencia



Tres exigencias (Anderson, 1972): se consulta en **cada** acceso *(complete mediation)*, no se puede alterar y es tan pequeño que se puede verificar.

La base de cómputo confiable (TCB)

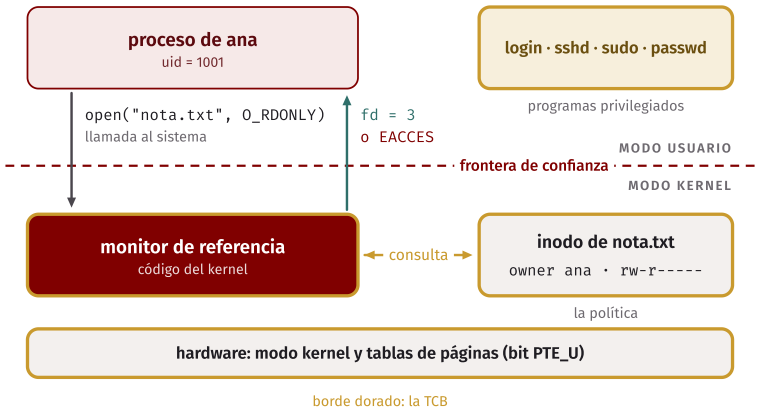


TCB ***(trusted computing base)***

Todo lo que *tiene* que funcionar bien para que se cumpla la política. Un error ahí rompe la seguridad de todos.

- Las aplicaciones quedan fuera: si fallan, el daño se limita a su usuario.
- Linux tiene decenas de millones de líneas. Por eso importa que cada programa privilegiado sea pequeño y haga poco.

La frontera de confianza



Principios de diseño (Saltzer y Schroeder, 1975)

1. **Economía de mecanismo**: simple y pequeño, para poder revisarlo.
2. **Valores por defecto seguros**: sin permiso explícito, se niega.
3. **Mediación completa**: se verifica *cada* acceso.
4. **Diseño abierto**: el atacante conoce el diseño; el secreto está solo en las claves.
5. **Separación de privilegios**: exigir dos condiciones.
6. **Mínimo privilegio**: lo justo, y solo mientras se necesita.
7. **Mínimo mecanismo común**: compartir poco entre usuarios.
8. **Aceptabilidad**: si es incómodo, la gente lo evita.

Los van a reconocer hoy

Claves privadas con `chmod 600` (2); dos factores de autenticación (5); un programa `setuid root` que solo necesita abrir un archivo (6, violado).

En xv6: la seguridad es el aislamiento

kernel/riscv.h y kernel/vm.c

```
#define PTE_V (1L << 0) // valid
#define PTE_U (1L << 4) // user can access

// walkaddr(): solo páginas de usuario
pte = walk(pagetable, va, 0);
if (pte == 0)
  return 0;
if ((*pte & PTE_V) == 0)
  return 0;
if ((*pte & PTE_U) == 0)
  return 0;
```

- Sin `PTE_U`, el hardware no deja que el modo usuario toque la página: fallo de página.
- El trampoline está en todo espacio de direcciones, pero *sin* `PTE_U` (`kernel/proc.c`).
- `exec` deja una página de guarda bajo la pila: `uvmclear` le quita el `PTE_U`.
- La MMU es el monitor de referencia de la memoria: revisa *cada* acceso.

En xv6: no confiar en los punteros del usuario

kernel/syscall.c

```
1  int
2  fetchaddr(uint64 addr, uint64 *ip)
3  {
4    struct proc *p = myproc();
5    if (addr >= p->sz ||
6        addr + sizeof(uint64) > p->sz)
7      return -1;
8    if (copyin(p->pagetable, p->sz,
9               (char *)ip, addr,
10              sizeof(*ip)) != 0)
11     return -1;
12   return 0;
13 }
```

- El kernel corre con su propia tabla de páginas: el puntero del usuario no significa nada ahí.
- `copyin`/`copyout` (`kernel/vm.c`) lo traducen *en software* con `walkaddr`: solo páginas `PTE_U`.
- `copyout` rechaza páginas sin `PTE_W`.
- `argstr`: el nombre de `open` se copia con tope `MAXPATH`.
- Si el kernel confiara en el puntero, un programa pasaría a `read` la dirección de una estructura del kernel y la sobrescribiría.

SECCIÓN 02

Autenticación

02

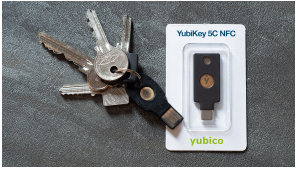
¿Quién eres? Tres tipos de prueba

Algo que sabes
contraseña, PIN

Algo que tienes
celular, llave USB

Algo que eres
huella, rostro

- El kernel no autentica: `login` o `sshd` comprueban la contraseña y crean un proceso con el `uid` correcto. Desde ahí, el uid es la identidad.
- **Multifactor** *(MFA)*: dos tipos *distintos*. Ejemplo: un código de 6 dígitos que el celular calcula con una clave compartida y la hora *(TOTP)*.
- Dos contraseñas no son dos factores: son el mismo tipo de prueba.



Llave de seguridad USB en un llavero. Daniel Aleksandersen, CC BY 4.0, Wikimedia Commons.

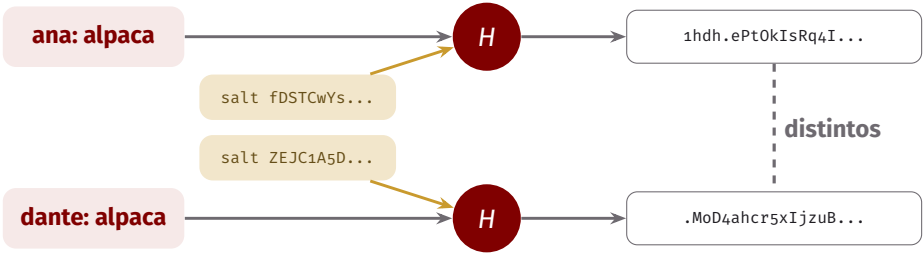
¿Cómo guardar las contraseñas?

	Se guarda	Si roban el archivo…
1	`alpaca`	Tienen todas las contraseñas.
2	$H(\texttt{alpaca})$	Calculan H del diccionario una vez y comparan con todos.
3	$s, H(s \,\|\|\, \texttt{alpaca})$	Repiten el diccionario para cada usuario.
4	$s, H_{\text{lento}}(s \,\|\|\, \texttt{alpaca})$	Además, cada intento cuesta milisegundos.

Hash criptográfico H

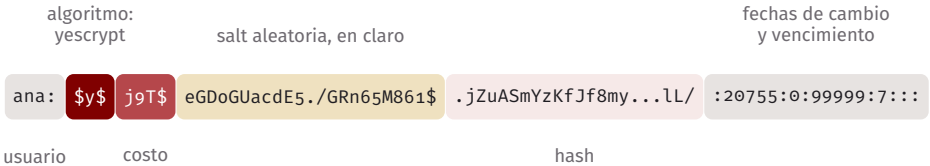
De un solo sentido: fácil de calcular, imposible en la práctica de invertir. Para verificar no hace falta invertirlo: se calcula H de lo que escribió el usuario y se compara.

Salt: la misma contraseña, hashes distintos



La salt *(sal)* es aleatoria, distinta por usuario y se guarda en claro junto al hash. No es secreta: su trabajo es que cada usuario sea un problema aparte.

`/etc/shadow` por dentro



- Línea real del contenedor: `echo 'ana:alpaca' | chpasswd` y `getent shadow ana`. PAM usa yescrypt (`/etc/pam.d/common-password`). El prefijo `$6$` sería SHA-512 iterado.
- `/etc/passwd` es legible por todos y solo dice `x`. Los hashes están en `/etc/shadow`: `rw-r-- root:shadow`. Ocultarlos es una defensa más, no la principal.

`crypt(3)`: calcular y verificar

hashpw.c (resumido)

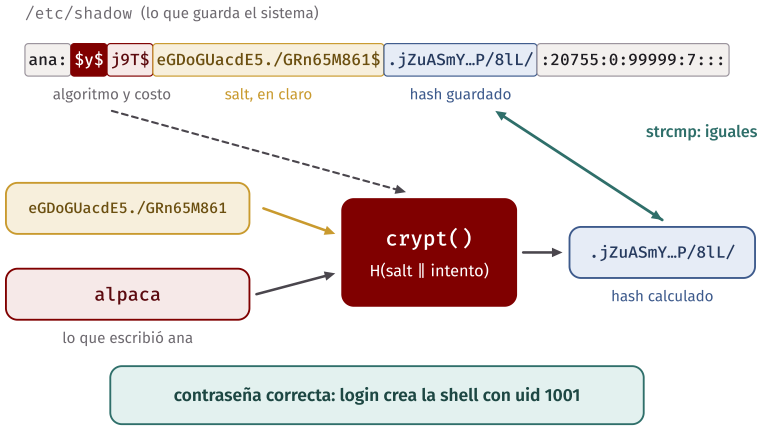
```c
char *salt = crypt_gensalt("$y$", 0, NULL, 0);
strcpy(stored, crypt(pw, salt));  // guardar
char *h = crypt(intento, stored);
if (strcmp(h, stored) == 0)
    puts("correcta");
```

- Se enlaza con `-lcrypt`.
- El prefijo (`$y$`, `$6$`) elige el algoritmo.
- Para verificar, el hash guardado hace de salt: trae algoritmo y salt.
- Nunca se “descifra”: se recalcula y se compara.

./hashpw alpaca

```
yescrypt $y$j9T$2nCAUFqI1gJ22SdN.kN.d/$iOYsopXGtufsZX1w30po9HAZ8UZau35Qa3gW.jmX2o.
yescrypt $y$j9T$rYhnGsEOcZgTYfuyrrWJo.$LZjABUb0drLfh0SOS7ej6TdQ3LVKFiIwat0JtUkEho4
verificar "alpaca": correcta
verificar "Password": incorrecta
```

Verificar no es descifrar



Ataque de diccionario: ¿cuánto cuesta?

```
./crack

== sin salt ==
  ana     -> alpaca
  dante   -> alpaca
  llamadas a crypt: 12
== con salt ==
  llamadas a crypt: 38
```

$|D| = 10^6$ palabras, $|U| = 10^4$ usuarios:

	Hashes	**1 núcleo**
Sin salt, hash rápido	10^6	$\approx$ 0,1 s
Con salt, hash rápido	10^{10}	$\approx$ 17 min
Con salt, SHA-512 crypt	10^{10}	$\approx$ 4 meses
Con salt, yescrypt	10^{10}	$\approx$ 3 años

Rápido: $\sim 10^7$/s. En el contenedor: SHA-512 crypt 1,1 ms y yescrypt 8,8 ms por llamada.

La salt no salva a una contraseña que está en el diccionario: con yescrypt, un usuario cae en $10^6 \times 8{,}8$ ms $\approx 2{,}5$ horas.

Comparar sin delatarse: tiempo constante

timing.c

```c
// insegura: corta al primer fallo
for (int i = 0; i < LEN; i++)
    if (a[i] != b[i])
        return 0;
return 1;
// constante: siempre recorre todo
for (int i = 0; i < LEN; i++)
    diff |= a[i] ^ b[i];
return diff == 0;
```

Aciertos	Ingenua	Constante
0	2,1 ns	119,6 ns
16	17,7 ns	117,5 ns
32	32,9 ns	117,4 ns
48	48,1 ns	118,4 ns
63	73,7 ns	117,7 ns

Secreto de 64 bytes, $2 \cdot 10^6$ llamadas.

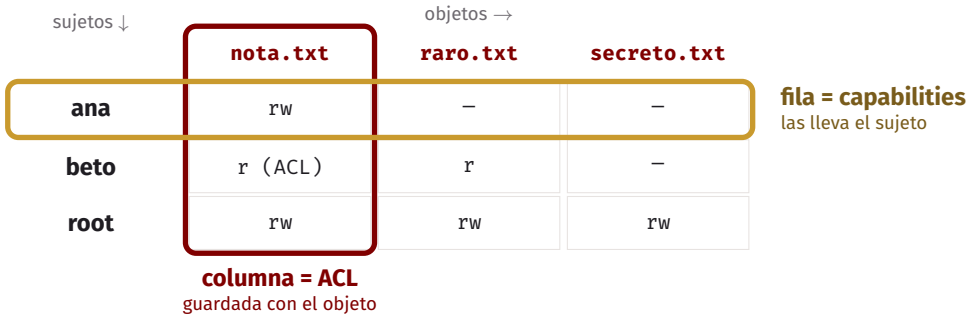
Un canal lateral *(side channel)*: el tiempo delata cuántos bytes acertó. Se adivina byte por byte, 64×256 intentos en vez de 256^{64}.

SECCIÓN 03

Control de acceso

03

La matriz de acceso



La matriz completa sería enorme y casi vacía. Nadie la guarda entera: se guarda por columnas o por filas.

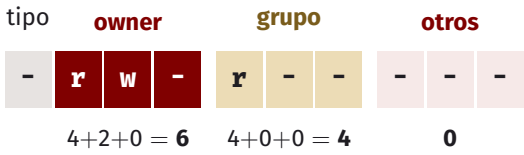
ACLs frente a capabilities

	ACL (el portero con la lista)	**Capability** (la entrada al cine)
Dónde vive	En el objeto (el inodo)	En el sujeto (el proceso)
¿Quién accede a X?	Fácil: leer la lista	Difícil: buscar en todos
¿A qué accede Ana?	Difícil: recorrer todo	Fácil: sus tokens
Revocar	Fácil: borrar la entrada	Difícil: ya se copió y entregó
Requisito	Saber *quién* es el sujeto	Que no se pueda falsificar

Unix mezcla las dos

`open` revisa la ACL (los bits) y entrega un descriptor, que actúa como capability: los `read` siguientes no revisan nada. Un `chmod` posterior no afecta a quien ya tiene el archivo abierto.

Los bits de permiso de Unix



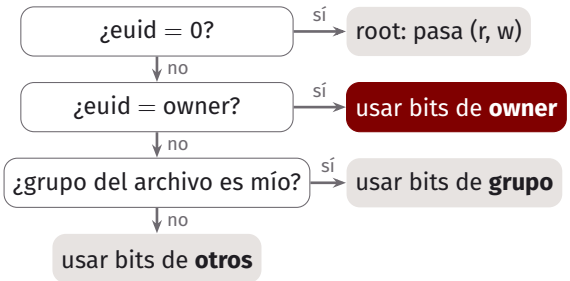

```
1 ana curso 16 nota.txt
```

- `chmod 640 nota.txt` o `chmod g-w,o= nota.txt`
- `chown ana:curso nota.txt` (solo root cambia el owner)
- En un directorio: `r` lista, `w` crea y borra, `x` permite *atravesarlo*.

`umask`: lo que se quita al crear

`open(..., 0666)` con `umask 022`:
666 & ¬022 = **644** → `rw-r--r--`

El algoritmo: se elige *una* clase



Ejemplo: **ana** quiere leer `raro.txt`:

`-r--r---  ana ana`

1. euid de ana $= 1001 \neq 0$.
2. El owner es ana: se usan los bits de owner y no se mira más.
3. Bits de owner: –. Denegado.

Beto, que no es owner, sí puede leerlo (otros: `r`).

No se busca “alguna clase que permita”: gana la primera que coincide. Así se puede excluir al owner o a un grupo.

`perms.c`: el algoritmo frente al kernel

perms.c

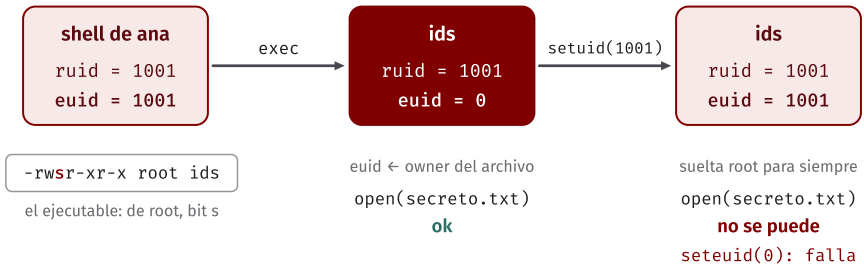
```c
uid_t uid = geteuid();
if (uid == 0)  // root: r y w siempre
    return want != 1 ||
           (st->st_mode & 0111) != 0;
int bits;
if (uid == st->st_uid)
    bits = (st->st_mode >> 6) & 7;
else if (getegid() == st->st_gid ||
         group_member(st->st_gid))
    bits = (st->st_mode >> 3) & 7;
else
    bits = st->st_mode & 7;
return (bits & want) == want;
```

bash

```
$ setpriv --reuid=ana \
    --regid=ana --init-groups \
    ./perms raro.txt
uid=1001 euid=1001 gid=1002
raro.txt  0044 ---r--r--
  leer     clase=owner
           nuestro=no kernel=no
```

`kernel` es la respuesta de `eaccess(2)`: coincide. Root solo ejecuta si hay *algún* bit `x`.

`setuid`: ejecutar con la identidad del owner



- **uid real** *(ruid)*: quién lanzó el proceso. **uid efectivo** *(euid)*: con quién se verifican los permisos.
- Así `passwd` (de root, con bit s) escribe `/etc/shadow` aunque lo ejecute ana. `setgid` hace lo mismo con el grupo.
- Mínimo privilegio: hacer lo que requiere root y soltarlo enseguida, de forma irreversible.

`ids.c`: verlo en el contenedor

ids.c (resumido)

```c
show("al empezar:", argv[1]);
if (setuid(getuid()) < 0) exit(1);
show("tras soltar:", argv[1]);
if (seteuid(0) < 0)
    printf("seteuid(0) falla\n");
```

Instalado como root:

```
chown root ids
chmod 4755 ids
```

`show` imprime `getuid()`, `geteuid()` y si pudo abrir el archivo.
`setuid(getuid())` suelta root para siempre.

bash

```
$ setpriv --reuid=ana --regid=ana --init-groups ./ids secreto.txt
al empezar:    uid=1001 euid=0  open(secreto.txt): ok
tras soltar:   uid=1001 euid=1001  open(secreto.txt): no se puede
volver:        seteuid(0) falla: los privilegios se perdieron
```

Por qué `setuid root` es peligroso

- Corre como root, pero sus entradas las controla otro usuario: argumentos, entorno, archivos.
- Un solo error (un buffer sin límite, un `PATH` confiado) y el usuario hace *cualquier cosa* como root.
- Cada programa `setuid root` es parte de la TCB: pocos y revisados. `find / -perm -4000` los lista.

sudo

Un `setuid root` que **autentica** al usuario, consulta una política (`/etc/sudoers`: quién ejecuta qué) y **registra** cada uso. Mejor que compartir la contraseña de root.

Pregunta de repaso

Un `setuid root` que solo necesita abrir un archivo viola el **mínimo privilegio**. Mejor: un grupo con `setgid`, o soltar root justo después del `open`.

ACLs POSIX: más allá de nueve bits

```
bash
$ setfacl -m u:beto:r nota.txt
$ getfacl -c nota.txt
user::rw-
user:beto:r--
group::r--
mask::r--
other::---
$ ls -l nota.txt
-rw-r-----+ 1 ana curso 16 nota.txt
```

- Entradas para usuarios y grupos *con nombre*.
- El `+` avisa que hay ACL. Los bits de grupo de `ls` ahora son la **máscara**: el tope de las entradas con nombre y del grupo.
- `perms` como beto:
  `leer clase=otros`
  `nuestro=no kernel=sí`

`perms.c` solo mira nueve bits; el kernel también revisa los *usuarios con nombre*. Moraleja: se pregunta al kernel (`access`), no se reimplementa.

Capabilities, DAC y MAC

Dos cosas con el mismo nombre

Capability: token infalsificable. El descriptor de archivo se parece: lo guarda el kernel, el proceso solo tiene un índice, y se hereda con `fork`.

Capabilities de Linux (`CAP_*`): el poder de root partido en unos 40 pedazos. `CAP_NET_BIND_SERVICE`: usar el puerto 80 sin ser root.

- **DAC** *(discretionary)*: el *owner* decide. Bits, ACLs.
- **MAC** *(mandatory)*: una política del sistema que ni el owner cambia. SELinux, AppArmor: "el servidor web solo lee `/var/www`".
- **RBAC** *(role-based)*: permisos por *rol* (administrador, docente, alumno).

Con MAC, aunque tomen el servidor web, el atacante sigue encerrado en lo que la política le permite: mínimo privilegio impuesto desde afuera.

En xv6: no hay usuarios

kernel/fs.h

```
1  struct dinode {
2    short type;    // File type
3    short major;
4    short minor;
5    short nlink;
6    uint size;
7    uint addrs[NDIRECT + 1];
8  };
```

- Ni owner ni bits en el inodo; `struct proc` no tiene `uid`.
- `sys_open` no verifica nada: cualquier proceso abre cualquier archivo.
- Lo único que protege xv6 es el aislamiento: `PTE_U` y `copyin`.

Para pensar: ¿qué haría falta? `uid` en `struct proc`; `uid` y `mode` en `dinode`; verificar en `sys_open`; una llamada `setuid`; un `login`.

SECCIÓN 04

Explotación y endurecimiento

04

Del `setuid` peligroso a cómo se rompe

- Vimos que un `setuid root` es peligroso porque sus entradas las controla otro usuario. Falta la pregunta concreta:
- ¿cómo una entrada mal manejada le da el control del programa? La respuesta explica *por qué* existen las defensas del SO.
- Todo aquí es **defensivo y local**: ningún ataque armado. El tope sería la idea clásica de laboratorio (saltar a una función `ganaste()` del mismo programa), que aquí no montamos.

Tres piezas

Buffer overflow en la pila · defensas (canario, NX/W^X, ASLR, PIE, CFI) · TOCTOU, la carrera entre el chequeo y el uso.

El marco de pila y el overflow



`strcpy(buf, entrada)` copia hasta el `\0`, sin medir `buf`. La copia va hacia direcciones altas: pisa el canario, el frame pointer y la dirección de retorno. Al retornar, la CPU salta adonde quedó.

`smash.c`: ver la dirección de retorno pisada

smash.c

```c
void vulnerable(const char *src){
  char buf[16];
  strcpy(buf, src);   // sin medir
}  // al retornar: salta a lo guardado
```

- Un manejador de señal informa adónde saltó la CPU. Sin ataque real.
- `0x41` es el ASCII de `A`: la dirección de retorno quedó llena de las letras copiadas.

sin canario (-fno-stack-protector)

```
$ setarch -R ./smash in_overflow
copie 30 bytes en un buffer de 16
[SIGBUS] la CPU intento ejecutar
          en 0x414141414141
```

con canario (-fstack-protector)

```
$ setarch -R ./smash_ssp in_overflow
*** stack smashing detected ***
```

Defensas: cada una rompe un paso

Defensa	Qué rompe	Cómo
Canario	Pisar la dirección de retorno linealmente	`-fstack-protector`
NX / W^X	Ejecutar código metido en la pila o el heap	bit `NX` en la PTE
ASLR	Saltar a una dirección fija	`randomize_va_space`
PIE	Direcciones fijas del propio programa	`-fPIE -pie`
CFI	Redirigir saltos y retornos a destinos no previstos	`-fcf-protection`

Se apilan: ninguna sola alcanza. Debajo de todas, la más barata: **mínimo privilegio** y poco código en la TCB.

ASLR y PIE: direcciones que cambian

./aslr (dos corridas, con ASLR)

```
codigo (main)  0xaaaac08d0d8d
heap  (malloc) 0xaaaace4a72a0
libc  (printf) 0xffffb7540590
---
codigo (main)  0xaaaacc7508d8
heap  (malloc) 0xaaab05c602a0
libc  (printf) 0xffffa8780590
```

- Casi todas cambian en cada corrida. `/proc/self/maps` las lista.
- `setarch -R ./aslr`: ASLR apagado, salen fijas (así se estudia un binario).
- `-no-pie`: `codigo` y `datos` quedan fijos (`0x4007e8`) aunque ASLR siga activo. PIE es lo que mueve también el código.

TOCTOU: una carrera entre el chequeo y el uso



Un `setuid root` comprueba con `access(ruta, R_OK)` (uid real) y luego `open(ruta, O_RDONLY)` (como root). Las dos resuelven la ruta por separado: si `ruta` es un enlace, el atacante lo reapunta en la ventana y `open` abre otro archivo.

TOCTOU: ganarla y cerrarla

make testhard (un bucle alterna el enlace y corre ./toctou vuln)

```
chequeo inodo=34297539  uso inodo=34297538  <-- TOCTOU
carreras ganadas: 122 de 300
```

El bucle alterna el enlace y se gana una fracción de las veces. “Chequear más rápido” no cierra la ventana.

La defensa: un solo paso, identidad correcta

Bajar el euid al uid real y dejar que `open` chequee (una syscall, sin ventana); `O_NOFOLLOW` contra el enlace. Luego, el descriptor.

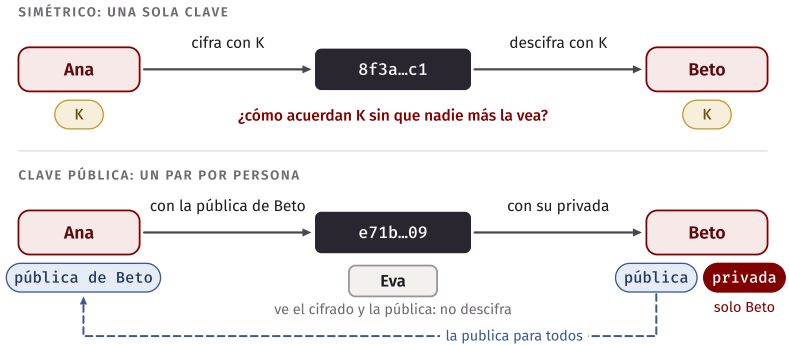
Otras clases, misma forma (una entrada sin validar cambia algo): **use-after-free**, **heap overflow**, **format string** (`printf(entrada)`), **integer overflow**.

SECCIÓN 05

Criptografía y redes (lectura autónoma)

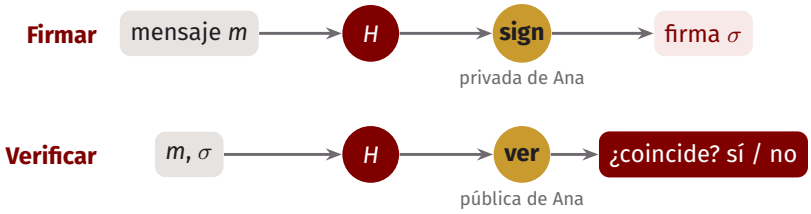
05

Cifrado simétrico y de clave pública



- En la práctica, **híbrido**: clave pública (RSA, curvas elípticas; lenta) para acordar una *K* de sesión, y simétrico (AES, ChaCha20; rápido) para los datos.
- Diseño abierto: algoritmos públicos y estudiados; el secreto es solo la clave. No inventen su propio cifrado.

Firmas digitales



- Integridad: si cambia un bit de m, la firma no verifica. Autenticidad: solo quien tiene la privada pudo firmar. Cualquiera verifica.
- Se firma el hash y no el mensaje: es corto y de tamaño fijo.
- Uso diario: paquetes de `apt`, actualizaciones, módulos del kernel.

Cifrado de discos y arranque seguro

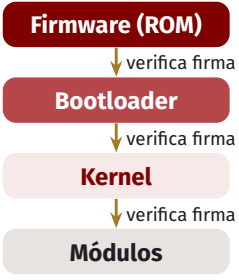
Cifrado de disco (LUKS, FileVault, BitLocker)

Cada bloque (o cada archivo, con fscrypt) se cifra con una clave simétrica. Esa clave se guarda cifrada con otra derivada de la contraseña (con un hash *lento*) o sellada en un chip TPM.

- Protege una laptop robada y apagada.
- No protege al sistema encendido: el kernel ya descifra para todos y mandan los permisos.

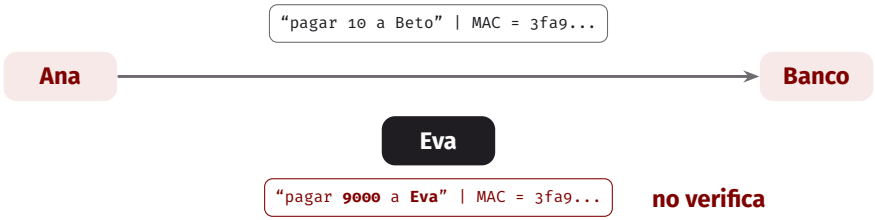


Módulo TPM. FxJ, dominio público, Wikimedia Commons.



Arranque seguro *(secure boot)*: una cadena de firmas desde la ROM. Un kernel alterado no arranca.

Autenticar mensajes en la red



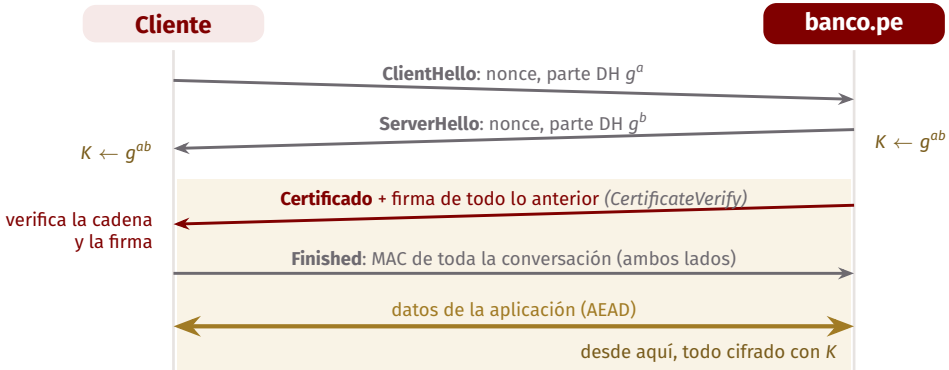
- **MAC** *(message authentication code)*: un “hash con clave” (HMAC). Sin la clave K, Eva no puede calcular el MAC de su mensaje.
- Cifrar no basta: se pueden alterar bits del texto cifrado a ciegas. Se usa **cifrado autenticado** *(AEAD)*: AES-GCM, ChaCha20-Poly1305.
- **Repetición** *(replay)*: Eva reenvía el mensaje auténtico diez veces. Defensa: números de secuencia o un *(nonce)* dentro de lo autenticado.
- **Autenticación remota**: la contraseña viaja solo dentro de TLS. Mejor, como `ssh`: el servidor manda un reto aleatorio y el cliente lo firma con su clave privada.

Distribuir claves: certificados

- Con claves simétricas, n usuarios necesitan $\frac{n(n-1)}{2}$: para $n = 1000$, 499 500.
- Con clave pública basta una por usuario. Pero ¿cómo sé que *esta* clave es de mi banco y no de Eva?
- **Certificado**: "la clave de `banco.pe` es *P*", firmado por una **autoridad de certificación** *(CA)*.
- Las CAs raíz vienen con el sistema: el contenedor trae 120 en `/etc/ssl/certs`. Quien agregue una ahí puede suplantar cualquier sitio: solo root la modifica.



TLS 1.3, simplificado



Diffie-Hellman acuerda K sin enviarla; la firma con la clave del certificado prueba que al otro lado está banco.pe y no Eva en el medio.

Cierre del curso: lo que construimos

Virtualización

procesos, `fork`/`exec`, traps
planificación: MLFQ, lotería
tablas de páginas, TLB, swap

Concurrencia

hilos, locks, variables de condición
semáforos, deadlock
`pzip`

Persistencia

discos, RAID, SSDs
inodos, FFS, journaling
archivos grandes en xv6

Seguridad: aislamiento por páginas, permisos sobre inodos, firmas en el arranque

La idea que atraviesa todo

Un recurso físico, una abstracción limpia encima y un kernel que media cada acceso. Mecanismo y política separados. Medir antes de opinar.

Resumen

- Metas: **confidencialidad, integridad, disponibilidad**, contra un **modelo de amenazas** explícito. El **monitor de referencia** media cada acceso; la **TCB** debe ser pequeña.
- Contraseñas: se guarda s y $H_{\text{lento}}(s \,\|\, p)$ (yescrypt en `/etc/shadow`). La salt obliga a atacar usuario por usuario; el hash lento encarece cada intento. MFA: dos tipos de prueba.
- ACL = columna de la matriz (fácil revocar); capability = fila (fácil delegar). Unix elige *una* clase: owner, grupo u otros. `setuid` cambia el euid.
- Buffer overflow: pisa la **dirección de retorno**. Defensas que se apilan: canario, NX/W^X, ASLR, PIE, CFI. TOCTOU: no separar chequeo y uso; bajar el privilegio y dejar que `open` chequee.
- Firma = hash + clave privada. Cadenas de firmas en el arranque seguro y en los certificados. TLS: DH para la clave, firma para la identidad, AEAD para los datos.
- xv6 no tiene usuarios: su seguridad es `PTE_U`, `copyin` y la frontera usuario/kernel.

Laboratorio, entregas y examen final

Laboratorio (en el contenedor, bajo `/tmp`)

1. Usuarios y permisos: `useradd`, `chmod`, `chown`, `umask`; el `setuid` de `ids.c` y su peligro.
2. ACLs POSIX con `setfacl`/`getfacl`: la máscara y el `+`.
3. `crypt(3)`: `hashpw.c` y `crack.c`, con y sin salt.
4. Endurecimiento: `make testhard` corre `smash`, `aslr` y `toctou`.
5. Repaso para el examen final.

Código en `code/week-16`; `make test` arma todo.

Entregas de esta semana

P5: archivos grandes y enlaces simbólicos.
Bono: `fork` con copy-on-write.

Examen final

En el periodo de exámenes. Entra todo el curso. Repasen las preguntas de cada guía semanal.