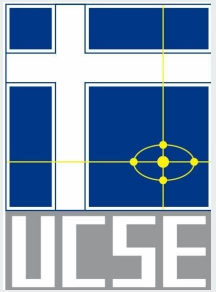
# Seguridad

UCSE - SEIA



# Cuánto hay para aprender de este tema?

- Mucho!.

- Es prácticamente un mundo aparte. Sus propias prácticas, libros, conferencias, carreras, herramientas, etc.

- Nuestro objetivo con esta clase es ver algunos temas básicos, pero muy importantes.

# General approach: handmade solutions!

- Problema: tengo que guardar la contraseña en algún formato que sea seguro, de forma que no sea interpretable por otros.

- Solución: Implementemos un algoritmos que le sume 3 al número de cada letra e invierta los últimos 3 elementos y después ...

# General approach: hand-make solutions!

# Seguridad por oscuridad (i)

Ejemplos:

- "Nadie puede desencriptar este paquete de red, porque nadie sabe cómo lo encripté".
- "Nadie puede acceder a este servidor, porque nadie sabe cómo lo configuré".
- "Nadie puede crackear este programa, porque nadie sabe cómo valido que sea original".

- Confiar en que algo es seguro, porque otros no saben cómo fue hecho "seguro".

# Seguridad por oscuridad (ii)

- Por lo general, es una muy mala idea. La práctica mostró siempre que si es importante, se termina descubriendo.
- Una buena práctica o algoritmo de seguridad, es seguro por más que el atacante sepa qué técnica estamos usando.
  Ej: crackear una key AES-256 usando 10^38 supercomputadoras Tianhe-2 (la 4ta más potente al 2019), tomaría más tiempo que la edad actual del universo. Y eso son muchísimas veces más que una supercomputadora por cada grano de arena en la Tierra.
- Los buenos algoritmos y prácticas de seguridad, son abiertos, estudiados y probados públicamente.
- Y por eso también es mala tu solución casera: no tiene cientos de expertos detrás. Y la oscuridad no te va a salvar de alguien decidido :)

# Encriptación simétrica vs. asimétrica (i)

**Simétrica**

A y B usan una misma clave, única, compartida, para encriptar y desencriptar sus mensajes. Ninguna otra clave funciona.

- Proceso simple
- Problema: tenemos que ponernos de acuerdo en la clave, que es un secreto pero tenemos que comunicarlo en algún momento (alguien puede atacar ese proceso).
- Alguien que accede a la clave, tiene conocimiento absoluto, y puede falsificar mensajes también.

# Encriptación simétrica vs. asimétrica (ii)

**Asimétrica**

Cada persona tiene un par de claves, una privada y una pública. Y el algoritmo garantiza que:

- Algo encriptado con una clave pública, sólo puede ser desencriptado con su clave privada correspondiente.
- Algo encriptado con una clave privada, solo puede ser desencriptado con su clave pública correspondiente.
- Proceso bastante más complejo
- Las claves privadas nunca se comparten!
- Igual tenemos que confiar en que conocemos la clave pública de alguien (pero no es un secreto!).

# Encriptación simétrica vs. asimétrica (iii)

**Extra (no se evalúa):** Cómo mandamos un mensaje con encriptación asimétrica?

Si A quiere mandarle algo a B, lo encripta dos veces (como una caja dentro de otra):

- Con la clave pública de B: entonces solo B puede desencriptar el mensaje (usando su privada).
- Y con su clave privada de A: entonces solo la clave pública de A va a poder desencriptarlo, lo que le asegura a B que el mensaje fue realmente escrito por A.
- Consecuencia: solo B puede leer el mensaje, y está seguro de que A fue el autor.

# Buenas prácticas y tecnologías específicas

## Guardado de passwords

- Encriptación?
- Las passwords no se encriptan!!!
- Las passwords se hashean.
- Un algoritmo de encriptación es de "ida y vuelta": puedo desencriptar el dato encriptado, usando la clave.
- Un algoritmo de hashing es de "solo ida": entra el dato, sale un hash. Distintos datos dan distintos hashes. Pero a partir de un hash, no puedo obtener el dato original. No hay forma.
- El dato de entrada para el algoritmo de hashing es más que solo la password. Normalmente se usa una semilla y data extra.

# Buenas prácticas y tecnologías específicas

**HTTPS**

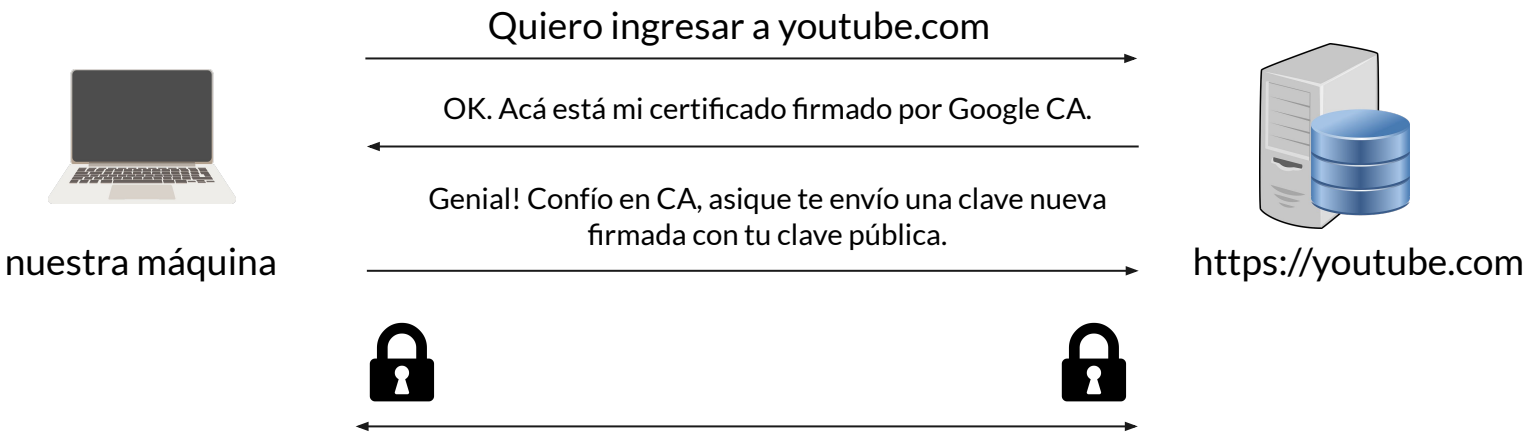
Por default, las requests y responses de HTTP son texto que viaja en paquetes de TCP.

Esto genera dos problemas gigantes:

- Privacidad: Cualquier intermediario haciendo de pasamanos del paquete, puede ver su contenido!. Y cualquier compañero de red wifi puede ver todos los paquetes que se están pasando por la red también!
- Autenticidad: Cualquier intermediario podría retener la request, armar una response, y devolvérsela. Cómo sé si fue Google el que me dio esta response, o fue Fibertel metiéndose en el medio? Le puedo dar data privada?

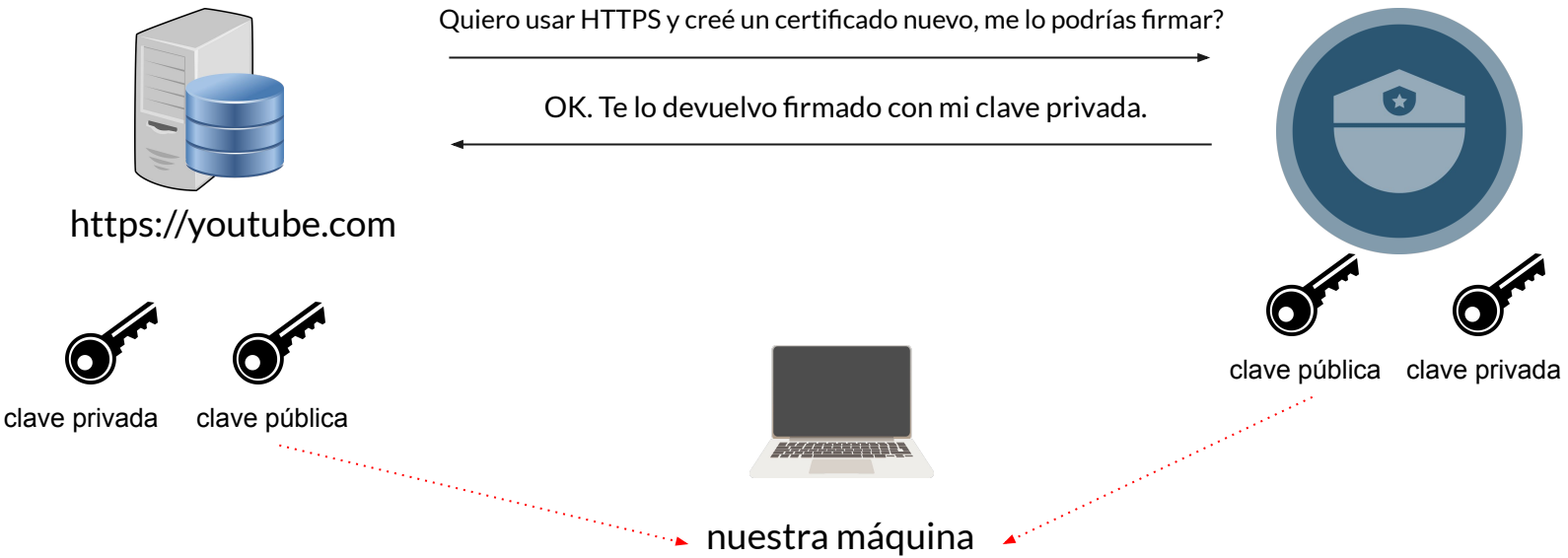
# Buenas prácticas y tecnologías específicas

**Solución: HTTPS.**



# Buenas prácticas y tecnologías específicas

## Solución: HTTPS.



# Buenas prácticas y tecnologías específicas

**Solución: HTTPS.**

- Privacidad: HTTPS encripta el contenido de las requests y responses, por lo que los paquetes de red van a ser ilegibles para cualquiera que no sea el origen o destino de ellos.
- Autenticidad: HTTPS introduce un mecanismo de certificados que permiten validar si una response vino del dominio que dice venir. Un certificado no es más que un par de claves pública y privada, asociadas a un dominio en particular, que se usan para firmar las responses. Y además, existen entidades que certifican las claves públicas asociadas a cada dominio.
- Requiere tener los certificados de estas entidades en nuestra PC, guardados (los tiene el sistema operativo, se actualizan).
- Los certificados vencen, hay que renovarlos.

# Buenas prácticas y tecnologías específicas

**Problemas de HTTPS**

- ~~Costo del certificado~~ Ya no! Lets Encrypt (https://letsencrypt.org/) nos lo provee gratis y automatizado.
- Costo de procesamiento: firmar, encriptar y desencriptar son cosas pesadas.
- Podemos usarlo en algunas partes. Pero por lo general, es mejor asumir el costo y usar HTTPS en todo.

# Buenas prácticas y tecnologías específicas

## Ataque: SQL injection

Imaginemos un formulario para buscar contactos por email en nuestro sitio:

```
def buscar_contacto():
    query = "SELECT * FROM usuarios WHERE email = '" + form.email + "'"
    resultados = db.run(query)
    return resultados
```

email: _____

# Buenas prácticas y tecnologías específicas

## Ataque: SQL injection

```
def buscar_contacto():
    query = "SELECT * FROM usuarios WHERE email = '" + form.email + "'"
    resultados = db.run(query)
    return resultados
```

Qué pasa si el usuario rellena el form así?:

email: `bla' OR 'hola' = 'hola`

# Buenas prácticas y tecnologías específicas

## Ataque: SQL injection

```
def buscar_contacto():
    query = "SELECT * FROM usuarios WHERE email = '" + form.email + "'"
    resultados = db.run(query)
    return resultados
```

Y si lo completa así?:

email: `email: bla'; DROP TABLE usuarios; SELECT 'hola`

O así?:

email: `email: bla'; UPDATE usuarios SET admin = True WHERE email = johndoe@gmail.com'; SELECT 'hola`

# Buenas prácticas y tecnologías específicas

**Solución: NUNCA concatenen input de usuarios con SQL.**

Usen consultas preparadas! Todos los motores de bases de datos lo soportan.

```
def buscar_contacto():
    query = "SELECT * FROM usuarios WHERE email = ?"
    resultados = db.run(query, form.email)
    return resultados
```

El motor sabe que lo que venga en los parámetros no es ejecutable, sino solo un valor (en este caso, lo trataría como un texto entero).

La mayoría de ORMs utilizan este enfoque para ir a la base de datos :)

# Buenas prácticas y tecnologías específicas

## Ataque: XSS (cross-site scripting)

Imaginemos que mostramos los comentarios de nuestros usuarios así:

```
<h3>{{ comentario.usuario.nombre }}: {{ comentario.fecha }}</h3>
<p>{{ comentario.texto }}</p>
```

Qué pasa si un usuario postea este comentario?:

```
Hola!
   <script>
       while (True) {
           alert("aguante river!");
       }
   </script>
```

O peor: puede estar por ejemplo extrayendo datos, o redireccionando a otro sitio, o impidiendo que el usuario haga algo, etc.

# Buenas prácticas y tecnologías específicas

**Solución:** Todos los frameworks tienen alguna herramienta para sanitizar input de usuarios al mostrarlo dentro del HTML.

Muchos lo hacen por default (Django por ejemplo), y pueden desactivarlo sin confían en algún input en particular.

No intenten sanitizar a mano. Es un problema complejo, usen herramientas ya probadas por la comunidad.

# Buenas prácticas y tecnologías específicas

## Ataque: XSRF o CSRF (cross-site request forgery)

Imaginemos que en nuestro sitio permitimos que los usuarios posteen links en sus comentarios, poniendo la url y texto que quieran.

Qué sucedería si un atacante hace esto?:

comentario: <a href="http://altaspublis.com/espadasgratis.php">Regalamos espadas a todo el mundo !</a>

El sitio del atacante muestra un form que pide "cuántas espadas querés gratis?", pero además tiene algunos campos ocultos con otros datos, y es un form que hace POST a la url nuestrositio.com/postear_comentario.

Qué va a suceder cuando la gente haga click en su comentario, rellene ese form, y haga click en el submit del form?

No queremos que cualquiera pueda decirle al navegador "hacé este POST al sitio, usando la cookie de sesión del usuario", sin que el usuario haya pedido un form y posteado él mismo.

Queremos que solo se pueda postear cuando el usuario realmente estaba viendo un form nuestro!

# Buenas prácticas y tecnologías específicas

**Solución:** Siempre que generamos un form para que el usuario postee, creamos junto un token, lo incluímos en el form y lo guardamos también en la db como "pendiente de postear".

Cuando el usuario postea nos llega su request con el token dentro del form: validamos el token y lo borramos de la db.

Si nos llega un post sin token, o con un token ya usado, rechazamos la request, es un atacante!

# Buenas prácticas y tecnologías específicas

## Ataque: Explotación de vulnerabilidades

Todo el software que usemos, va a tener vulnerabilidades: sistema operativo, servidor web, base de datos, lenguaje de programación, frameworks y libs externas, etc.

Esas vulnerabilidades se publican cuando se descubren. Es muy fácil ir a ver la lista de vulnerabilidades conocidas de la mayoría de las cosas.

Por eso es muy fácil para un atacante aprovechar las vulnerabilidades conocidas, si sabe qué software estamos usando.

# Buenas prácticas y tecnologías específicas

**Solución:** Dar la menor cantidad de información posible.

Que las requests no digan qué server y versión usamos. Que el html no diga qué framework y versión lo generó. Que si ocurre un error, no se le muestre al usuario la página de debug con toooda la info interna de la aplicación. Etc.

"No es seguridad por oscuridad????"

No. Porque nuestra seguridad no se "basa" solo en que eso no se sepa. Solo estamos complicando un poco más lo que un atacante tiene que hacer (va a tener que probar más cosas, adivinar, etc).

# Almacenamiento de secrets

Nuestro código va a tener que usar data secreta:

- Tokens de OAuth
- Semillas para los algoritmos de hashing
- Usuario y password de la base de datos
- Usuario y password de la cuenta de mail usada para los correos del sitio
- ...

Pero si alguien logra acceder al repo de código? O si el código lo estamos publicando como software libre??

O si no queremos que los programadores puedan acceder a la data privada de los usuarios?? (ej: google y tus emails)

Por lo general, no es buena idea que esos datos secretos importantes estén guardados en el código.

# Almacenamiento de secrets

**Solución:** Hay herramientas específicas para administrar secrets en producción (especies de bases de datos para esa data sensible).

Nuestro código usa valores por default falsos, que andan solo en desarrollo, pero intenta obtener datos del entorno de producción para reemplazarlos. Cuando corre en local, no ve esos datos, usa los valores por default. Cuando corre en producción, tiene acceso a esos datos, y usa esos valores reales.

# Resumen

- Descartar soluciones caseras, por más buenas que nos parezcan.
- Tomar este tema en serio, ponerlo en carpeta !
- Cuanto más sepamos, menos expuestos vamos a estar a posibles ataques. Pero no hay garantías, "it's just another brick in the wall … " :)
- Buen recurso: https://owasp.org/ (top 10 vulnerabilidades ranking 2017)

# Seguridad

UCSE - SEIA