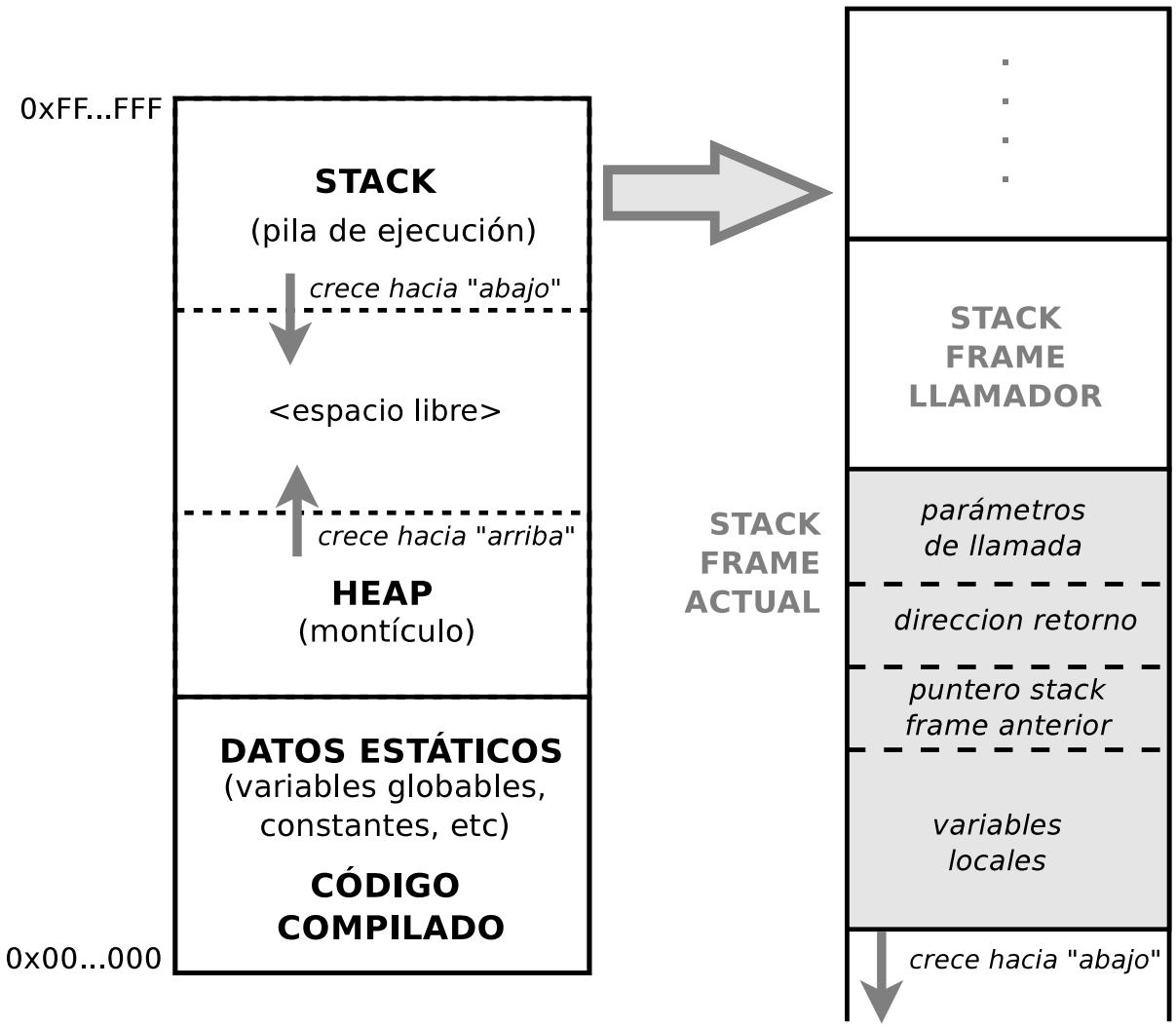
Tema 3. Seguridad en el desarrollo de aplicaciones

Seguridad en Sistemas Informáticos

Octubre-2019

Contenido

Previo: disposición de programas en memoria



Desbordamiento de buffer (I)

Causado por errores/descuidos de programación

- Típico en lenguajes que soportan acceso directo a memoria (C, C++)
  - Lenguajes modernos con validaciones de tipado fuertes son menos vulnerables
- Uso de funciones que no validan tamaño/contenido de los datos recibidos
  - Ejemplos: `gets()`, `strcpy()`, `strcat()`, `...`

Se permite almacenar datos superando la capacidad disponible para un buffer de tamaño fijo

- Buffer puede ubicarse en zona de stack, heap o variables globales
- Datos pueden provenir de entradas de consola, conexiones de red, ficheros, etc

Desbordamiento de buffer (II)

Hace posible sobreescribir regiones de memoria adyacentes al buffer

- Corrupción de los datos del programa
- Transferencia de control no prevista
- Violación en privilegios de acceso a memoria
- Ejecución de código arbitrario inyectado por el atacante

Vulnerabilidad ”clásica” y bien conocida

- Mecanismo clásico de explotación en intrusiones, virus, etc
- Disponibles técnicas de prevención/atenuación
- ”Latente” en código heredado (*legacy code*)
- Aparecen ocasionalmente en nuevas aplicaciones

Posible (aunque más complejo) desbordamiento de buffer en zona de *heap* o sobre buffers en variables globales (sección de código suele ser de sólo lectura)

Desbordam. de buffer en pila (I)

Desbordamiento de pila (*stack buffer overflow*) [también *stack smashing*]

- Esquema más habitual para aprovechar desbord. de buffer

Idea base:

- ”Desbordar” variables locales situadas ”debajo” de la dirección de retorno de la función
- Los datos que ”desbordan” el buffer sobreescriben la dirección de retorno
- Se escribe una nueva dirección de retorno a la que se saltar al finalizar la ejecución de la función actual
  - dirección dentro de código malicioso cargado por el atacante en el buffer desbordado
  - dirección de una librera del sistema (funciones `system()`, `execv()`)
  - dirección de otro programa en memoria

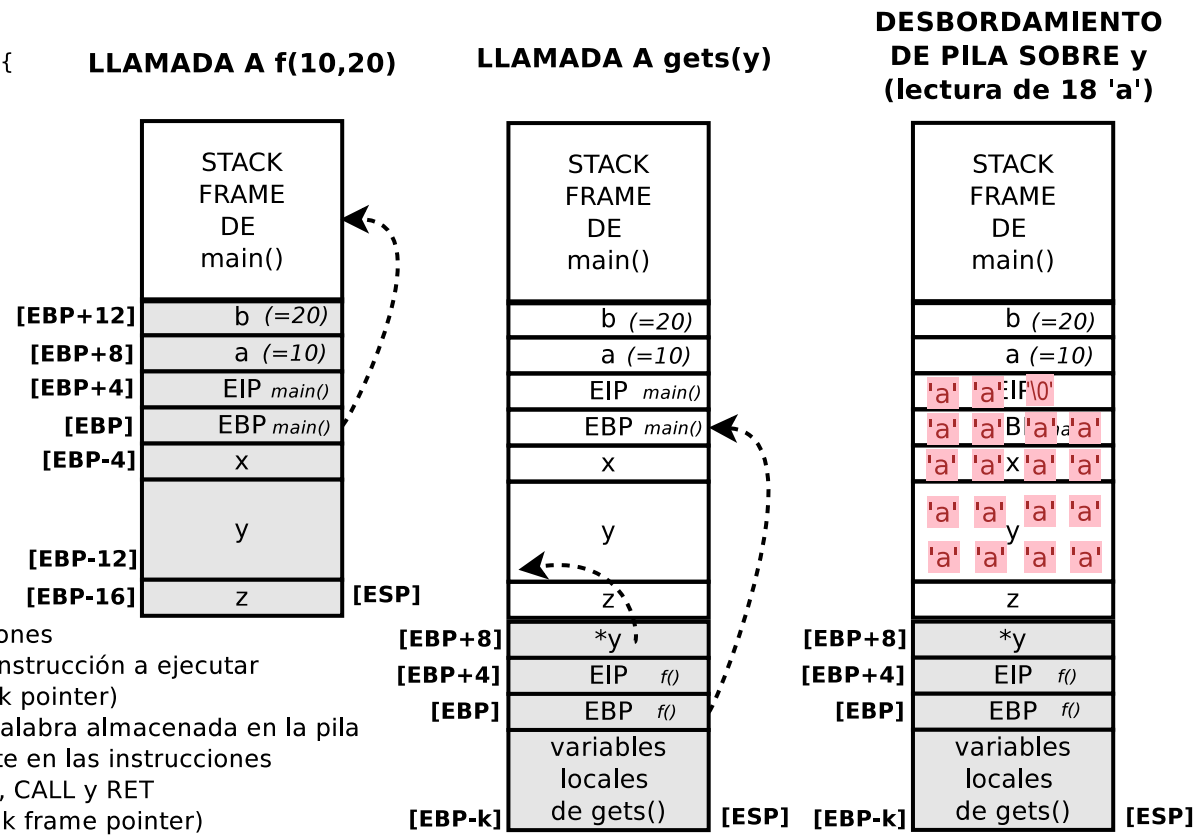
Esquema desbord. de pila en arquitecturas x86 (32 bits)

```
void main(int argc,
          char** argv) {
    int valor;
    valor = f(10,20);
}

int f(int a, int b) {
    int x;
    char[8] y;
    int z;

    x = a + b;
    gets(y);
    z = x + strlen(y);

    return z;
}
```



REGISTROS

EIP: contador de instrucciones
- apunta a siguiente instrucción a ejecutar

ESP: puntero de pila (stack pointer)
- apunta a la última palabra almacenada en la pila
- usado implícitamente en las instrucciones máquina PUSH, POP, CALL y RET

EBP: puntero "base" (stack frame pointer)
- apunta(identifica) al stack frame de la función actual (apunta al EBP de la función llamadora)

Desbordam. de buffer en pila (II)

Para explotar un desbordamiento de bufer en pila, atacante debe

1. identificar una vulnerabilidad de desb. de buffer en un programa (inspección/auditoría del código, traza de ejecución en debugger, ”probar” entradas inesperadas *fuzzing*)
2. conocer cómo se organiza el buffer y la pila en memoria para llevar a cabo la explotación
   - identificar direcciones destino del salto
   - componer *shellcode*/payload a inyectar

Efecto perseguido: ejecución de código arbitrario aportado por atacante

- Normalmente *”shellcode”* que proporciona intérprete de comandos al atacante (local o remoto)
- Objetivos: demonios del sistema, utilidades del sistema, liberas
- Código máquina inyectado específico para cada arquitectura (CPU) y SO
  - independiente de su posición (dirs. relativas)
  - uso de NOPs para ”facilitar” ubicación destino del salto
- Herramientas automatizan creación de *exploits* (ej. Metasploit)
  - diversidad de *exploits*: código para explotar vulnerabilidades conocidas
  - diversidad de *payloads*: lanzar *shell*, conexin inversa al atacante, lanzar Meterpreter

Contramedidas (I)

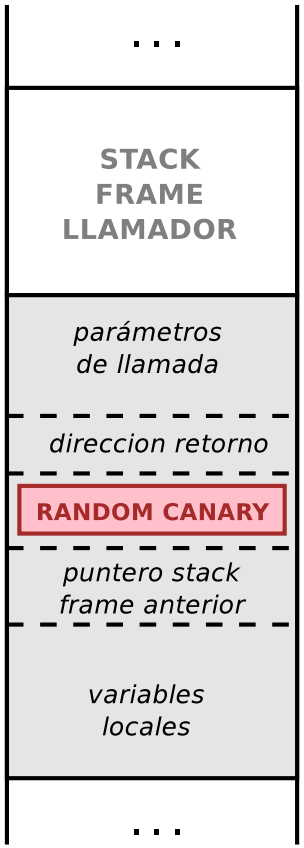
(1) En tiempo de desarrollo/compilación

Objetivo: fortalecer nuevos programas

- Uso de lenguajes con tipado fuerte
  - menos vulnerables a desbordamiento de buffer
  - compilador fuerza comprobaciones de rangos y de operaciones admitidas sobre las variables en función de su tipo
- Fomentar que programadores desarrollen software seguro (buenas prácticas)
  - restricciones de seguridad incluidas en requisitos y diseño
  - revisión/auditoría del código desarrollado (y del antiguo)
- Uso de versiones seguras de funciones sin control de desbordamiento
  - `fgets()`, `strncpy()`, `strncat()` limitan nº máximo de bytes escritos
  - uso de librerías ”seguras”

Contramedidas (II)

- Compiladores que incluyan código adicional en las secuencias de llamada y retorno para detectar alteraciones en la pila



- ”canarios aleatorios” (*random canaries*) : valores aleatorios colocados en las *stack frames* como ”separadores” entre variables locales y datos de control (dirección de retorno)
- la secuencia de retorno generada por el compilador comprueba si al retornar de una llamada a función, estos ”canarios” fueron sobreescritos por un desbordamiento, abortando el programa
- Ejemplos: StackGuard en GCC, opción /GS en MS Visual Studio

Contramedidas (III)

(2) En tiempo de ejecución

Objetivo: gestión de ataques en programas existentes

- En CPUs y S.O. con soporte para protección de ejecución de datos (*non executable address space*)
  - impide ejecutar código que provenga de direcciones en porciones de memoria marcadas como ”de datos” (no ejecutables)
  - marcando la zona de *stack* como sólo datos se impide ejecutar el código inyectado por el atacante
  - no protege contra ataques que sobreescriban la dir. de retorno para llamar a una función de la librería del sistema `libc` como `system()` o `execv()` (ataques *return to libc*)
  - programador debe configurar compilador para que marque las regiones de *stack* y *heap* del programa como no ejecutables
  - En MS Windows: *Data Execution Prevention(DEP)* configurable.

Contramedidas (IV)

- Aleatorización de posiciones relativas de stack, heap y código al cargar un ejecutable en memoria (*address space layout randomization*)
  - manipular (si lo soporta el S.O.) la localización de las porciones del programa *stack*, *heap* y *datos estáticos*, usando desplazamientos aleatorios diferentes en cada ejecución del programa
  - aleatoriza la ubicación de buffers y de las funciones de las librerías estándar
  - dificulta que atacante puede ”encontrar” una dirección de destino para sobreescribir el puntero de retorno
- Uso de IDS (*intrusion detection systems*) que analicen y detecten tráfico con firmas sospechosas de intentos de desbordamiento de buffer (secuencias NOPs, fragmentos de shellcode, etc)

Contramedidas (V)

- Seguimiento de alertas de vulnerabilidades e instalación de parches, actualizaciones y correcciones para el software instalado
- Bases de datos de vulnerabilidades
  - National Vulnerability Database (`http://nvd.nist.gov/`)
  - Common Vulnerabilities and Exposures (`http://cve.mitre.org/`)
    - Interfaz alternativo: `http://www.cvedetails.com/`
  - Common Weakness Enumeration (`http://cwe.mitre.org/`)
  - Security Focus (`http://www.securityfocus.com/bid`)
  - Open Source Vulnerability Database (`http://osvdb.org/`)
  - Computer Security Vulnerabilities (`http://securityvulns.com/`)

1. Vulnerabilidades software
   - Desbordamiento de buffer
   - Contramedidas desbordamiento de buffer

2. Vulnerabilidades WEB
   - OWASP top 10 de vulnerabilidades WEB
   - Inyección SQL
   - Cross Site Scripting

OWASP Top Ten (I)

OWASP (*Open Web Application Security Project*): proyecto abierto sin fines de lucro destinado a mejorar la seguridad de las aplicaciones (fundamentalmente web)

- Todo el material distribuido con licencias libres (publicaciones, normas, software, ...)
- Herramientas y documentos organizados en 3 categorías
  - **Protect.** protección contra debilidades de implementación y/o diseño
  - **Detect.** localización de fallo de implementación y/o diseño
  - **Live cycle.** inclusión de actividades relativas a la seguridad en el ciclo de vida del desarrollo del software
- Principales aportaciones
  - Herramientas: WebGoat (herramienta de aprendizaje), WebScarab (chequeo de vulnerabilidades), etc
  - OWASP Application Security Verification Standard Project
  - OWASP Code Review Guide
  - OWASP Testing Guide
  - OWASP Top Ten Project

OWASP Top Ten (II)

OWASP Top Ten Project 2017: selección de las 10 vulnerabilidades más críticas de las aplicaciones web

- A1: Inyección (SQL, LDAP, comandos SO)
- A2: Deficiencias de autenticación y manejo de sesiones
- A3: Exposición de datos sensibles
- A4: Entidades XML Externas (XXE)
- A5: Deficiencias en el control de accesos
- A6: Deficiencias en las configuraciones de seguridad
- A7: *Cross-Site Scripting* (XSS)
- A8: Deserialización insegura
- A9: Uso de componentes con vulnerabilidades conocidas
- A10: Registro y monitorización insuficientes

Nota: Eliminados de Top Ten 2013

- A8-2013: Falsificación de peticiones entre sitios (CSRF: *Cross-Site Request Forgery*)
- A10-2013: Redirecciones y reenvíos sin validación

A1 Inyección (SQL, LDAP, comandos SO)

Diversos tipos de vulnerabilidades por inyección de código interpretable

- Inyección SQL
- Inyección LDAP
- Inyección XPATH
- Inyección de comandos del S.O.

Implica la manipulación de datos proporcionados por la aplicación cliente (parámetros URL, cookies, cabeceras HTTP, ...)

- Si el servidor no valida los datos recibidos y los pasa directamente a una aplicación externa (servidor BD, S.O., ...) se abre la posibilidad de que se inyecten comando arbitrarios
- Caso más habitual: Inyección SQL

Efectos: pérdida o corrupción de datos, acceso no autorizado, escalada de privilegios, ...

A2 Deficiencias de autenticación y manejo de sesiones

Defectos en los mecanismos de autenticación

- Difícil construir mecanismos de autenticación robustos y fiables
- Deficiencias en gestión de contraseñas, mecanismos de recuperación de contraseñas, preguntas secretas, ...
- Uso de identificadores de usuario y/o sesión en las URLs manejadas, etc
- Parámetros de autenticación de usuarios y sesiones fáciles de adivinar (predecibles)
- Defectos en la regeneración de credenciales de acceso

Defectos en la gestión de sesiones

- Ausencia o deficiencias en el mecanismo de logout
- Ausencia de timeouts o tiempo de vida de sesiones excesivo
- Cookies de sesión predecibles y/o en claro $\Rightarrow$ facilita ”robo de sesiones”

A3 Exposición de datos sensibles

Almacenamiento criptográfico inseguro

- Ausencia de cifrado de datos sensibles (en las peticiones o en el almacenamiento en BD)
- Deficiencias en el uso de la criptografía: algoritmos débiles, hashes sin *salt*, contraseñas simples (por defecto, predecibles o "de diccionario"), no renovación de claves, ...

Protección insuficiente en la capa de transporte

- Ausencia de protección del tráfico de peticiones y respuestas sensibles (credenciales de autenticación, contenidos privados, etc)
- Ausencia o uso inadecuado de SSL/TLS: certificados caducados, ausencia de protección de URL sensibles, mezcla de URLs SSL y no-SSL, etc

A4 Entidades XML Externas (XXE)

En aplicaciones o servicios web que acepten o descarguen directamente XML, existe la posibilidad de que el parser XML procese datos no fiables al tratar con XXE (*XML eXternal Entities*) convenientemente definidas.

Permite { denegación de servicio
acceso a servicios/archivos

A5 Deficiencias en el control de accesos

Fusión antiguos
- A4: Referencias directas a objetos inseguras
- A7: Ausencia de control en el acceso a funcionalidades

Referencias directas a objetos inseguras

- Aplicaciones vulnerables que dan acceso directo a recursos mediante URLs directas sin verificar si el usuario efectivamente está autorizado a acceder a ellos .

Ausencia de control en el acceso a funcionalidades

- Ausencia de controles (configuración del servidor, controles a nivel de aplicación/framework) que permitan al atacante acceder a URL privadas o ”ocultas” (consolas de administración, pantallas de configuración)
- Presencia de elementos y/o enlaces inactivos
- Ausencia/deficiencia de controles de autenticación en el lado del servidor
- El resultado es que usuarios anónimos o con menos privilegios acceden a recursos/funcionalidades de usuarios de más alto nivel

A6 Deficiencias en configuraciones de seguridad

- Uso de versiones no actualizadas de servidores o frameworks $\Rightarrow$ vulnerable ante problemas conocidos y ya resueltos
- Disponibilidad de consolas de instalación/administración de servidores, frameworks y/o aplicaciones (no protegidas o con contraseñas por defecto o débiles)
- Exposición de información del servidor: listado de directorios habilitado, páginas/URLs no usadas accesibles, componentes/módulos opcionales no utilizados accesibles, mensajes de error/depuración, etc

A7 *Cross-Site Scripting* (XSS)

- Caso particular de inyección
- Atacante consigue inyectar contenido interpretable (código HTML o Javascript) de forma temporal o permanente
- El contenido inyectado será enviado por el servidor e interpretado por el navegador de un usuario legítimo
  - el ataque lo ”sufre” el navegador del usuario final no el servidor donde corre la aplicación vulnerable
- Efectos: robo de sesiones y cookies, phising, redirección a sitios maliciosos, etc.

A8 Deserialización insegura

Deserialización de objetos (JSON, XML, binario) manipulados por atacante en servicios web y aplicaciones vulnerables

A9 Uso de componentes con vulnerabilidades conocidas

Fallos conocidos en servidores o componentes de frameworks que pueden ser atacados/explotados de forma automática

A10 Registro y monitorización insuficientes

Deficiencias en la monitorización y registro de actividades (autenticación, inicio de sesión, transacciones, errores, ...) permiten a atacantes logar sus objetivos sin ser detectados.

Vulnerabilidades de Top Ten 2013 eliminadas I

A8 (2013) Falsificación de peticiones entre sitios (CSRF)

CSRF (*Cross-Site Request Forgery*): Inyección de código HTML/JavaScript en lado del usuario (similar a XSS) para generar peticiones HTTP falsas

- Atacante genera (fabrica) peticiones HTTP falsas (replicando los parámetros de peticiones legítimas a una aplicación)
- Peticiones ilegítimas se difunden incluyéndolas en parámetros de imágenes, elementos `iframe`, etc, construidos por el atacante
- Un usuario legítimo que acceda a esos enlaces ”maliciosos” mientras está autenticado con la aplicación ”atacada” (sesión no cerrada/caducada) hará que la petición falsa/fabricada se presente con sus credenciales ante el servidor.
  - aplicación no puede diferenciar las peticiones legítimas de las falsas
  - todas proceden de un usuario logueado con una *cookie* válida

Vulnerabilidades de Top Ten 2013 eliminadas II

Soluciones CSRF:

- añadir info. vinculada a la sesión no predecible (y no repetible) [en ocasiones cifrada/firmada] en las peticiones de la aplicación (*tokens* en campos *hidden*, parámetros HTTP)
  - dificulta al atacante generar peticiones HTTP válidas
  - aplicación sabe que las peticiones legítimas llegan con un *token* válido
- exigir autenticación/confirmación adicional en operaciones delicadas (autenticación por doble factor)

A10 (2013) Redirecciones y reenvíos sin validación

- Redirecciones gobernadas por parámetros presentes en las URLs o peticiones HTTP $\Rightarrow$ atacante puede construir una URL/petición que redirija al usuario legítimo a una URL bajo su control (robo de credenciales/sesiones/cookies)
- Posibilidad de omitir procesos de autenticación creando redirecciones a URL no permitidas

Inyección SQL (I)

Inserción de sentencias/comandos SQL en datos de entrada

- SQL posibilita la comunicación de datos entre WEB y almacenamiento → generación dinámica de sentencias SQL usando los parámetros que proporciona el usuario
- Posibilidad de cambiar la naturaleza de la petición que va a ejecutar el servidor de BD
- Introducción de sentencias o porciones de sentencias SQL en la URL, en campos de formulario de una página o en parámetros HTTP que serán usados para generar sentencias SQL dinámicamente

Este ataque es posible cuando hay entradas de usuario que se envían directamente desde la aplicación Web a la BD

no validación datos recibidos (con SQL ”no previsto”) + uso directo de datos de usuario para construir consultas SQL mediante **concatenación** → altera semántica prevista de la consulta ejecutada por servidor BD

Objetivo: acceso no autorizado o no restringido a la base de datos (potencialidad de daño muy alta)

- Obtener información sensible contenida en una base de datos sin ser un usuario autorizado del sistema
- Modificar los datos almacenados
- Destruir la base de datos

Inyección SQL (II)

Ejemplo 1: login contra BD

- Parámetros entrada: nombre + clave
- Los parámetros de entrada no se comprueban

Código de login en el servidor (php)

```
$login    = $_POST["login"];
$password = $_POST["password"];

$query = "SELECT * FROM users";
$query .= "  WHERE user_login = '$login'";
$query .= "  AND   user_pass  = '$password'";
$result = $mysqli->query($query);

if ($result) {
    //Existe el usuario y la clave, acceder
}
else {
     //Acceso no permitido
}
```

Inyección SQL (III)

Si como nombre de usuario se escribe `' OR 1=1 ; #` la consulta real siempre devolvería resultados

```
SELECT * FROM users WHERE user_login = '' OR 1=1 ; # AND user_pass = ''
```

- La tautología `1=1` es siempre verdadera, por lo que anula la condición de la cláusula `WHERE`
- El servidor BD usa `'` como símbolo de cadenas y `#` como marca de comentario (anula la segunda condición del `WHERE`)

Posibilidad de ejecución de comandos SQL arbitrarios

- Concatenar nuevas sentencias SQL separadas por `;`
- Permite modificar los contenidos de la BD

En el caso anterior, usando como nombre de usuario

```
'; INSERT INTO users (userLogin,userPassword) VALUES ('atacante','123'); #
```

La sentencia SQL resultante sería:

```
SELECT * FROM users WHERE user_login = ' ';

INSERT INTO users (userLogin,userPassword) VALUES('atacante','123'); # AND user_pass =''
```

Inyección SQL (III)

Ejemplo 2: Acceso a datos mediante adición de `UNION SELECT`

- Parámetro entrada: id de producto en un buscador de productos
- Los parámetros de entrada no se comprueban

Código PHP de la consulta

```
$id = $_GET["id"];

$query = "SELECT nombre, descripcion, precio FROM productos";
$query .= "  WHERE id = $id";
$result = $mysqli->query($query);
if ($result) {
   //Mostrar nombre, descripción y precio del producto
}
```

Se introduce como valor del *query param* `id` de la URL la siguiente cadena:

```
999999 UNION SELECT login,password,null FROM usuarios; #
```

- El uso de un `id` inexistente anula la primera subconsulta
- Los únicos resultados a mostrar serán los valores del segundo `SELECT`.

Prevención Inyección SQL (I)

Fuente: `https://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet`

(1) Uso de consultas parametrizadas

Separa compilación de la consulta (generación del *query plan*) de su ejecución con los parámetros indicados.

- Servidor BD procesa la consulta y crea el plan de ejecución sin intervención de los datos proporcionados por el usuario
- Aplicación ”rellena” los parámetros de la consulta con los datos del usuarios
- El posible código SQL inyectado nunca se combinará con el de la consulta real para construir nuevas sentencias SQL (potencialmente peligrosas)

Ejemplos

- Java: objetos `PreparedStatement`

```
PreparedStatement stmt =
      connection.prepareStatement("SELECT * FROM users "+
                                  "WHERE USERNAME = ? AND PASSWD = ?");
stmt.setString(1, username);
stmt.setString(2, passwd);
stmt.executeQuery();
```

- `PDOStatement` en el framework de acceso a datos PDO *(PHP Data Objects)* de PHP

```
$dbh = new PDO("mysql:host=localhost;dbname=test","root","ssi");
$stmt = $dbh->prepare("SELECT * FROM users "+
                                  "WHERE USERNAME = ? AND PASSWD = ?");
$stmt->execute(array($_POST['username'], $_POST['paswd']));
```

Prevención Inyección SQL (II)

- PHP: con el API `mysqli` de acceso a MySQL (reemplazo de `mysql`)

```
$dbh = mysqli_connect("localhost", "root", "ssi", "test");
$stmt = mysqli_prepare($dbh, "SELECT * FROM users "+
                                    "WHERE USERNAME = ? AND PASSWD = ?``);
mysqli_stmt_bind_param($stmt, 'ss', $_POST['username'], $_POST['paswd']);
mysqli_stmt_execute($stmt);
```

Más ejemplos en otros lenguajes en `http://bobby-tables.com/`

Mayor robustez si se pasan parámetros "tipados"

- Código SQL inyectado que no conforme un dato del tipo esperado (entero, real, fecha) provocará un fallo/excepción
- Java: métodos `setInt()`, `setDAte()`, `setString()`, `setBigDecimal()` en `PreparedStatment`
- PHP: métodos `bindParam(...)`, `bindValue(...)` de `PDOStatement` en PDO
- PHP: método `mysqli_stmt_bind_param()` en el API `mysqli` de PHP

Prevención Inyección SQL (II)

(2) Uso de procedimientos almacenados Código de acceso a datos reside en el gestor BD en forma de procedimientos almacenados (ya compilados).

- Los datos manejados por la aplicación web se pasan a los procedim. almacenados
- Si están bien diseñados (no construyen consultas dinámicamente) no hay posibilidad de generar dinámicamente consultas SQL maliciosas

(3) Uso de frameworks ORM *(Object Relational Mapping)*

- Ejemplos: Hibernate ó JPA en Java, PDO en PHP, Lint en .NET
- Suelen ofrecer su propio lenguaje de consultas y (habitualmente) hacen uso por defecto de consultas parametrizadas
- Lenguaje de consultas puede seguir siendo vulnerable ante consultas dinámicas construidas ”a mano”
- Posibilidad de lanzar consultas directas sobre SQL ($\Rightarrow$ sigue siendo posible inyección ”clásica”)

Prevención Inyección SQL (II)

(4) ”Escapado” de los caracteres problemáticos en SQL

- Evitar enviar la entrada de usuario tal cual ”escapando” los caracteres especiales problemáticos (`, . " ' ( ) | + #`)
- La consulta resultante contendrá las cadenas `\' \# \" \;` en lugar de las incluidas por usuario
- Dependiente de la implementación de cada gestor BD de su dialecto SQL
- Ejemplo: funciones `mysqli_real_escape_string` o `addslashes` en PHP

(5) Filtrado de cadenas peligrosas *(lista negra)*

- Comprobar la presencia de cadenas o subcadenas peligrosas en la entrada de usuario, descartándola si estás presentes
- Omitir palabras clave SQL, caracteres no previstos en el tipo de dato previsto para cada entrada, etc
- No excesivamente robusto, posibilidad de evasión/ofuscación

XSS: Cross Site Scripting (I)

Ejecución de código script arbitrario ”incrustado” por un tercero en campos de datos no validados que son enviados tal cual desde el servidor WEB al navegador de la víctima

- Se incluirán datos que serán interpretados como código legítimo por el navegador de la víctima y que modificarán el comportamiento o aspecto de la página

no validación datos recibidos (con HTML/Javascript ”no previsto”)	+	no validación HTML generado y remitido a víctima	→	navegador víctima interpreta ese HTML/Javascript

Peligro: cuando se ejecuta un script se tiene acceso a la página actual y a toda la información contenida en la misma

- obtener cookie del usuario (robando su id de sesión)
- redirigir a una página alternativa
- presentar contenido falso (*phishing*) para que introduzca info. privada

Tipos
- XSS reflejado
- XSS almacenado
- XSS basado en DOM

XSS: Cross Site Scripting (II)

Etiquetas ”problemáticas”: `<script> ... </script>`

- Introducidas en algún campo de formulario, en parámetros de la URL o en un parámetro de una petición HTTP:
  `<script> alert("Esto es vulnerable a XSS") </script>`

También posibilidad de inyección de scritps JS en otras etiquetas:

```
<img src="javascript:alert('Vulnerable a XSS')"/>
```

```
<img src='#' onerror=alert('Vulnerable a XSS')/>
```

La mayor parte de etiquetas que admitan atributos STYLE, SRC, HREF, TYPE son susceptibles de ser usadas

`<html>`, `<body>`, `<embed>`, `<frame>`, `<applet>`, `<iframe>`, `<meta>`, `<object>`, `<style>`

Cross Site Scripting (III)

Protecciones: filtrar los datos que el usuario introduce y (especialmente) los que el servidor sirve

- ”Escapar” caracteres problemáticos (`<`, `>`, `'`, `"`, `&`) conviertiéndolos en entidades HTML
- Descartar cadenas con etiquetas sospechosas (lista negra)
- Especial atención en ”zonas peligrosas” (etiquetas susceptibles de desencadenar ejecución código Javascript)
- Algunos frameworks Web (p.e. JSF *Java Server Faces*) incluyen soporte para evitar XSS proporcionando mecanismos para ”emitir” código HTML *sanitizado* a partir de datos de la aplicación.

Nota: También existen filtros con protección contra XSS en los navegadores web

`https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet`

Limitaciones:

- Atacante puede ofuscar/codificar cadenas introducidas para que pasen desapercibidas
- No siempre fácil detectar scripts introducidos dentro de etiquetas HTML

  Ver: `https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet`