# Auditorías

Manuel García Cervigón

# Indice

- **Introducción**
- **Vulnerabilidades comunes**
- **Information Gathering**
- **Escaneo de red y enumeración**
- **Análisis**
- **Explotación**

# Introducción

- Que es una Auditoría?
- La **auditoría** consiste en la revisión sistemática de una actividad o de una situación para evaluar el cumplimiento de las reglas o criterios objetivos a que aquellas deben someterse, según la definición dada por la Real Academia de la Lengua Española.

- En nuestro caso consiste en **revisar** la seguridad y la integridad de aplicaciones o servidores y **evaluar** las posibles vulnerabilidades encontradas.

# Introducción

- Cuándo se debe hacer una auditoría?

- Cuando un servidor, ofrece servicios web y al mismo tiempo contiene o tiene conexión a datos confidenciales de cualquier nivel.

- Se debe tener el **consentimiento** del/la Organización/ Webmaster del servidor a auditar.

*Atención a los servicios en nube o hosting por los posibles efectos colaterales.*

# Introducción

- Como se empieza una auditoría?
- Se debe pactar con el administrador:
  - La localización del servidor o servidores.
  - Las partes a auditar.
  - Si la auditoría se realizará con o sin credenciales. (Caja negra, Caja Blanca), *interna / externa*
  - Si está permitida la intrusión.

# Introducción

- Resultados de las auditorías:
  - El auditor debe hacer una evaluación de la seguridad y un resumen de las vulnerabilidades encontradas.
  - **Atención**: El auditor **no soluciona** vulnerabilidades, pero si que recomienda/marca directrices para resolver el problema. *Separación de funciones como garantía*
  - Finalmente se entrega al administrador un documento con el resumen del trabajo hecho sobre la aplicación.
  - Todos los datos extraídos son confidenciales y deben ser destruidos o guardados para posteriores auditorías por el auditor.
  - *Acuerdo de confidencialidad!*

# Vulnerabilidades comunes

- XSS
- Inyecciones SQL
- Buffer Overflow

# Vulnerabilidades comunes
# XSS

- Un Cross-Site Scripting ocurre por una falta de validación y tratamiento de los datos de entrada a la aplicación web; lo que es aprovechado por los atacantes para realizar inyecciones de código script (Vbscript, Javascript,…)

- *Tipos de Cross–Site Scripting*

- **Persistente**: Los datos sin filtrar son almacenados en una base de datos

- **Reflejado**: Los datos sin filtrar son reflejados desde una entrada en la aplicación web (campo de formulario, campo hidden,…)

# Vulnerabilidades comunes
# XSS

- *Ejemplo 1*
- Consideremos una página web que tenga un mensaje de bienvenida “bienvenido “%NombreUsuario%”

# Vulnerabilidades comunes
# XSS

- Podemos sospechar que cada posible entrada de datos puede resultar en un ataque XSS. Para analizarlo, el auditor jugara con la variable de usuario e intentara provocar la vulnerabilidad. Si probamos a introducir:
- *http://ejemplo.com/index.php?user=<script>alert(123)</script>*

# Vulnerabilidades comunes
# XSS

- *Ataques que se pueden llevar a cabo:*
- Secuestrar el navegador de otro usuario.
- Capturar información sensible que vean los usuarios de la aplicación.
- Defacement “aparente” de la aplicación.
- Envío dirigido de exploits basados en navegador.

# Vulnerabilidades comunes
# XSS

- ¿Defacement?

# Vulnerabilidades comunes
# Inyección SQL

Los ataques de inyección SQL son un tipo de ataques de inyección, en los que ordenes SQL son inyectadas en texto para afectar la correcta realización de una consulta SQL predefinida.

El éxito en una inyección SQL puede conllevar:

- Leer datos sensibles de la base de datos
- Modificar los datos (insertar/actualizar/borrar)
- Realizar operaciones de administración sobre la base de datos (como reiniciar el DBMS)
- En algunos casos, ejecutar ordenes en el sistema operativo.

# Vulnerabilidades comunes
# Inyección SQL

*Pruebas a realizar:*

El primer paso en esta prueba es entender cuando nuestra aplicación conecta a un servidor de bases de datos (BBDD) para acceder a algún dato.

- *Formularios de autenticación*: cuando la autenticación es realizada usando un formulario web, es probable que las credenciales de un usuario sean comprobadas contra una BBDD que contenga todos los usuarios y claves.
- *Motores de búsqueda:* la cadena enviada por el usuario puede ser usada en una consulta SQL que recupere todos los registros relevantes de la BBDD.

# Vulnerabilidades comunes
# Inyección SQL

- *Pruebas a realizar:*

  - *Webs de E-Commerce*: los productos y sus características (precio, descripción, disponibilidad,…) son siempre almacenados en una BBDD relacional.

- La prueba mas básica consiste en añadir una comilla simple ( ‘) o un punto y coma (;) al campo que se quiere probar. Lo primero es usado como fin de cadena y, si la aplicación no esta filtrada puede llevar a una consulta incorrecta.

# Vulnerabilidades comunes
# Inyección SQL

- *Pruebas a realizar:*

- *Microsoft OLE DB Provider for ODBC Drivers error '80040e14'*
- *[Microsoft][ODBC SQL Server Driver][SQL Server]Unclosed quota/on mark before*
- *the character string ''.*
- */target/target.asp, line 113*

- También los comentarios (- -) y otras palabras reservadas como 'AND' y 'OR' pueden ser usadas para intentar modificar una consulta. Una técnica muy simple pero aún efectiva es insertar una cadena en un campo donde se espera un numero, se puede generar un error como el siguiente:

- *[Microsoft][ODBC SQL Server Driver][SQL Server]Syntax error converting the varchar value 'test' to a column of data type int.*

# Vulnerabilidades comunes
# Inyección SQL

- *Ejemplo*

- *SELECT * FROM Users WHERE Username='$username' AND Password='$password'*

- $username = 1' or '1' = '1
- $password = 1' or '1' = '1

- La consulta sería:

- *SELECT * FROM Users WHERE Username='1' OR '1' = '1' AND Password='1' OR '1' = '1'*

- La consulta devuelve un valor (o conjunto de valores) porque la condición siempre es verdadera (OR 1=1). En esta caso el sistema autentica al usuario sin conocer el Username ni el Password.

# Vulnerabilidades comunes
# Inyección SQL

- *Ejemplo 2*

- Otra prueba es el uso del operador UNION. Este operador es usado en las inyecciones SQL para unir una consulta. El resultado es crear una consulta que unida a la original permita a un auditor obtener valores de los registros de otras tablas.

- *SELECT Name, Phone, Address FROM Users WHERE Id=1 UNION ALL SELECT creditCardNumber, 1,1 FROM CreditCarTable*

- Lo que nos daría como resultado todas las tarjetas de crédito de los usuarios. La palabra reservada ALL es necesaria para obtener todos los resultados.

# Vulnerabilidades comunes
# Inyección SQL. Blind SQL.

- No siempre tenemos la suerte de poder ver el resultado de la consulta de manera directa.

- Sin embargo si sabemos que un campo de texto se utiliza en una consulta podemos probar de obtener información “a ciegas”.

# Vulnerabilidades comunes

## Inyección SQL. Blind SQL.

- La idea básica es:
  1. Hacer una consulta de resultado cierto o falso
  2. Observar si obtenemos un resultado diferente en el caso cierto o en el caso falso. En caso contrario:
     1. Si el resultado es falso, no hacer nada
     2. Si el resultado es cierto, provocar un comportamiento visible (como esperar 10 segundos)
- De esta manera podemos ir obteniendo información sobre la base de datos.

# Ejemplo de aplicación web vulnerable

La aplicación web vulnerable se explica en estos tutoriales:

http://resources.infosecinstitute.com/the-bodgeit-store-part-1/
http://resources.infosecinstitute.com/the-bodgeit-store-part-2/

En este vídeo se "soluciona" usando Snort como IPS:

https://www.youtube.com/watch?v=GLtBiSaqIOk&list=PL42_rrzeJXbtadx6UJEhamCgcZQi_buSi

La descarga desde aquí:

https://github.com/psiinon/bodgeit

Y el análisis se puede hacer con:

https://www.owasp.org/index.php/ZAP

# Otras fuentes de herramientas de análisis de vulnerabilidades

http://www.hackerhighschool.org/lessons.html

https://www.enisa.europa.eu/activities/cert/training/training-resources/resources
https://www.enisa.europa.eu/activities/cert/training/courses
https://www.enisa.europa.eu/activities/cert/training/training-resources/technical-operational

# Vulnerabilidades comunes
## Buffer Overflow

- Ocurre al no llevarse acabo una **comprobación** de una **entrada** la cual supera el tamaño reservado.

- Por ejemplo, si tenemos una entrada "**abcdefghijklmno**" y, mediante por ejemplo **strcpy()**, la asignamos a un **char buff[10]**.

# Vulnerabilidades comunes
## Buffer Overflow

- A través del buffer overflow se puede **escribir** en partes de la **memoria** donde el programador en principio no tenía intención que escribieramos.
- Si conseguimos **sobreescribir** algún punto clave (por ejemplo una **dirección de retorno**) podemos llegar a obtener el **control completo** de la aplicación afectada.

# Vulnerabilidades comunes
## Buffer Overflow

- Suelen encontrarse o bien mediante un **análisis** del código fuente (si es código abierto), o mediante **fuzzing** de las entradas de una aplicación.

# Vulnerabilidades comunes
# Gestión de vulnerabilidades

- Es necesario conocer, almacenar y gestionar las vulnerabilidades que se descubren:
  - Podemos comprobar rápidamente si se conoce alguna vulnerabilidad para alguna plataforma si conocemos su versión.
  - También permite encontrar exploits a partir de las vulnerabilidades

# Vulnerabilidades comunes
# Gestión de vulnerabilidades

- Avisos de vulnerabilidades
  - CVE – *Common Vulnerabilities and Exposures*
    - Identificador estándar de vulnerabilidades
  - CPE – *Common Platform Enumeration*
    - Identificador estándar de plataforma
  - Fuente
    - Quien haya emitido el aviso de vulnerabilidad
  - Descripción
    - Efectos que provoca la vulnerabilidad. *Severidad*
  - Solución

# Vulnerabilidades comunes
# Gestión de vulnerabilidades

- Avisos de vulnerabilidades
  - CVE – CVE-2014-0268
  - CPE
    - cpe:/a:microsot :internet_explorer:11:
    - cpe:/a:microsot :internet_explorer:10
    - cpe:/a:microsot :internet_explorer:9
    - cpe:/a:microsot :internet_explorer:8
  - Descripción
    - Hay una vulnerabilidad de elevación de privilegios en Internet Explorer durante la validación de la instalación de archivos locales y durante la creación segura de claves del Registro.

# Vulnerabilidades comunes
# Gestión de vulnerabilidades

- Fuentes de información
  - NIST – http://web.nvd.nist.gov/view/vuln/search
    - Contiene avisos de vulnerabilidades de todos las plataformas que publiquen CVEs
  - ALTAIR http://escert.upc.edu/servicios#g_vulnerabilidades - Hace la función del NIST pero en castellano
  - Páginas de fabricante
    - También si nos interesa una aplicación en concreto podemos mirar la página del fabricante para investigar las posibles vulnerabilidades descubiertas

# Information gathering

- Es importante a la hora de llevar a cabo una auditoría obtener el máximo de información posible de los servidores o servicios auditados
- No es solo informarnos, sino también buscar “conocimientos indebidos” como pueden ser la estructura interna de la red, archivos que deberían de estar ocultos…
- Es decir, queremos encontrar información que no deberíamos poder encontrar

# Information gathering

- DNS snooping
  - Consiste en enviar peticiones a un servidor DNS para obtener información sobre que páginas tiene en cache y deducir que accesos se suelen producir

# Information gathering

- PTR (Reverse DNS Lookup)
  - Consiste en llevar a cabo peticiones PTR, que a partir de una IP nos dan el nombre de dominio.

# Information gathering

- Dnsenum
  1. Get the host's addresses (A record).
  2. Get the nameservers (threaded).
  3. Get the MX record (threaded).
  4. Perform axfr queries on nameservers and get BIND versions(threaded).
  5. Get extra names and subdomains via google scraping (google query ti "allinurl: -www site:domain").
  6. Brute force subdomains from file, can also perform recursion on subdomain that have NS records (all threaded).
  7. Calculate C class domain network ranges and perform whois queries on them (threaded).
  8. Perform reverse lookups on netranges ( C class or/and whois netranges) (threaded).
  9. Write to domain_ips.txt file ip–blocks.

# Information gathering

- Google hacking
  - Es una técnica que utiliza el buscador de google con parámetros específicos que permiten filtrar por url o ciertas strings que nos pueden permitir descubrir archivos internos del servidor que estén indexados.

# Information gathering

- Deep Web & Pastebin
  - Siempre es buena idea buscar las apariciones de la entidad que estemos auditando en la Deep Web (andar con ojo) y Pastebin, dado que muchos grupos de… “moralidad dudosa” suelen dejar ahí la información que descubren.

# Information gathering

- FOCA
  - Combina y automatiza muchas técnicas
  - Permite extraer información de los metadatos de los ficheros encontrados

# Escaneo de red y enumeración

- Una de las primeras cosas que necesitamos saber es qué servicios están al descubierto, es decir, que son accesibles desde fuera
- Queremos realizar un escaneo para detectar qué IPs y puertos están abiertos y qué servicios hay tras ellos
- Hay diversas herramientas útiles para llevar esto a cabo de manera automática

# Escaneo de red y enumeración

- nmap
  - Herramienta de escaneo de red y puertos muy potente por linea de comandos de linux.
    - nmap -v -A scanme.nmap.org
      - Scan detallado con detección de OS, versiones, scripts y traceroutes
    - nmap –F scanme.nmap.org
      - Scan rápido que detecta sólo los puertos abiertos o filtrados

# Escaneo de red y enumeración

- nikto
  - Herramienta especializada de escaneo de servicios web.
    - nikto scanme.example.org

# Escaneo de red y enumeración

- Wapiti
  - Herramienta de escaneo de aplicaciones web que detecta puntos de entrada y mediante fuzzing (fuerza bruta) comprueba una serie de vulnerabilidades:
    - File disclosure (Local and remote include/require, fopen, readfile...)
    - Database Injection (PHP/JSP/ASP SQL Injections and XPath Injections)
    - XSS (Cross Site Scripting) injection (reflected and permanent)
    - Command Execution detection (eval(), system(), passtru()...)
    - CRLF Injection (HTTP Response Spli~ n g, session fixation...)
    - XXE (Xml eXternal Entity) injection
    - Use of know potentially dangerous files (thanks to the Nikto database)
    - Weak .htaccess configurations that can be bypassed
    - Presence of backup files giving sensitive information (source code disclosure)

# Escaneo de red y enumeración

- OpenVAS
  - Plataforma de pentesting que detecta una gran cantidad de vulnerabilidades basada en detección de versiones de los servicios detectados tras puertos abiertos
  - NO lleva a cabo pruebas intrusivas por defecto
  - Utiliza nmap y nikto internamente entre otras herramientas

# Escaneo de red y enumeración

- OSSIM –Open Source Security Information Management
  - Es una plataforma enorme que contiene, entre otras cosas: Snort, openvas, Arpwatch, Nagios…

# Análisis
# Fuzzing

- Fuzzing es una técnica de detección de vulnerabilidades que consiste en introducir valores aleatorios, no válidos o con contenido “peligroso” con la intención de provocar un fallo o comportamiento indebido en un programa o aplicación.

# Análisis
# Fuzzing

- Se utiliza sobretodo para encontrar nuevas vulnerabilidades desconocidas o errores de programación/configuración.

# Análisis
# Fuzzing. Herramientas.

- Wfuzz
  - Conocida herramienta que permite atacar servicios mediante fuzzing y es altamente configurable
  - Wfuzz.py –c –z file,commons.txt –hc 404 –o html http://www.site.com/FUZZ 2> res.html

# Análisis
## Fuerza bruta

- Similar al fuzzing, pero se aplica con la intención de encontrar una entrada o combinación de entradas que provoque un resultado deseado (por ejemplo usuario y contraseña para accede a algún lugar).

# Análisis
## Fuerza bruta

- Se suelen utlizar listados de combinaciones o entradas habituales.
- Se pueden utilizar diccionarios.
- En última instancia, se puede intentar recorrer todas las posibles entradas (esto suele tardar mucho tiempo).

# Análisis
# Análisis Manual

- No podemos encontrar todas las vulnerabilidades existentes mediante análisis automáticos ya que:
  - Generan falsos positivos
  - No encuentran todas las vulnerabilidades posibles
  - A veces no distinguen bien cosas que a un experto le resultan “obvias”

# Análisis
# Análisis Manual

- Suele requerir conocimientos específicos de lo que quieres analizar
- La intuición y la creatividad del auditor puede jugar un papel importante
- La única manera de aprender es **haciéndolo**

# Análisis
# Análisis Manual. OWASP.

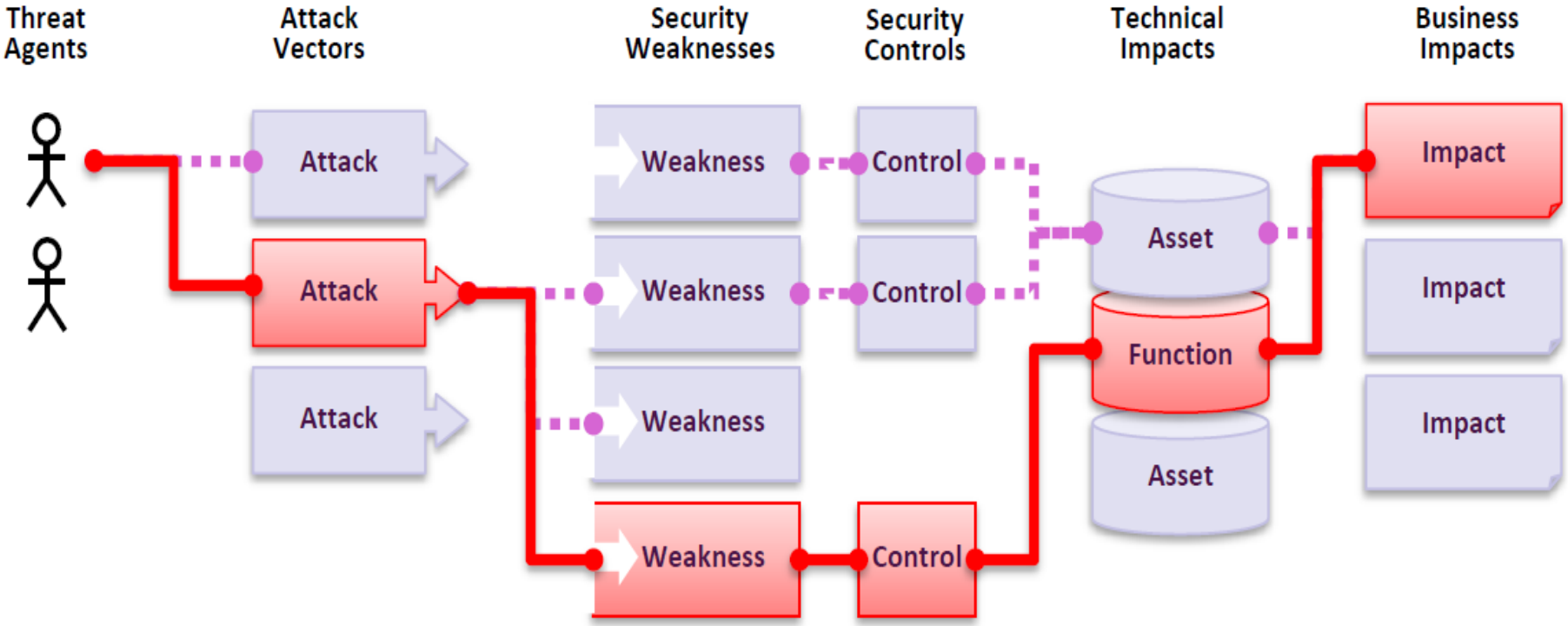
- OWASP es una comunidad **abierta** dedicada a habilitar a las organizaciones para **desarrollar**, **comprar** y **mantener** aplicaciones **confiables**.
- Todas las herramientas, documentos, foros y capítulos de OWASP son **gratuitos** y **abiertos** a cualquiera interesado en mejorar la **seguridad** de aplicaciones.
- https://www.owasp.org/index.php/Top_10_2013-Top_10

Open Web Application Security Project

| OWASP Top 10 – 2013 (Previous) | OWASP Top 10 – 2017 (New) |
| --- | --- |
| A1 – Injection | A1 – Injection |
| A2 – Broken Authentication and Session Management | A2 – Broken Authentication and Session Management |
| A3 – Cross-Site Scripting (XSS) | A3 – Cross-Site Scripting (XSS) |
| A4 – Insecure Direct Object References - Merged with A7 | A4 – Broken Access Control (Original category in 2003/2004) |
| A5 – Security Misconfiguration | A5 – Security Misconfiguration |
| A6 – Sensitive Data Exposure | A6 – Sensitive Data Exposure |
| A7 – Missing Function Level Access Control - Merged with A4 | A7 – Insufficient Attack Protection (NEW) |
| A8 – Cross-Site Request Forgery (CSRF) | A8 – Cross-Site Request Forgery (CSRF) |
| A9 – Using Components with Known Vulnerabilities | A9 – Using Components with Known Vulnerabilities |
| A10 – Unvalidated Redirects and Forwards - Dropped | A10 – Underprotected APIs (NEW) |

# What Are Application Security Risks?

Attackers can potentially use many different paths through your application to do harm to your business or organization. Each these paths represents a risk that may, or may not, be serious enough to warrant attention.




| Threat Agents | Attack Vectors | Weakness Prevalence | Weakness Detectability | Technical Impacts | Business Impacts |
|---|---|---|---|---|---|
| App Specific | Easy | Widespread | Easy | Severe | App / Business Specific |
| | Average | Common | Average | Moderate | |
| | Difficult | Uncommon | Difficult | Minor | |

# OWASP TOP10 vulnerabilities 2017

| | |
|---|---|
| **A1 – Injection** | Injection flaws, such as SQL, OS, XXE, and LDAP injection occur when untrusted data is sent to an interpreter as part of a command or query. The attacker's hostile data can trick the interpreter into executing unintended commands or accessing data without proper authorization. |
| **A2 – Broken Authentication and Session Management** | Application functions related to authentication and session management are often implemented incorrectly, allowing attackers to compromise passwords, keys, or session tokens, or to exploit other implementation flaws to assume other users' identities (temporarily or permanently). |
| **A3 – Cross-Site Scripting (XSS)** | XSS flaws occur whenever an application includes untrusted data in a new web page without proper validation or escaping, or updates an existing web page with user supplied data using a browser API that can create JavaScript. XSS allows attackers to execute scripts in the victim's browser which can hijack user sessions, deface web sites, or redirect the user to malicious sites. |
| **A4 – Broken Access Control** | Restrictions on what authenticated users are allowed to do are not properly enforced. Attackers can exploit these flaws to access unauthorized functionality and/or data, such as access other users' accounts, view sensitive files, modify other users' data, change access rights, etc. |
| **A5 – Security Misconfiguration** | Good security requires having a secure configuration defined and deployed for the application, frameworks, application server, web server, database server, platform, etc. Secure settings should be defined, implemented, and maintained, as defaults are often insecure. Additionally, software should be kept up to date. |

# OWASP TOP10 vulnerabilities 2017

| | |
|---|---|
| **A6 – Sensitive Data Exposure** | Many web applications and APIs do not properly protect sensitive data, such as financial, healthcare, and PII. Attackers may steal or modify such weakly protected data to conduct credit card fraud, identity theft, or other crimes. Sensitive data deserves extra protection such as encryption at rest or in transit, as well as special precautions when exchanged with the browser. |
| **A7 – Insufficient Attack Protection** | The majority of applications and APIs lack the basic ability to detect, prevent, and respond to both manual and automated attacks. Attack protection goes far beyond basic input validation and involves automatically detecting, logging, responding, and even blocking exploit attempts. Application owners also need to be able to deploy patches quickly to protect against attacks. |
| **A8 – Cross-Site Request Forgery (CSRF)** | A CSRF attack forces a logged-on victim's browser to send a forged HTTP request, including the victim's session cookie and any other automatically included authentication information, to a vulnerable web application. Such an attack allows the attacker to force a victim's browser to generate requests the vulnerable application thinks are legitimate requests from the victim. |
| **A9 – Using Components with Known Vulnerabilities** | Components, such as libraries, frameworks, and other software modules, run with the same privileges as the application. If a vulnerable component is exploited, such an attack can facilitate serious data loss or server takeover. Applications and APIs using components with known vulnerabilities may undermine application defenses and enable various attacks and impacts. |
| **A10 – Underprotected APIs** | Modern applications often involve rich client applications and APIs, such as JavaScript in the browser and mobile apps, that connect to an API of some kind (SOAP/XML, REST/JSON, RPC, GWT, etc.). These APIs are often unprotected and contain numerous vulnerabilities. |

# OWASP TOP10 vulnerabilities dropped from 2013

| | |
|---|---|
| **A4-Insecure Direct Object References** | A direct object reference occurs when a developer exposes a reference to an internal implementation object, such as a file, directory, or database key. Without an access control check or other protection, attackers can manipulate these references to access unauthorized data. |
| **A7-Missing Function Level Access Control** | Most web applications verify function level access rights before making that functionality visible in the UI. However, applications need to perform the same access control checks on the server when each function is accessed. If requests are not verified, attackers will be able to forge requests in order to access functionality without proper authorization. |
| **A10-Unvalidated Redirects and Forwards** | Web applications frequently redirect and forward users to other pages and websites, and use untrusted data to determine the destination pages. Without proper validation, attackers can redirect victims to phishing or malware sites, or use forwards to access unauthorized pages. |

# OWASP Top Ten Proactive Controls 2016

is a list of security techniques that should be included in every software development project.

1. Verify for Security Early and Often
2. Parameterize Queries
3. Encode Data
4. Validate All Inputs
5. Implement Identity and Authentication Controls
6. Implement Appropriate Access Controls
7. Protect Data
8. Implement Logging and Intrusion Detection
9. Leverage Security Frameworks and Libraries
10. Error and Exception Handling

# Análisis
## Análisis Manual. WebGoat.

- WebGoat es una plataforma desarrollada por OWASP que permite practicar el análisis manual sobre aplicaciones web mediante **retos** o **ejercicios**.

- Para ponerla en marcha se necesitan dos procesos: el WebGoat (dentro de tomcat) y el WebScarab.

# Explotación de vulnerabilidades

- Una vez hemos encontrado las (posibles) vulnerabilidades, el paso final de las auditorias es explotarlas.

- Hay que tener en cuenta que según lo acordado con el cliente habrán vulnerabilidades que no podremos comprobar explotándolas.

# Explotación de vulnerabilidades

- Cuando identificamos una vulnerabilidad, una de las primeras cosas que necesitamos buscar es si existe algún *exploit* conocido

- Muchas veces son vulnerabilidades relativamente sencillas de comprobar (un fichero que debería estar oculto es accesible, un campo de formulario vulnerable a inyecciones SQL…)

# Explotación de vulnerabilidades

- A veces necesitamos algún tipo de *exploit* más complejo, así que debemos primero buscar si ya existe.

- Lo más típico es mirar en *exploit-db* o directamente en *metasploit*.

# Explotación de vulnerabilidades

- Muchas veces encontramos exploits para justo la vulnerabilidad que queremos explotar.
- Otras veces necesitamos modificar un exploit existente.
- En última instancia, puede ser que necesitemos crear nuestro propio exploit.

# Exploits simples

- Algunas vulnerabilidades son extremadamente simples de explotar.

- Por ejemplo, si lanzamos un nmap a la Metasploitable vemos, entre otras cosas, los siguientes puertos abiertos:
  - 512/tcp  open  exec
  - 513/tcp  open  login
  - 514/tcp  open  shell

# Exploits simples

- Estos puertos se conocen como los servicios ‘r’ y no parecen estar bien configurados ya que los estamos viendo externamente.

- Para explotar este error, necesitamos tener el cliente “rsh-client” instalado.
  - sudo apt-get install rsh -client

# Exploits simples

- Una vez tenemos la herramienta necesaria, ejecutamos:
  - rlogin –l root 192.168.15.130

- Descubrimos que… sí, efectivamente tenemos acceso root

# Uso de exploits avanzados

- Evidentemente, no siempre es tan sencillo explotar una vulnerabilidad, muchas veces require de algún mecanismo bastante más complicado.

- Otro puerto que nos puede interesar, para ejecutar código C de forma distribuida, es:
  - 3632/tcp  open  distccd

# Uso de exploits avanzados

- Hay algunos servicios que, si bien no son maliciosos, se comportan casi como puertas traseras por propia naturaleza.

- La Metasploitable utiliza distccd, un programa para escalar trabajos de compliación que sean muy grandes a traves de muchos sistemas… pero puede ser explotado para ejecutar el commando que queramos.

# Uso de exploits avanzados

- Para explotar esta vulnerabilidad, utilizaremos un módulo de Metasploit:
  - msfconsole
  - msf> use exploit/unix/misc/distcc_exec
  - msf exploit(distcc_exec) > set RHOST 192.168.15.130
  - msf exploit(distcc_exec) > exploit

# Uso de exploits avanzados

- Si nos fijamos en la ejecución del exploit, veremos que se envía un comando (echo uk3UdiwLUq0LX3Bi;) y que la respuesta del servidor es, efectivamente, “uk3UdiwLUq0LX3Bi”.

- Metasploit envía un comando “echo” para evitar ser excesivamente intrusivo, pero este comando podría ser cualquier cosa…

# Modificando un exploit

- ¿Y si queremos que en lugar de hacer un simple “echo” haga algo más… interesante?

- Podemos ir a:
  - /usr/share/metasploit-framework/modules/exploits/unix/misc
- y editar el fichero:
  - leafpad distcc_exec.rb &

# Modificando un exploit

- Vamos a la sección def_exploit, donde se encuentra el código del exploit

- Aquí podemos modificar libremente el código, pero necesitaremos reiniciar la msfconsole para que se aplique

# Modificando un exploit

- Si por ejemplo cambiamos el código del echo, veremos los cambios que hemos ejecutado

# Creando un exploit

- A veces necesitamos crear nosotros mismos el *exploit* para poder llevarlo a cabo

- Vamos a tratar el ejemplo de un buffer overflow en un servidor, y como podemos llegar a obtener acceso telnet al mismo a través de él

# Creando un exploit

- Si observamos el código, hay un fragmento que tiene casi todo lo que podríamos desear para poder llevar a cabo un exploit:

```
void pr( char *str)
{
  char buf[500]="";
  strcpy(buf,str);
}
```

# Creando un exploit

- Por una parte la instrucción *char buf[500]* reserva 500 bytes de memoria.
- Por otra parte *strcpy(buf,str)* copia datos a este buffer ***sin comprobación alguna***. Esto es un *buffer overflow* asegurado.
- Finalmente, hay seguido un retorno de función, lo cual quiere decir que de sobrescribir la dirección de retorno, inmediatamente saltaremos a ella.

# Creando un exploit

- Primero comprobaremos que, efectivamente, el programa termina en error si le introducimos más datos en la entrada de los que caben en el búfer.

- Y si miramos el estado de los registros… hemos sobrescrito el EIP

# Creando un exploit

- El EIP es el registro que apunta a la siguiente instrucción que se ha de ejecutar, si lo podemos modificar, puede decir que podemos controlar el flujo del programa, y por tanto podemos ejecutar el código que queramos.

# Creando un exploit

- Si investigamos un poco podemos pensar que lo lógico sería que al escribir más de 500 bytes, el 501, por estructura de la pila en la función correspondiente, modificaría el EIP.

- Pero esto no es así, necesitamos insertar 504 bytes en lugar de 500, ¿porqué?

# Creando un exploit

- El búfer en *char buf[500]* se ha creado con tipo de array de chars.
- Esto quiere decir que se colocan 500 bytes de información y luego un byte adicional para marcar el final.
- Pero los tamaños reservados (por temas de funcionamiento del procesador) necesitan ser múltiples de 4, por lo tanto se acaban reservando 504 bytes.

# Creando un exploit

- Ahora que sabemos como colocar el valor que queramos en el EIP, necesitamos poner un valor útil.
- Lo más típico es introducir algo similar a *jmp esp*.
- Esto es porque el ESP es una referencia fija a partir de la cual podemos decidir donde introducir nuestro código malicioso.

# Creando un exploit

- Sin embargo, la instrucción *jmp esp* varía según el procesador y sistema operativo… así que hace falta ejecutar la herramienta findjmp.exe en una máquina de características similares a la que queremos atacar (o la misma máquina)

- Así obtenemos el valor que nos interesa introducir en el EIP.

# Creando un exploit

- Ahora podemos controlar a qué instrucción saltamos, así que solo falta introducir nuestro código malicioso.

- Por suerte, metasploit nos ayuda:
  - msfconsole
  - use payload/Windows/Shell_bind_tcp
  - generate –e x86/alpha_upper –o LPORT=5555,EXITFUNC=seh –t pl

# Creando un exploit

- Introducimos el código generado en nuestro programa y… ¡Voilà!
- Hemos creado ya nuestro exploit, pero… ¿Y si vamos un paso más?
- Convirtámoslo en un módulo de Metasploit.

# Creando un exploit

- Los módulos de Metasploit están escritos en Ruby y siguen una estructura:
  1. Incluir módulos necesarios
  2. Definir la clase de exploit
  3. Incluir módulos específicos del exploit
  4. Establecer los parámetros que utiliza el exploit
  5. Llevar a cabo el exploit

# Creando un exploit

- Siguiendo la guía o mirando otros exploits hechos, podemos observar cómo es el resultado de hacer un módulo de Metasploit.

- Finalmente, podemos probar ejecutar el exploit a través de Metasploit mediante msfconsole!