Corso di Tecnologie e Linguaggi per il Web
A.A. 2020-2021

XSS-Security

Alessandro Ravizzotti
922845

Indice

Introduzione

Personalmente sono molto interessato all'ambito della cyber security e ho più volte provato a svolgere attività di bug bounty tramite il sito web Bug Crowd.
I Cross-Site-Scripting sono delle vulnerabilità che permettono a persone malintenzionate di prendere dati dai dispositivi degli utenti che visitano una pagina infetta.
Questo progetto nasce dal voler automatizzare e rendere più veloci gli attacchi XSS al fine di verificare la presenza di queste vulnerabilità nelle pagine web e aiutare gli sviluppatori a rendere più sicuri i propri siti tramite dei semplici controlli.

Breve analisi dei requisiti

Destinatari

I destinatari di questa app si dividono in due categorie. Esperti nel settore della sicurezza o sviluppatori di siti web (indispensabile è la conoscenza del linguaggio javascript) e coloro che non conoscono le basi della programmazione per il web ma possiedono un sito e vogliono testarne la sicurezza riguardante possibili vulnerabilità per attacchi XSS.
L'interfaccia pulita e intuitiva permette a chiunque di comprendere i risultati dei test. Inoltre, le specifiche presenti nei report consentono agli esperti, tramite l'aiuto della INFO page, di riuscire a risolvere o minimizzare le vulnerabilità.
Adattandosi automaticamente alle dimensioni del device utilizzato, permette una fruizione ottimale da tutti i dispositivi. È inoltre presente la dark mode che viene attivata automaticamente in base alle impostazioni del device.
La quantità di banda richiesta dalla web app è stata minimizzata per quanto possibile al fine di rendere la navigazione più veloce e piacevole. Essa può dipende dal numero di <input> presenti nella pagina testata.
Gli utilizzatori dell'applicazione saranno principalmente guidati da una motivazione di tipo informativo, tramite ricerca diretta e attiva per tale servizio.

Modello di valore

Il valore dell'applicazione è dato principalmente dalla scansione di una o più pagine web e dagli attacchi eseguiti in modo automatico su tutti gli <input> presenti nei siti. Questo permette di rilevare possibili vulnerabilità in modo facile e veloce, sia per un utente esperto che per un proprietario di un sito web.
Questo progetto genera un valore personale riguardo la ricerca nell'ambito della cyber security e bug bounty mentre non presenta alcun valore di tipo economico in quanto non è monetizzato.

Il progetto è e sarà sempre open source, per permettere ad altri programmatori di contribuire nello sviluppo dell'app.

Flusso dei dati

L'utente fa una richiesta alla web app. Questa, tramite la libreria puppeteer, cerca nel sito target la presenza di <form>, <input> e possibili elementi che potrebbero compromettere il risultato dei report (esempio: google recaptcha). In fine il sito target viene attaccato dalla web app (tramite degli script e puppeteer) e viene registrato il suo comportamento dal quale vengono generati i dati dei report che verranno poi inviati all'utente.
Al fine di garantire il più alto livello di privacy possibile, questi dati possono essere archiviati temporaneamente come file .html e .pdf (per un tempo inferiore ai 30 minuti) solamente se l'utente richiede il download dei report; altrimenti nessun dato viene conservato.
La web app raccoglie unicamente le informazioni necessari per generare un report. Questi dati non necessitano di ulteriori trasformazioni e vengono inviati all'utente come file .json.

Aspetti tecnologici

Server-side

Il server, basato su Node.js, utilizza come web framework Express il quale fornisce tutte le funzionalità necessarie per offrire il contenuto all'utente, tramite pagina web.
Moduli utilizzati:

- Express - Web Framework
- Puppeteer - Fornisce API per il controllo di Chromium

Ricevuto l'url del sito da testare, il server crea un oggetto con una serie di attributi i quali verranno modificati a seguito dell'andamento dei test effettuati tramite puppeteer. In fine il server risponde con un oggetto JSON, il quale servirà al client per la creazione dei report.
Il server, al fine di poter generare i pdf dei report richiesti dall'utente, presenta una dipendenza dal programma "wkhtmltopdf", il quale prende in input un file html creato dal server e restituisce un file PDF. Entrambi i file generati verranno cancellati automaticamente dopo lo scadere di un timer.

Client-side

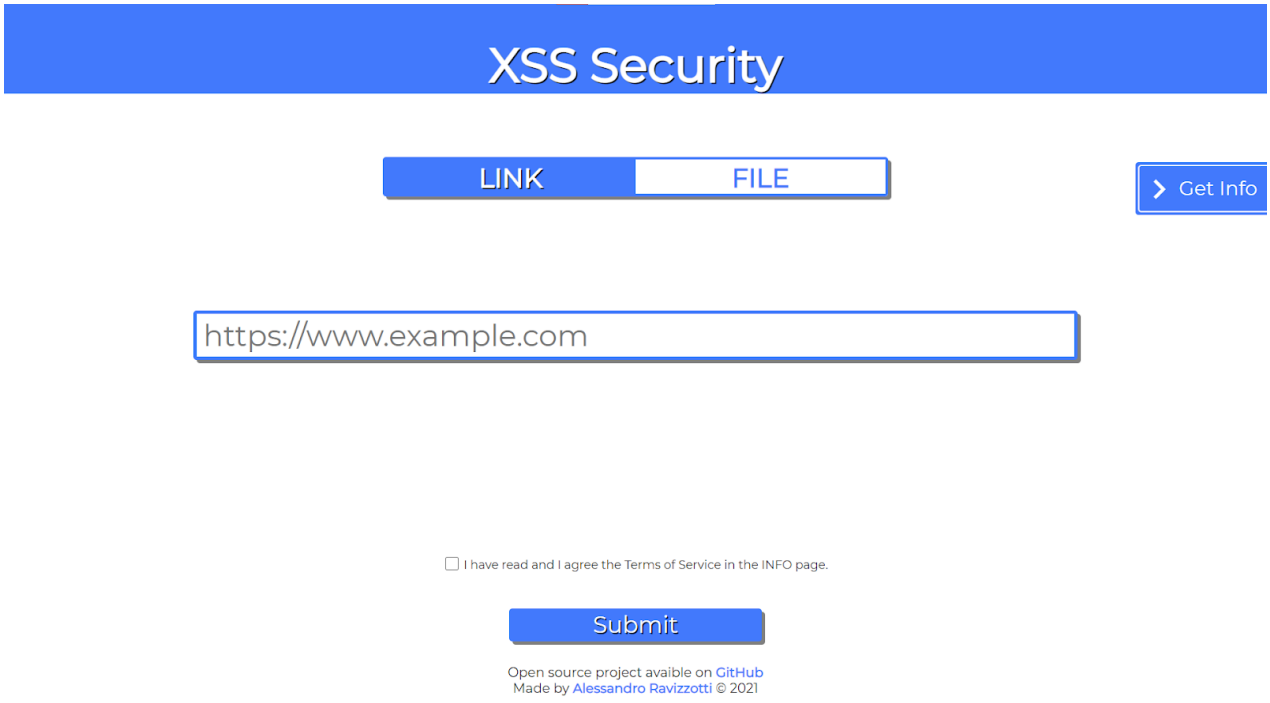
Le pagine web sono conformi allo standard HTML5 e il loro stile è composto da dei file .css da me creati.
Nella pagina principale è presente un form nel quale possono essere inseriti in mutua esclusione due input differenti. In essa sono presenti i cookie per impostare come metodo di input l'ultimo utilizzato e per mostrare o nascondere il collegamento alla pagina delle informazioni.
Con il primo tipo di input si decide di inserire un url, mentre con il secondo si vuole inoltrare un file di tipo .txt, nel quale saranno presenti una serie di url separati da "spazio" e/o "a capo". Inoltre è presente una checkbox, senza la quale non è possibile procedere con i test, che ha il compito di verificare che l'utente abbia letto e accettato i termini di servizio presenti nella pagina delle informazioni.
Una volta che i dati vengono inviati al server tramite POST la pagina principale viene modificata tramite javascript per poter mostrare i risultati dei test i quali verranno caricati in modo asincrono.
Un'altra pagina è quella delle informazioni, totalmente statica, nella quale sono presenti sia i termini di servizio che metodi per riuscire a minimizzare la vulnerabilità dei XSS.

Commenti

La separazione tra la computazione del server e quella del client è stata in parte obbligata a causa della tipologia dell'attacco XSS, la quale non può essere svolta dal client per il "Cross-Origin Resource Sharing". Il CORS non permette l'accesso ai codici interpretati e sorgenti di un sito web se quest'ultimo non è stato precedentemente configurato per tale condivisione di dati.
L'iniziale scelta di eseguire i test tramite il modulo di Node.js "Axios" fu scartata, perché tramite quest'ultimo, si riusciva ad ottenere unicamente il codice sorgente della pagina target e non il codice interpretato, il quale risulta essere molto più importante e utile al fine di svolgere l'attacco. Si è deciso quindi di utilizzare il modulo "Puppeteer" il quale risolve questa problematica.
Un ulteriore decisione che è stata fatta per rendere l'applicazione più veloce per l'utente è quella della creazione dei pdf in locale sul server, tramite l'utilizzo del programma "wkhtmltopdf".

Interfacce

Pagina principale



La schermata principale presenta molte parti interattive e può avere due aspetti differenti in base alla modalità di input selezionata.

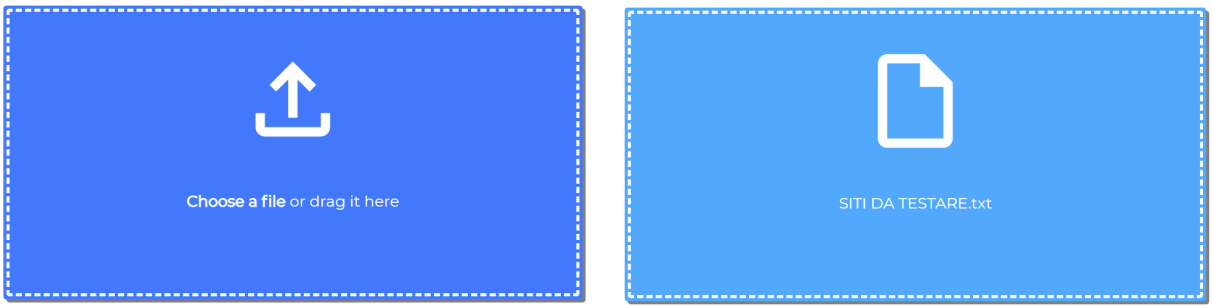
Bottoni di selezione della modalità di input

Due bottoni che servono per cambiare la modalità di input selezionata. Il tasto di colore blu indica l'attuale modalità. Variandola si invertiranno i colori dei due blocchi e si cambia modalità di input, nascondendo la forma attuale e mostrando l'altra.

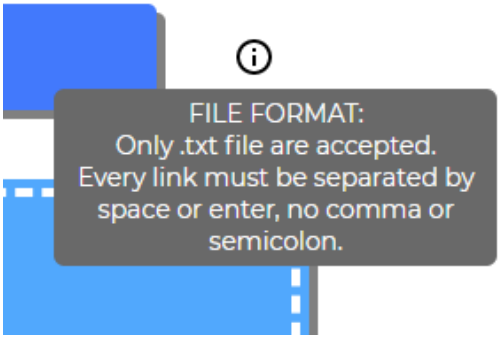
Modalità di input: url

Input testo di tipo url.

Modalità di input: file



Al centro della pagina è presente un box di colore azzurro. Cliccando su di esso si può inoltrare un file formato .txt. È possibile inviare un documento tramite sistema "drag and drop". Quando il file sarà sopra il riquadro, quest'ultimo cambierà di colore.



Inoltre selezionando questa modalità di input verrà visualizzato, di fianco ai bottoni di selezione della modalità di input, una piccola immagine. Spostando il puntatore sopra di essa si visualizzeranno le informaizoni riguardanti il contenuto del documento da inoltrare.

Checkbox

Campo da compilare obbligatoriamente per poter procedere con i test, con il quale l'utente afferma di aver preso visione e accettato i termini di servizio presenti nella pagina delle informazioni.
Se si provano ad eseguire i controlli senza aver prima spuntato la checkbox, si riceverà un messaggio di errore rappresentato dal cambiamento di colore, da bianco a rosso, della casella.

Tasto Submit

Premendo questo pulsante viene letto il file o la casella di input. Si inseriscono gli url all'interno di un array e tutti gli elementi all'interno del main vengono eliminati.
A questo punto vengono caricati gli elementi della pagina report e per ogni stringa presente nell'array, se è effettivamente un url, si effettua una XMLHttpRequest al server (inviando

l'url), altrimenti viene automaticamente generato un blocco di report per gli input che non corrispondono al formato degli url.

Get Info box

Cliccando sul testo di questo box si naviga verso la pagina delle informazioni. Premendo sulla freccia si può minimizzare o ingrandire il box.

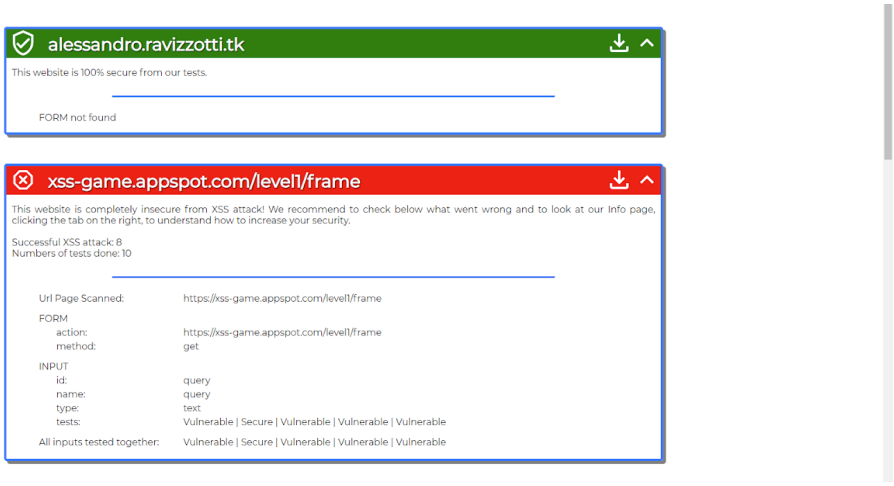
Footer

Sono presenti due link, di colore azzurro, che reindirizzano l'utente al repository di GitHub oppure alla mia pagina personale.

Pagina report

Dopo aver premuto il tasto submit nella pagina principale, la schermata diventerà come quella dell'immagine qui sopra. In seguito alla ricezione dei dati dal server, riferiti ad un singolo sito web, verrà creato il blocco del report. Un volta che tutti i dati sono arrivati il cerchio di caricamento scomparirà.

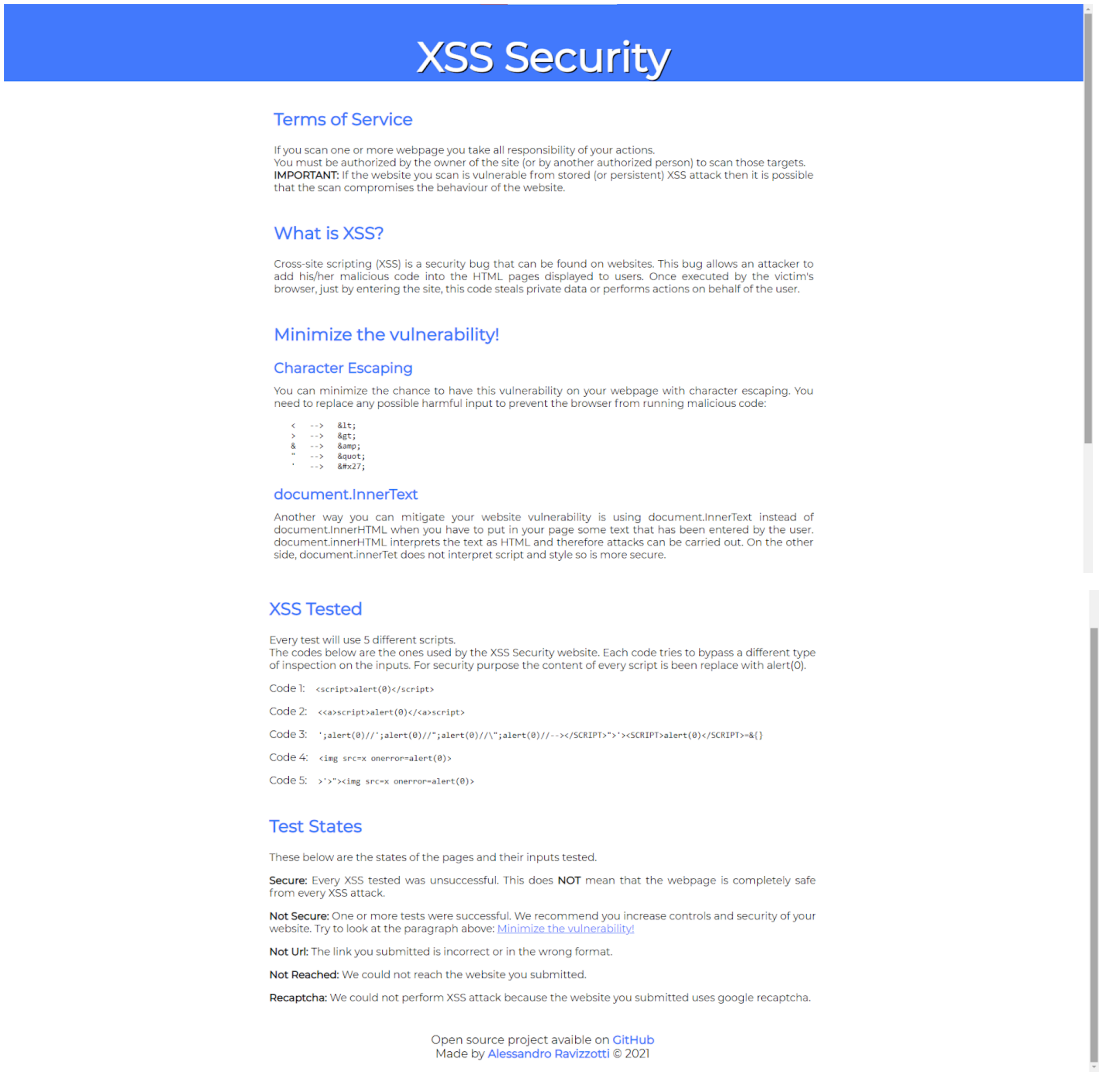
Ogni blocco presenta un titolo e una descrizione sull'andamento dei test. I blocchi possono avere 3 colori differenti in base al risultato del report.
Il colore di sfondo del titolo sarà giallo se non è stato possibile raggiungere un sito oppure se la pagina web presentava una sicurezza contro i bot (esempio: google recaptcha).
I blocchi di colore verde e rosso, nella barra del titolo, presentano due bottoni. Il primo che se cliccato invierà al server una richiesta di download del pdf del singolo report. Il secondo ha lo scopo di mostrare o nascondere le specifiche tecniche dei test eseguiti sul singolo report.



Nelle specifiche tecniche sono presenti tutti i dati raccolti (e ovviamente non archiviati) dal server per effettuare i test.
In alto alla pagina sono presenti due bottoni che servono per mostrare o nascondere le specifiche dei test di tutti i report.

In basso alla pagina è presente un bottone che se cliccato invierà al server la richiesta di download del pdf con all'interno tutti i report presenti nella pagina.
Anche in questa schermata è presente, come in quella principale, il "Get Info" box e il footer.

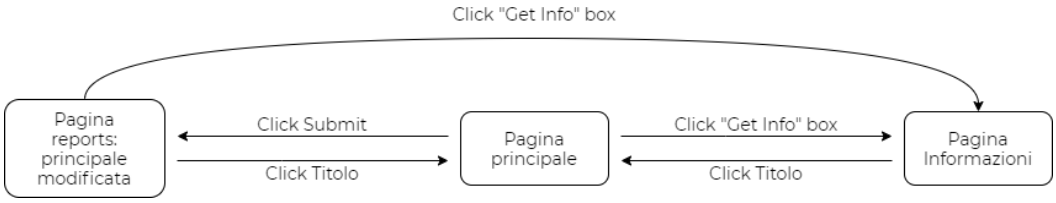
Pagina informazioni e termini di servizio



In questa pagina si possono vedere i termini di servizio i quali devono essere accettati per poter testare un sito e altre informazioni riguardanti XSS, come minimizzare queste vulnerabilità, quali scripts vengono utilizzati per attaccare le pagine target e quali sono i possibili stati dei report.
Cliccando sul titolo "XSS Security" si ritorna alla pagina principale, mentre i due link presenti nel footer (testo di colore azzurro) reindirizzano l'utente al repository di GitHub oppure alla mia pagina personale.

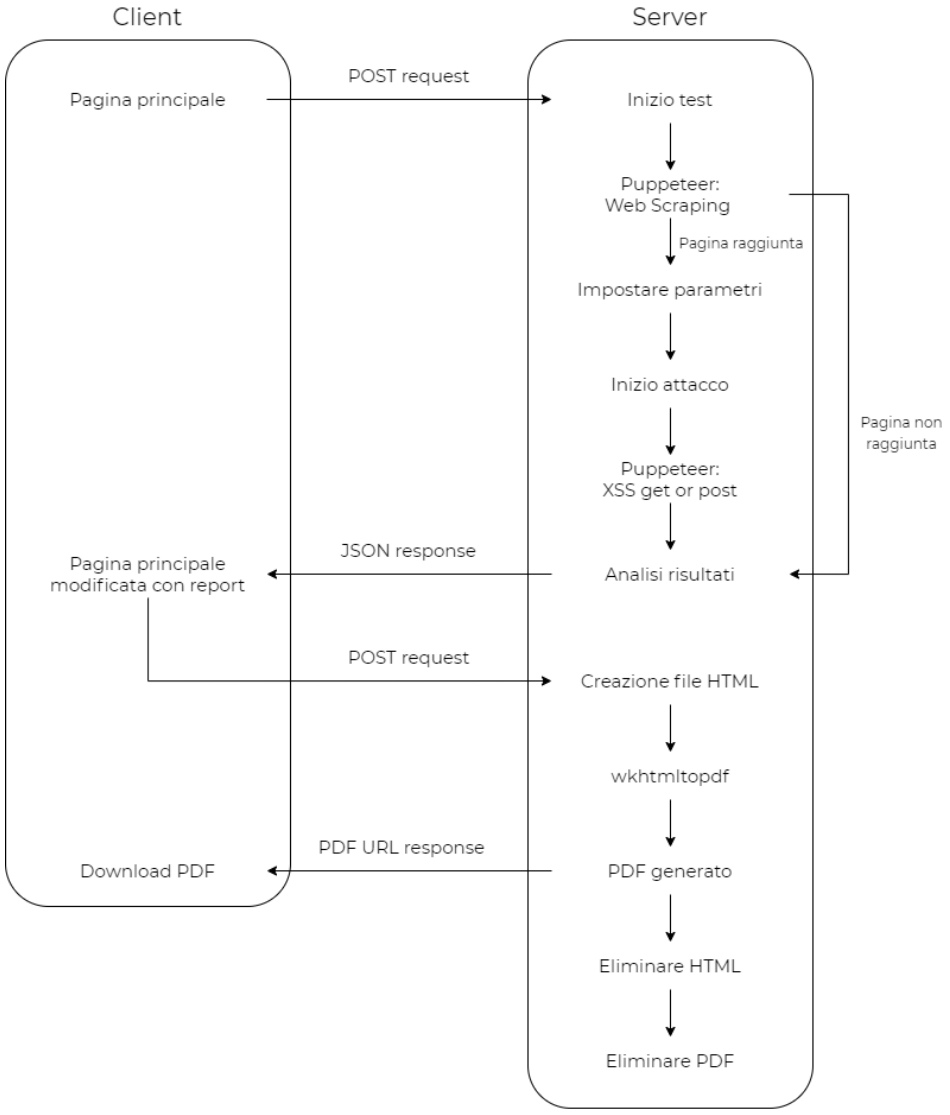
Architettura

Diagramma delle risorse



Pagina principale: index.html
Pagina reports: index.html (modificato tramite javascript)
Pagina informazioni: info.html

Flusso Dati



Codice

Tutto il codice di questo progetto è open source e disponibile sul GitHub.

Parti significative

Html

Il codice qui sotto è presente nella pagina delle informazioni. Ho dovuto utilizzare diverse entità HTML per poter mostrare agli utenti gli script utilizzati nei test dalla web app, senza però farli eseguire.

```
<code>&lt;script&gt;alert(0)&lt;/script&gt;</code>
<code>&lt;&lt;a&gt;script&gt;alert(0)&lt;/&lt;a&gt;script&gt;</code>
<code>';alert(0)//';alert(0)//&quot;;alert(0)//\&quot;;alert(0)//--&gt;&lt;/SCRIPT&gt;&quo
t;&gt;'&gt;&lt;SCRIPT&gt;alert(0)&lt;/SCRIPT&gt;=&{}</code>
<code>&lt;img src=x onerror=alert(0)&gt;</code>
<code>&gt;'&gt;&quot;&gt;&lt;img src=x onerror=alert(0)&gt;</code>
```

Css

Dark theme

Al fine di riuscire ad impostare automaticamente, in base alle impostazioni del dispositivo, il tema scuro ho dovuto utilizzare @media (prefers-color-scheme: dark)

```
@media (prefers-color-scheme: dark) {
    body{
        background-color: var(--backDark);
    }
    …
}
```

Loader

Per riuscire a creare ed animare il cerchio di caricamento ho dovuto usare un keyframes.

```
#loader{
    animation: spinner 0.9s linear infinite;
}
@keyframes spinner  {
    to{
        transform: rotate(360deg);
    }
}
```

Api

Questa funzione riceve in input il nome del file pdf che avrà dopo essere stato scaricato e il path del documento sul server. Aggiunge automaticamente alla pagine un link non visibile all'utente, scarica il pdf presente sul server e infine elimina l'elemento "a"

```
function downloadElement(filename, pdfURL){
    var element = document.createElement('a');
    element.setAttribute('href', pdfURL);
    element.setAttribute('download', filename);
    element.style.display = 'none';
    document.body.appendChild(element);
    element.click();
    document.body.removeChild(element);
}
```

Node.js

Test totale

Il codice sottostante rappresenta la funzione che viene eseguita su ogni singolo url ricevuto dal client. Nei primi passaggi si avvia la funzione getInfo, la quale serve per prendere tutte le informazioni (form action, form method, lista di input con relativi attributi) per poi svolgere gli attacchi XSS. In caso si riesca a raggiungere il sito si iniziano gli attacchi con la funzione xssAttack, avendo come scripts gli elementi dell'array scripts[].

```
async function xssTest(res, url){
  // create obj params
  var params = new p.Params(url);

  // scraping the web page the find all the data we need to perform the attack
  var data = await webScraping.getInfo(url);

  // if page is reached perform the attack
  if (data.comment != "Page not reached"){
    // set the params
    await p.setAfterScraping(params, data);

    // injected code
    // document.body.innerHTML=\"<p>XSS SUCCESSFUL</p>\"
    const scripts = [
      "<script>document.body.innerHTML='<p>XSS SUCCESSFUL</p>'</script>",
      "<<a>script>document.body.innerHTML='<p>XSS SUCCESSFUL</p>'</<a>script>",
```

```
      "';document.body.innerHTML='<p>XSS SUCCESSFUL</p>'//';document.body.innerHTML='<p>XSS
SUCCESSFUL</p>'//\";document.body.innerHTML='<p>XSS
SUCCESSFUL</p>'//\\\";document.body.innerHTML='<p>XSS
SUCCESSFUL</p>'//--></SCRIPT>\">'><SCRIPT>document.body.innerHTML='<p>XSS
SUCCESSFUL</p>'</SCRIPT>=&{}",
      "<img src=x onerror='document.body.innerHTML=\"<p>XSS SUCCESSFUL</p>\"'>",
      ">'>\"><img src=x onerror='document.body.innerHTML=\"<p>XSS SUCCESSFUL</p>\"'>"
    ];

    // check if attack is possible
    if (p.canXSS(params)){
      // attack with each script
      for(var i=0; i<scripts.length; i++){
        await xss.xssAttack(params, scripts[i], i).catch((err) =>{
          process.stdout.write("----------|FINISHED\n");
        });
      }
      // set params secure attribute
      p.isSecure(params);
    }
  }else{
    // set params comment attribute
    params.comment = data.comment;
  }
  console.log(params);
  console.log(params.input);

  // send data back to the web page
  res.json(params);
  res.end();
}
```

Attacco XSS

Con questa funzione vengono svolti gli attacchi ad un sito, sia che esso utilizzi il metodo get che il metodo post. Il tutto avviene tramite puppeteer che dopo aver aperto il sito con i codici infetti su chromium, controlla se la pagina web risulta essere vulnerabile al singolo script o meno.

```
async function xssAttack(params, inputValue, index){
    if (params.form.method === "get"){
        // method get attack
        var inputNameAll = "";
        for(var i=0; i<params.input.length; i++){
            // attack with every script
            var browser = await puppeteer.launch();//{headless: false}
            var att = params.input[i].name + "=" + urlEncode(inputValue);
```

```
            var data = await get(params.form.action, att, browser).catch(async (err) =>{
                // error with single input
                // check with all the inputs
                var nameAll = composeInputNameAll(params, inputValue);
                await xssGetAllAttack(params, nameAll, browser, index);
            });

            inputNameAll = endFor(params, data, i, index, inputNameAll, att, browser);
            await browser.close();
        }
        var browser = await puppeteer.launch();//{headless: false}
        await xssGetAllAttack(params, inputNameAll, browser, index);
    }else if (params.form.method === "post"){
        // method post attack
        var inputNameAll = "";
        for(var i=0; i<params.input.length; i++){
            // attack with every script
            var browser = await puppeteer.launch();//{headless: false}
            var att = params.input[i].name + "=" + urlEncode(inputValue);
            var data = await post(params.form.action, att, browser).catch(async (err) =>{
                // error with single input
                // check with all the inputs
                var nameAll = composeInputNameAll(params, inputValue);
                await xssPostAllAttack(params, nameAll, browser, index);
            });

            inputNameAll = endFor(params, data, i, index, inputNameAll, att, browser);
            await browser.close();
        }
        var browser = await puppeteer.launch();//{headless: false}
        await xssPostAllAttack(params, inputNameAll, browser, index);
    }
}
```

Creare il pdf

Questa funzione ha il compito di creare e indentare il pdf contenente uno o più report. Come primo passaggio viene creato un file html con relativi css, il quale andrà in input al programma wkhtmltopdf.exe che genererà il pdf successivamente scaricato dal utente. Entrambi i file html e pdf verranno eliminati dopo lo scadere di un timer, al fine di occupare il minor spazio su disco possibile.

```
function generatePDF(html){
    // format pdfText for batch
    var text = head() + header() + "<body>" + html + "</body>" + footer();
    var c = countFileName;
    countFileName++;
```

```
    var htmlFile = ".\\htmlToPDF\\htmlReport" + c + ".html";
    var fileLocation = '.\\public\\pdf\\XSS Security Report' + c +'.pdf'
    var command = 'wkhtmltopdf.exe ' + htmlFile +  ' "' + fileLocation + '"';

    // make file html
    fs.writeFileSync(htmlFile, text, (err) => {
        if (err) {
            if(err != null){
                console.error(err)
            }
            return;
        }
    });

    // run command
    exec(command, (error) => {
        if (error) {
            console.error(error);
        }
    });

    var timer = 30;//seconds
    setTimeout(() => {
        fs.unlink(htmlFile, (err) => {
            if (err) {
                if(err != null){
                    console.error(err)
                }
                return
            }
        });
        console.log("File deleted: " + htmlFile);
    }, timer * 1000);

    return fileLocation;
}
```

Conclusioni

Questo progetto è stato molto interessante e importante da sviluppare, non solo per una crescita personale ma anche per il possibile utilizzo in ambito lavorativo. Sicuramente, col passare del tempo e migliorando le mie tecniche di bug bounty, seguiranno aggiornamenti dell'applicazione.
Non avendo mai provato a creare applicazioni per il web è risultato molto impegnativo, sia per il front-end, gestendo animazioni e transizioni, che per il back-end, non avendo mai approfondito il linguaggio javascript e non avendo mai provato ad utilizzare librerie esterne ad esso. La parte più complicata è stata certamente riuscire ad automatizzare gli attacchi XSS, che alle origini di questo progetto era per me l'ostacolo più grande.

Nota bibliografia e sitografia

Html, Css e Javascript

W3school	https://www.w3schools.com/
Mozilla Developer	https://developer.mozilla.org/en-US/
Node.js	https://nodejs.org/en/
Express	https://expressjs.com/
Puppeteer	https://pptr.dev/

Cross-Site Scripting

Owasp	https://owasp.org/www-community/attacks/xss/
Google XSS game	https://xss-game.appspot.com/

Convertire HTML in PDF

Wkhtmltopdf	https://wkhtmltopdf.org/