VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

NÁSTROJ NA ANALÝZU JAVASCRIPTU PRE DETEKCIU DOM XSS ZRANITEĽNOSTÍ VO WEBOVÝCH APLIKÁCIACH

TOOL FOR ANALYSIS OF JAVASCRIPT TO DETECT DOM XSS VULNERABILITIES IN WEB APPLICATIONS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **DIANA BARNOVÁ**
AUTHOR

VEDOUCÍ PRÁCE **Ing. IVAN HOMOLIAK, Ph.D.**
SUPERVISOR

BRNO 2021

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav inteligentních systémů (UITS) Akademický rok 2020/2021

Zadání bakalářské práce

Studentka: **Barnová Diana**
Program: Informační technologie
Název: **Nástroj na analýzu JavaScriptu pro detekci DOM XSS zranitelností ve webových aplikacích**
Tool for Analysis of JavaScript to Detect DOM XSS Vulnerabilities in Web Applications
Kategorie: Bezpečnost

Zadání:

1. Seznamte se s problematikou DOM-based Cross-Site Scripting (XSS) útoků na webové aplikace.
2. Nastudujte možnosti detekce DOM-based XSS zranitelností ve webových aplikacích.
3. Navrhněte způsob detekce DOM-based XSS zranitelností.
4. Navrhněte nástroj pro analýzu Javascriptu a identifikaci DOM-based XSS zranitelností.
5. Implementujete navržený nástroj pro evaluaci webových aplikací na zranitelností DOM-based XSS.
6. Eticky otestujte vytvořen nástroj na vhodném vzorku webových aplikací.

Literatura:

- STUTTARD, Dafydd a Marcus PINTO. *The Web Application Hacker's Handbook: Finding and Exploiting Security Flaws*. Second Edition. Indianapolis: John Wiley, 2011. ISBN 978-1-118-02647-2.
- KIM, Peter. *The hacker playbook 3: Practical Guide to Penetration Testing*. 3. North Charleston (SC): Secure Planet, 2018. ISBN 9781980901754.

Pro udělení zápočtu za první semestr je požadováno:

- Body 1 až 4.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Homoliak Ivan, Ing., Ph.D.**
Vedoucí ústavu: Hanáček Petr, doc. Dr. Ing.
Datum zadání: 1. listopadu 2020
Datum odevzdání: 12. května 2021
Datum schválení: 11. listopadu 2020

Abstrakt

Cieľom práce bolo navrhnúť nástroj na analýzu JavaScriptu za účelom detekovať zraniteľnosť DOM–based XSS vo webových aplikáciách, následne ho implementovať a eticky otestovať. Cross–site Scripting (XSS) je jedným z najbežnejších injekčných útokov na webové aplikácie, ktorý vkladá škodlivý kód do inak dôveryhodnej stránky. Na detekciu a následnú exploatáciu DOM–based XSS zraniteľností je potrebná interpretovaná odpoveď prehliadačom preto navrhnutý nástroj odchytáva odpoveď z proxy serveru Burp Suite. Analýza tejto odpovedi využíva dva samostatné regulárne výrazy zamerané na vyhľadávanie vstupov (sources) a výstupov (sinks) v zdrojovom kóde odpovede. Pomocou sady payloadov sa zistí, či je stránka exploitovateľná. Následne je užívateľ upozornený na možné nebezpečenstvo. Výstupom je textový súbor so sumarizáciou výsledkov pre danú URL.

Abstract

The main goal of this thesis is to design a tool for analisys of JavaScript to detect DOM–based XSS vulnerability in web applications. Then to implement it and test it ethically. Cross–side Scripting (XSS) is one of the most common injection attacks on web applications that insert malicious code in an otherwise trusted site. An interpreted response by the browser is required for the detection and subsequent exploitation of DOM–based XSS vulnerabilities, therefore the tool captures the response from the Burp Suite proxy server. The analysis of this response uses two separate regular expressions aimed at searching for sources and sinks in the source code of the response. A set of payloads is used to determine if a site is exploitable. Subsequently, the user is warned of the possible danger. The output is a text file summarizing the results for the URL.

Kľúčové slová

XSS, DOM–based, HTTP, zraniteľnosť, útok, skriptovanie, detekcia zraniteľnosti, JavaScript

Keywords

XSS, DOM–based, HTTP, vulnerabilities, attack, scripting, detection of vulnerabilities, JavaScript

Citácia

BARNOVÁ, Diana. *Nástroj na analýzu JavaScriptu pre detekciu DOM XSS zraniteľností vo webových aplikáciach.* Brno, 2021. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Ivan Homoliak, Ph.D.

Nástroj na analýzu JavaScriptu pre detekciu DOM XSS zraniteľností vo webových aplikáciach

Prehlásenie

Prehlasujem, že som túto bakalársku prácu vypracovala samostatne pod vedením Ing. Ivana Homoliaka, Ph.D. Ďalšie informácie mi poskytol Martin Koppon, MSc. Uviedla som všetky literárne pramene, publikácie a ďalšie zdroje, z ktorých som čerpala.

.....
Diana Barnová
28. júla 2021

Poďakovanie

Rada by som sa poďakovala vedúcemu bakalárskej práce pánovi Ing. Ivanovi Homoliakovi, Ph.D. za možnosť pracovať na individuálnej téme, za odborné vedenie, konzultácie, trpezlivosť a podnetné návrhy k práci.

Tiež by som sa rada poďakovala odbornému konzultantovi a zadávateľovi práce Martinovi Kopponovi, MSc., ktorý venoval čas a odborné rady tomu, aby ma naviedol na správny smer a poskytol mi cenné informácie, vďaka ktorým som porozumela problematike tejto práce.

Obsah

Kapitola 1

Úvod

V reálnom svete vidíme viacero prípadov zraniteľnosti webu. Pod pojmom zraniteľnosť rozumieme programátorskú chybu v softvéri alebo hardvéri, ktorá spôsobuje bezpečnostné riziko. Ak útočník toto miesto nájde a zneužije, nastáva tzv. *exploit*. Príčin takýchto chýb môže byť niekoľko a poväčšine platí, že čím je aplikácia robustnejšia tým je náchylnejšia na zraniteľnosti. To potvrdzuje aj známy odborník na webovú bezpečnosť, Mikko Hyppönen[1] v Hyppönenovom zákone: *Kedykoľvek sa zariadenie označí ako „inteligentné“, je zraniteľné [14].*

Jedným z najbežnejších útokov na webové aplikácie je tzv. Cross-site Scripting (XSS). Ide o typ injekčného útoku, ktorý vkladá nechcený a väčšinou škodlivý kód do inak dôveryhodnej stránky. Podstatou XSS útoku je schopnosť útočníka poslať škodlivý kód koncovému užívateľovi, pomocou webovej aplikácie. Tento škodlivý kód je väčšinou vo forme skriptu spúšťaného na strane prehliadača. Nanešťastie, chyby umožňujúce úspešné vykonanie týchto útokov sú relatívne dosť rozšírené, keďže sa môžu vyskytovať kdekoľvek, kde webová aplikácia narába s dátami poskytnutými užívateľom.

I. a II. druh XSS zraniteľnosti používa dáta kontrolované užívateľom a zobrazuje ich spätne užívateľovi cez zabezpečenú cestu - server. III. druh XSS zraniteľnosti - DOM–based má inú stratégiu útoku: JavaScript na klientovej strane má prístup do DOM prehliadača a tým pádom vie ovplyvniť URI^A, ktoré sa používa na načítanie danej webovej stránky. Ak aplikácia extrahuje dáta z URI, vykoná nejaké operácie s týmito dátami a potom ich použije na dynamické znovu-načítanie obsahov stránky, môže byť zraniteľná na DOM–based XSS. Napríklad, keď priamo zoberie hodnotu nejakého parametru z URI a zapíše ju do HTML dokumentu, tak tento kód bude dynamicky zapísaný do stránky a vykonaný rovnako ako keby bol prijatý zo serveru.[31]

V takomto stave má útočník prístup ku cookies užívateľa, relačnému tokenu alebo iným citlivým informáciám uchovávaným prehliadačom v súvislosti s danou stránkou. Dôležitú úlohu tu hrá aj JavaScript, ktorý je čím ďalej, tým viac využívaný vo webových aplikáciách. Existencia a nevyhnutnosť DOM vo webových aplikáciach súvisí aj s týmto faktom. Na správnu komunikáciu medzi programovacími jazykmi, či už HTML a JavaScript alebo pri použití viacerých verzií JavaScriptu sa využíva práve DOM.

V tejto práci sa venujem útoku DOM–based XSS a podrobne rozoberám zraniteľnosti vedúce k úspešnému vykonaniu útoku. Rozoberám typy XSS útokov, so zameraním hlavne na DOM–based XSS útok a ako im predísť. Ide o bežnejší typ útoku avšak menej intuitívny na pochopenie. Následne vysvetľujem problém detekcie a prevencie pred týmito útokmi a

[1] https://mikko.com/

význam JavaScriptu v kontexte tohto problému. Opisujem moje riešenie pre identifikáciu útoku vo webových aplikáciách a na záver sa venujem vyhodnocovaniu etických experimentov vykonaných na referenčných weboch.

Kapitola 2 je zameraná všeobecne na bezpečnosť aplikačnej vrstvy, protokol HTTP a niektoré jeho komponenty nevyhnutné na fungovanie webu a webových aplikácií. Nasledujúce dve kapitoly sa špecifikujú postupne na zraniteľnosť Cross–Site Scripting (XSS) (kap. 3) jeho typy a prevenciu proti útokom tohoto typu a v ďalšia kapitola sa zameriava už na konkrétny typ XSS zraniteľnosti - DOM–based (kap. 4). Kapitola 5 predstavuje môj navrhnutý nástroj a tiež opisuje niektoré už existujúce riešenia. V kapitole 6 sa venuje bližšie implementácií môjho nástroja *DOM–based Catcher* a tiež jeho etickému penetračnému testovaniu.

Kapitola 2

Bezpečnosť aplikačnej vrstvy

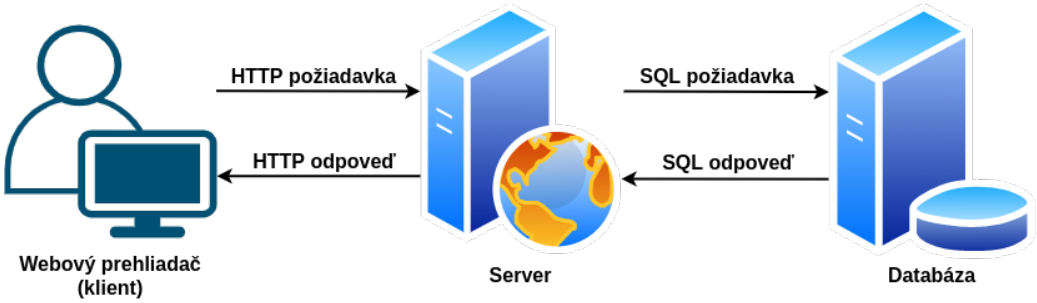
Aplikačná vrstva, ktorá tvorí najvyššiu vrstvu referenčného modelu OSI, definuje užívateľské procesy a aplikácie komunikujúce po sieti. Obsahuje veľké množstvo aplikačných protokolov od rôznych tvorcov a preto si každý protokol zodpovedá za svoju bezpečnosť samostatne.

Odlišné typy architektúr sieťových aplikácii reprezentujú odlišné príležitosti pre vznik zraniteľností v systéme. Preto, pre lepšie pochopenie bezpečnosti aplikácií a teda aj ich zraniteľností, je potrebné sa oboznámiť s ich architektúrou. Existuje viacej typov klient–server architektúr založených na vrstvách s ktorých sa skladajú:

- *Jedno–stupňová architektúra*, ktorá zahrňuje klienta, server a databázu na jednom počítači.
- *Dvoj–stupňová architektúra*, ktorá má klienta a server spojené na jednom počítači a komunikuje s databázovým serverom.
- *Troj–stupňová architektúra*, ktorá je založená na troch základných komponentoch: webový prehliadač (client), webový server (web application server) a databázový server.
- *N–stupňová architektúra*, ktorá zahrňuje delenie aplikácií do troch rôznych vrstiev: logická vrstva, prezentačná vrstva a databázová vrstva.

Výhod troj-stupňovej architektúry je niekoľko, napríklad intuitívnosť grafického rozhrania, flexibilita databázového systému či možnosť využitia jedného databázového serveru pre viaceré klientské systémy[27]. Pre webové aplikácie sa zväčša používa práve troj–stupňový typ architektúry[10] (viď. obr. 2.1):

- *Webový prehliadač (client)* sa stará o zobrazujúcu logiku (prezentačná vrstva). Táto logika má na starosti kontrolu cesty (URL), ktorou užívateľ komunikuje s aplikáciou a tiež kontroluje (validuje) užívateľský vstup. Väčšinou sa používajú programovacie jazyky HTML^A, CSS^A a JavaScript.
- *Webový server* alebo aplikačná vrstva, je jadrom celej architektúry. Informácie zozbierané vo webovom prehliadači sa tu spracujú a prípadne, pošle dotaz na databázový server. Sformuluje odpoveď a potom pošle naspäť klientovi. Väčšinou sa tu používa programovací jazyk ako Python, Java, PHP, ...
- *Databázový server* alebo dátová či databázová vrstva, uchováva a spracúva dáta pre aplikácie. Príkladom pre dátovú vrstvu, môže byť systém správy relačných databáz MySQL kde sa používa hlavne programovací jazyk SQL.



Obr. 2.1: Troj–stupňová architektúra webových aplikácií

2.1 HTTP[A]

Základným protokolom webových aplikácií, nad ktorým sa komunikácia klient–server odohráva je protokol HTTP. Pracuje nad TCP[A] protokolom s rezervovaným portom číslo 80. Nepoužíva šifrovanie, vďaka čomu je rýchly ale zároveň sa väčšinou používa len keď sa nenachádzajú na stránke nijaké citlivé informácie, ako napríklad pri blokoch. Pracuje nad aplikačnou vrstvou. Princíp komunikácie funguje na základe žiadosť/odpoveď, kde klient pošle požiadavku (angl. *request*) a server na každú danú požiadavku posiela odpoveď (angl. *response*). Tieto správy sa posielajú v textovom formáte ako sekvencia znakov. Táto kapitola je písaná podľa normy RFC2616[11], RFC 2818[29].

2.1.1 Požiadavka klienta (2.1)

```
Request = Request-Line *((general-header | request-header | entity-header)
                         CRLF) CRLF [ message-body ]

Request-Line = Method SP Request-URI SP HTTP-Version CRLF
```

Prvý riadok (*Request–Line*) obsahuje použitú metódu, ďalej nasleduje URI[A] a verzia použitého protokolu. Tieto informácie sú medzi sebou oddelené znakmi `SP` a na konci tohto riadku sa nachádza sekvencia znakov `CRLF`. Hodnota v metóde hovorí o tom, čo sa má vykonať zo zdrojom identifikovaným pomocou URI. Táto hodnota je `case-sensitive`. Môže obsahovať jednu z nasledujúcich možností:

- **OPTIONS** – informácie o komunikačných možnostiach dostupných pre danú URI požiadavku.
- **GET** – načítať akékoľvek informácie, ktoré sú identifikované pod URI požiadavkou (získanie tela správy ako napríklad HTML stránka, atď.).
- **HEAD** – má rovnaký účel ako metóda GET ale s tým rozdielom, že server nevráti telo správy (získanie HTTP hlavičky). Používa sa na získanie meta–informácií o danej entite.
- **POST** – server príjme posielané dáta a ďalej ich spracováva.
- **PUT** – entita uložená pod danú URI požiadavku. V prípade ak táto URI už odkazuje na existujúci zdroj, nová entita je považovaná za modifikovanú verziu pôvodnej entity.

- **DELETE** – server zmaže zdroj existujúci pod URI požiadavkou.
- **TRACE** – posiela kópiu obdržanú serverom naspäť klientovi (tzv. debugging).
- **CONNECT** – dynamické vytvorenie komunikačného tunela medzi klientom a serverom za použitia proxy serveru.

Request–URI musí byť zahrnutá v požiadavke. V prípade, že v pôvodnom URI sa nenachádza adresa zdroja na pôvodný server alebo gateway potom sa tu nachádza znak „/“ čo označuje koreň serveru. Koncový riadok (záver HTTP hlavičky) je prázdny a tiež ukončený reťazcom `CRLF` (tzn. na konci sa nachádza reťazec `CRLFCRLF`).

```
1 GET /Task/Rule1 HTTP/1.1
2 Host: www.insecurelabs.org
3 Upgrade-Insecure-Requests: 1
4 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko
      ) Chrome/90.0.4430.212 Safari/537.36
5 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
6 Accept-Encoding: gzip, deflate
7 Accept-Language: en-US,en;q=0.9
8 Connection: close
```

Príklad 2.1: Ukážka hlavičky požiadavky klienta

2.1.2 Odpoveď serveru (2.2)

```
Response = Status-Line *((general-header | response-header | entity-header)
                        CRLF) CRLF [ message-body ]

      Status-Line = HTTP-Version SP Status-Code SP Reason-Phrase CRLF
```

Prvý riadok (*Status–Line*) definuje verziu používaného protokolu (napr. HTTP/1.1) nasleduje číselný návratový kód a jeho slovné pomenovanie. Návratová hodnota je trojciferné číslo a môžeme ju rozdeliť do piatich základných kategórii:

- **Informačný** – návratový kód v tvare `1xx` znamená, že žiadosť prijatá alebo proces pokračuje (napr. „100 - Continue“)
- **Úspech** – návratový kód v tvare `2xx` znamená, že žiadosť bola úspešne prijatá, pochopená a vybavená (napr. „200 - OK“)
- **Presmerovanie** – návratový kód v tvare `3xx` znamená, že na dokončenie žiadosti je potrebné podniknúť ďalšie kroky (napr. „301 - Moved Permanently“)
- **Chyba na strane klienta** – návratový kód v tvare `4xx` znamená, že žiadosť obsahuje nesprávnu syntax alebo ju nemožno splniť (napr. „404 - Not Found“)
- **Chyba na strane serveru** – návratový kód v tvare `5xx` znamená, že serveru sa nepodarilo splniť platnú požiadavku (napr. „502 - Bad Gateway“)

Každý element je oddelený znakmi `SP` a na konci riadku je sekvencia znakov `CRLF`.

```
HTTP/1.1 200 OK
Server: nginx
Date: Thu, 10 Jun 2021 15:44:00 GMT
Content-Type: text/html; charset=utf-8
Connection: close
Vary: Accept-Encoding
Cache-Control: private
X-XSS-Protection: 0
Content-Length: 1176

<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title></title>
    <link href="/Content/Task.css" rel="stylesheet" type="text/css" />
    <script src="/Scripts/jquery-1.5.1.min.js" type="text/javascript"></script>
    <script src="../../Scripts/Task.js"></script>
</head>
<body>
    <div class="page">
        <section id="main">
            <a href="/Task" class="back">Back to task list</a>

    <h1>Task 1 - between tags</h1>

    <form method="get" action="">
        Search: <input type="text" name="query" value=""/><input type="submit" value="
    Search" />
    </form>
    <br /><br />
    <div class="wrong solution">
        <b>Wrongly escaped HTML:</b>
        <pre>Searched for <i></i></pre>
    </div>

    <div class="right solution">
        <b>Correctly escaped:</b>
        <div>
            Searched for
        </div>

        <b>Correctly escaped HTML:</b>
        <pre>Searched for <i></i></pre>
    </div>
        </section>
    <footer>
        www.insecurelabs.org is an educational tool. It's hosted by <a href="http://
    appharbor.com">AppHarbor</a>
    </footer>
    </div>
</body>
</html>
```

Príklad 2.2: Ukážka odpovede servera klientovi

2.1.3 HTTPS[A]

To, či je použitý HTTP alebo HTTPS napríklad na načítanie stránky je známe z formátu URI, keď pri HTTPS sa URI začína „*https://...*“. Používa sa na bezpečný prenos obsahu cez HTTP. Správy sa prenášajú cez vytvorený bezpečnostný tunel, ktorý je zabezpečený pomocou TLS[A]. Dochádza teda k uzatvoreniu šifrovaného spojenia SSL[A] certifikátom, ktorý konvertuje dáta do zašifrovanej podoby (asymetrickou šifrou). Implicitne komunikuje prostredníctvom protokolu TCP a cez rezervovaný port 443. Je pravda, že bez šifrovania je obsah nezabezpečený voči prípadným útokom ale výhodou HTTP stále ostáva rýchlosť, ktorá je niekedy viac potrebná. HTTPS pracuje nad transportnou vrstvou. Používa sa hlavne pri stránkach, ktoré odposielajú a prijímajú citlivé informácie ako napríklad čisla kreditnej karty či rôzdne heslá. Presná špecifikácia je definovaná na v RFC 2818 [29].

2.2 URI

Táto sekcia je písaná podľa štandardov RFC3986 [5]. Uniform Resource Identifier (URI) je kompaktná sekvencia znakov, ktorá identifikuje abstraktný alebo fyzický zdroj. URI sa skladá z URL (Uniform Resource Locator) a URN (Uniform Resource Name) kde URL označuje adresu konkrétneho uzlu v sieti a URN určuje zdroj menom bez uvedenia jeho umiestnenia. Keďže URI musí obsahovať len ASCII znaky, bol vytvorený IRI (Internation Resource Identifier), ktorý zabezpečuje podporu pre viac bajtovú kódovú sadu unicode. Generická syntax URI pozostáva z hierarchickej postupnosti komponentov (Obr. 2.2):

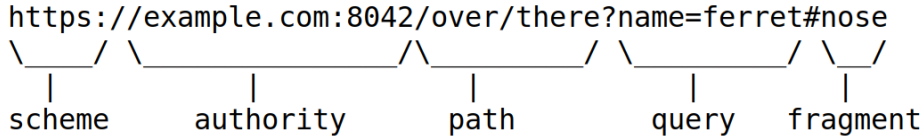
- *Schéma* označuje špecifikáciu na prideľovanie identifikátorov, ktoré sa majú použiť. V prípade webového servera to býva buď http alebo https.

- *Hier-part*, ktorá zahŕňa autoritu a cestu. Začiatok autority je dvomi za sebou idúcimi znakmi lomky („//“) a ukončený je ďalším znakom lomky („/“), za ktorým pokračuje hierarchická cesta, alebo znakom otáznik („?“), znakom mriežky („#“) či skončením adresy URI.

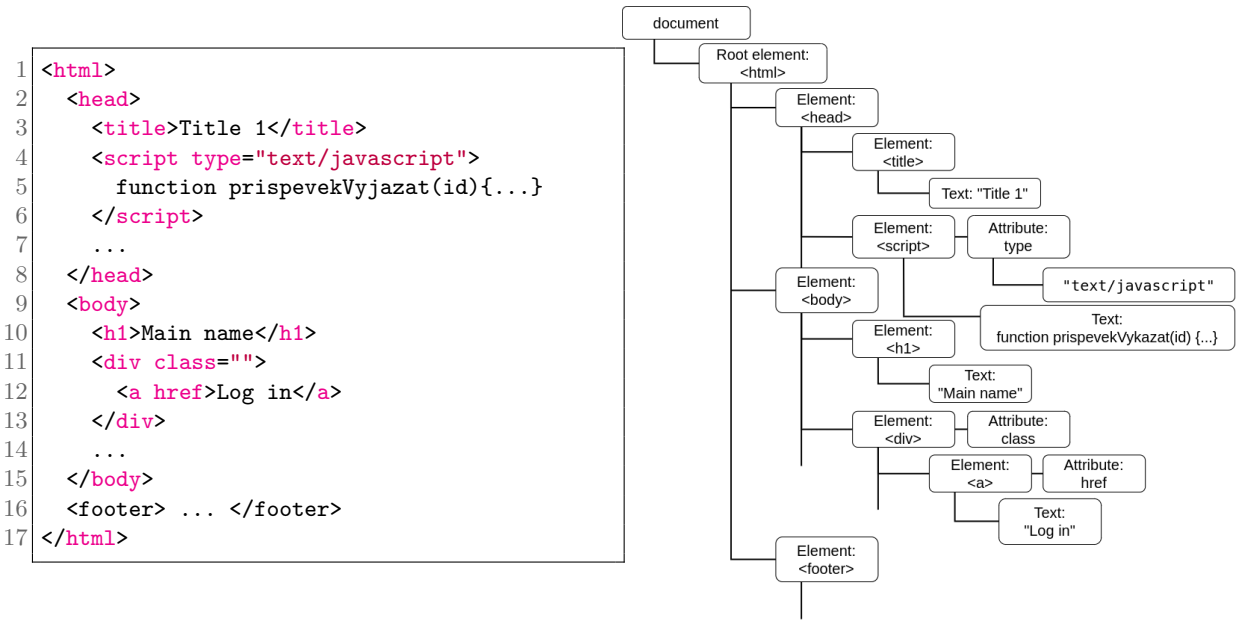
- *Query* časť URI začína znakom otáznika („?“). Obsahuje nehierarchické údaje, ktoré spolu s údajmi v komponente hierarchickej cesty slúžia na identifikáciu zdroja v rámci rozsahu schémy URI a autority (ak existujú).

- *Fragment* umožňuje nepriamu identifikáciu sekundárneho zdroja odkazom na primárny zdroj prípadne ďalšími identifikačnými informáciami.

V prípade, že by URI stránky v príklade na obr. 2.2 nebolo zabezpečené proti útoku XSS, útočníkovi by stačilo napríklad v *query* komponente s názvom `name` vložiť škodlivý kód na miesto slova „ferret“.



Obr. 2.2: Príklad adresy URI



Obr. 2.3: Príklad HTML kódu a jeho reprezentácia v DOM

2.3 DOM

Webová stránka je dokument, s ktorým je potrebné dynamicky a ľahko pracovať. Na zobrazenie kódu webovej stránky sú dve možnosti: buď v okne vyhľadávača (browser window) alebo ako zdrojový kód HTML jazyka. K tomu aby sa dalo prehľadne pracovať sa HTML kódom dopomáha aj *Document Object Model* – DOM. DOM je dynamická, dátová, stromová štruktúra, ktorá v sebe definuje prvky webových aplikácií a tvorí interface pre HTML a XML dokumenty. Vo svojej podstate je DOM objektovo-orientované zobrazenie webovej stránky, ktoré podporuje multiplatformové konvencie, ktoré sú nezávislé na jazyku pre prácu a reprezentáciu objektov v HTML, XHTML alebo XML dokumentoch. Prvky sú uzlami stromu a obsahujú referenciu na elementy pod nimi. Tieto uzly sú definované v HTML kóde a ten zase zahrnutý v CSS kóde. S hodnotami, ktoré sú uložené v DOM sa môže manipulovať aj vďaka JavaScriptu cez DOM API. Tá sa skladá z dvoch skupín:

- metódy prístupu k DOM (napr. getElementById(),...)
- metódy pre update hodnôt v DOM (napr. setAttribute(),...).

Základnou jednotkou DOM modelu, ako je vidieť na obrázku 2.3 je element. JavaScript vie pracovať s týmito elementami ako aj s ich atribútmi, presnejšie s hodnotami v nich. Práve pri modifikovaní DOM môže dôjsť k zraniteľnosti, keď sa zmení kontext obsahu v jednotlivých uzloch alebo v ich atribútoch. Medzi najčastejšie spôsoby vkladania kódu v jazyku JavaScript do webových stránok patria:

- *v jazyku HTML medzi párový tag <script>*
- *načítanie kódu v jazyku JavaScript z externého súboru*
- *In-line skripty* – Tento spôsob používa hodnoty atribútov iného tagu pre vloženie kódu v jazyku JavaScript. Tieto atribúty sú pomenované podľa udalostí, ktoré ich spúšťajú (napr. načítanie webovej stránky, stlačenie tlačidla, atď.).

- *Bookmarklety* – Znamená písanie kódu v jazyku JavaScript priamo do adresového riadku za direktívu `javascript:` (podobne ako je to pri protokole HTTP, či FTP). Tieto skripty sú označenie ako *self-contained*.

2.4 JavaScript[33]

Vzhľadom k tomu, že JavaScript je jeden z najviac rozšírených a používaných programovacích jazykov, jeho vývoj je veľmi dynamická záležitosť. Vznikol v roku 1995 ako dôsledok snahy o vytvorenie skriptovacieho jazyka, ktorý je dostatočne výkonný a zároveň je podobný jazyku C či C++. Neskôr sa stal štandardom pre ECMAScript a v súčastnosti je jeho najnovší update ECMAScript 2021[1].

Pod výrazom JavaScript v kontexte webových aplikácií rozumieme viacero prvkov: jedným z nich je zoskúpenie webových API^A, ktoré zahrňuje aj DOM. Práve prítomnosť JavaScriptu v DOM umožňuje vytvorenie interaktívneho webu alebo webovej aplikácie. Základom stále ostáva jazyk HTML, ktorý má na starosti štruktúru dokumentu a jeho obsah a jazyk CSS, ktorý zabezpečuje vzhľad a formátovanie stránky.

2.4.1 JavaScript v HTML

JavaScriptový kód sa do HTML stránky vkladá cez element `<script>` a je potrebné nastaviť atribút `type` pre identifikáciu skriptovacieho jazyka na `"text/javascript"`. Pomocou tohoto elementu sa dá vkladať priamo skript alebo špecifikovať URL externého súboru kde sa nachádza a to pomocou atribútu `src`. Skripty sa vykonávajú paralelne v poradí v akom sú napísané t.j. na vykonanie ďalšieho JavaScriptového kódu je potrebné dokončiť predošlý.

Hlavnou výhody vkladania JavaScriptu cez externý súbor je rýchlosť načítavania stránky ako aj aktualizácie stránky (napríklad ak je potrebné modifikovať JavaScript, stačí upraviť jeden súbor a nie všetký stránky, ktoré ho používajú)[30].

2.4.2 Obmedzenia JavaScriptu

JavaScript je programovací jazyk, ktorý je interpretovaný na strane klienta. Toto spôsobuje veľké obmedzenia pre vývojárov, keďže pre zachovanie bezpečnosti stránky nie je možné, napríklad využiť iba JavaScript na naprogramovanie počítadla návštevníkov stránky za účelom spracovania a uchovania informácií o chode webovej stránky. Avšak aj cez takéto významné obmedzenie je JavaScript veľmi využívaný (napríklad dáta o aktuálnom čase a dátume)[30].

2.5 Top 10 bezpečnostných rizík vo webových aplikáciách

Nezisková nadácie OWASP^A, ktorej hlavnou úlohou je zlepšiť bezpečnosť softwarov, pravidelne zostavuje rebríček najčastejších útokov a teda aj najväčšie zraniteľnosti webových aplikácií. Posledná aktualizácia je z roku 2017[25].

1. **Injection.** Toto riziko nastane keď nedôveryhodné dáta sa pošlú interpréterovi aplikácie napríklad ako súčasť príkazu. Útočník môže cez tieto dáta oklamať interpretéra tak aby dosiahol spustenie svojho kódu a získať tak napríklad prístup k dátam bez

[1] https://backbencher.dev/javascript/es2021-new-features

patričnej autorizácie. (SQL^A Injection, NoSQL Injection, OS Injection, and LDAP^A Injection)

2. **Broken Authentication.** Častou chybou je nesprávna implementácia funkcií v aplikáciach, ktoré súvisia s autentifikáciou a správou relácii. Takéto chyby dovoľujú útočníkovi dostať sa ku heslám, napríklad pri nesprávnom obnovení zabudnutého hesla. Prípadne nájsť iné nedostatky v implementácií napríklad na zneužitie používateľovej identity či už dočasne alebo na trvalo.

3. **Sensitive Data Exposure.** Mnohé API či webové aplikácie nesprávne chránia citlivé informácie ako napríklad financie či informácie o zdravotnej starostlivosti. Pokiaľ nie sú takéto dáta dobre zabezpečené napríklad šifrovaním, môže ich útočník ľahko ukradnúť či modifikovať (finančný podvod, krádež identity, ...).

4. **XML External Entities (XXE).** Pri nesprávnej alebo zastaralej konfigurácií XML procesory vyhodnocujú externé entity vrámci XML dokumentov, čo môže spôsobiť prístup k interným dátam, skenovanie portov v internej sieti, vzdialené vykonávanie kódu alebo odopretie služby (napr. DoS).

5. **Broken Access Control.** Overenia autorizácie užívateľa je častokrát slabo kontrolované. Toto sa dá veľmi jednoducho zneužiť útočníkom na prístup k neautorizovaným funkcionalitám alebo dátam. Napríklad prístup k užívateľským účtom, zmena v právach užívateľa, prístup k citlivým dátam, atď.

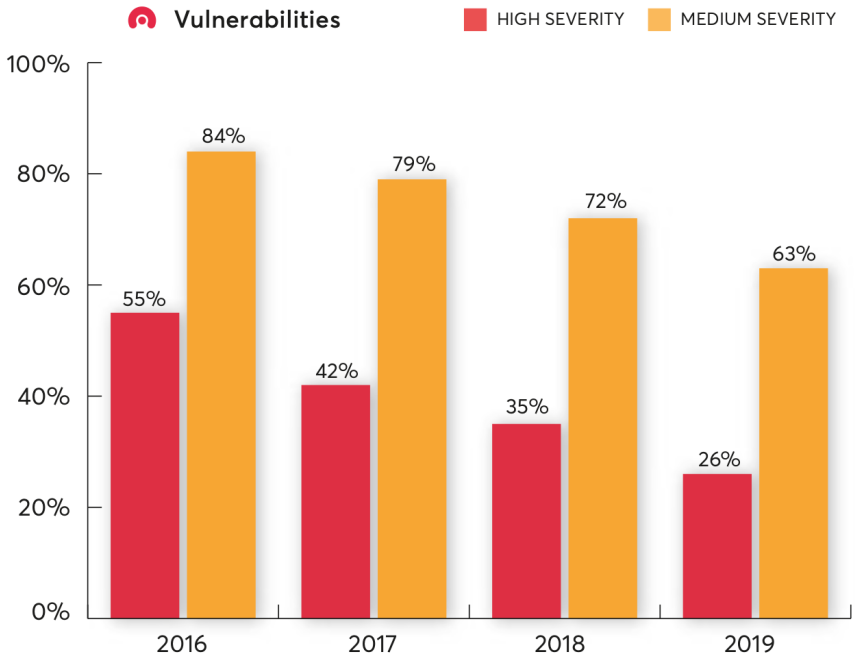
6. **Security Misconfiguration.** Ďalšia chyba je zlá bezpečnostná konfigurácia. Častým dôvodom sú predvolené nastavenia, ktoré nemusia byť bezpečné (napríklad ponechanie pôvodného hesla), nekompletná konfigurácia, atď. Okrem bezpečného nakonfigurovania sú tiež podstatné pravidelné aktualizácie.

7. **Cross-Site Scripting (XSS).** K tomuto typu útoku dochádza keď aplikácia vloží nedôveryhodné dáta od užívateľa do HTML kontextu stránky alebo JavaScriptu, bez predošlej validácie dát. Bližšie sa zraniteľnosti XSS^A venuje kapitola 3.

8. **Insecure Deserialization.** Najčastejšie vedie táto chyba ku vzdialenému spusteniu kódu ale je tu možnosť aj iných útokov ako napríklad replay útok (Replay attack), útok cez injektáž kódu (Injection attack) či útok na eskaláciu privilégií.

9. **Using Components with Known Vulnerabilities.** Komponenty, ktoré sa používajú na vytváranie webových aplikácií ako napríklad knižnice, frameworky alebo iné softwarové moduly, sú spúšťané s rovnakými právami ako aplikácie. Zraniteľnosti jednotlivých komponentov sa dá overiť napríklad pomocou verejne publikovanej databáze NVD[23].

10. **Insufficient Logging & Monitoring.** Nedostatočné logovanie a monitorovanie, spojené s chýbajúcim alebo nedostatočným riešením bezpečnostných incidentov, umožňuje útočníkovi postupovanie v útoku na ďalšie systémy, zabezpečenie perzistencie útoku, exfiltovanie dát alebo ich zničenie. Mnohé štúdie poukazujú na to, že zistenie takéhoto útoku môže trvať aj viac ako 200 dní.

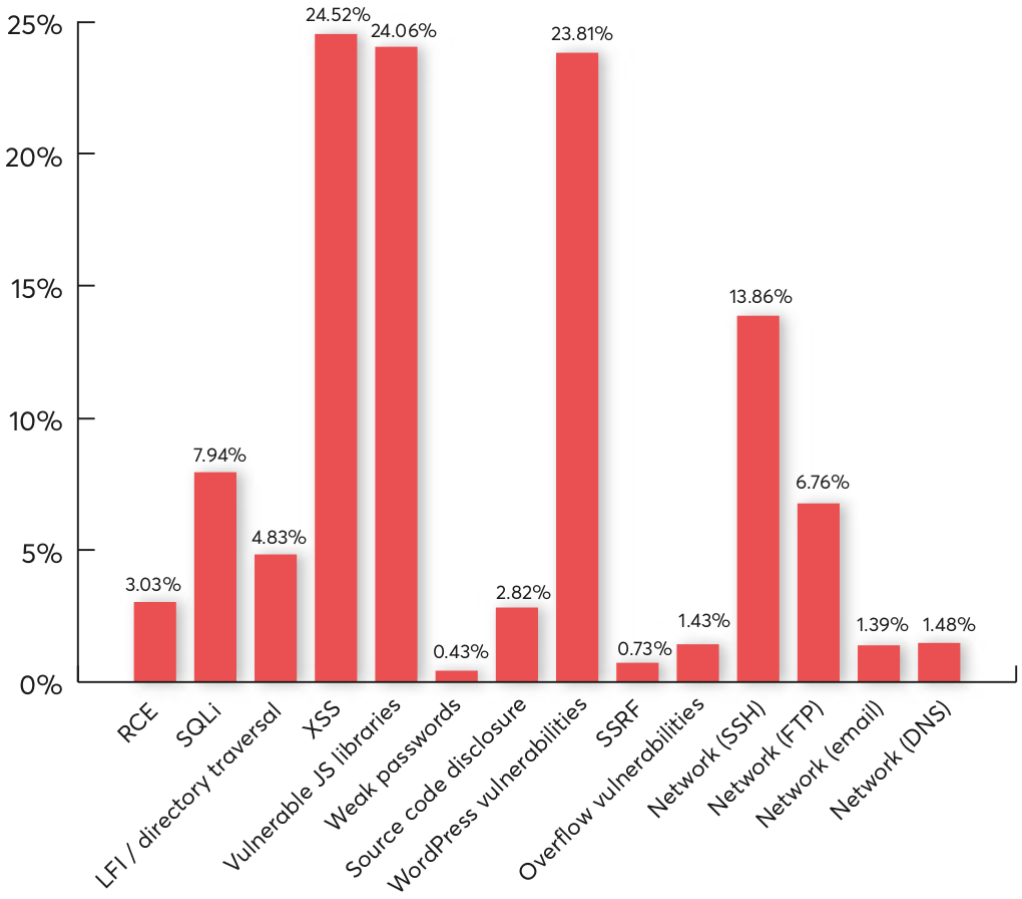


Obr. 2.4: Porovnanie zraniteľností webových aplikácií v priebehu rokov 2016 - 2019 rozdelených do kategórií s vysokou a strednou závažnosťou[1]

2.6 Bezpečnostné riziká webových aplikácií v roku 2020

Podľa správy zraniteľností za rok 2020 od spoločnosti Acunetix[2] sa miera zraniteľností webových aplikácii pomaličky znižuje ako aj ukazuje graf na obrázku 2.4. Avšak stále vyše 25% aplikácií dostupných na internete má aspoň jednu zo zraniteľností s vysokou závažnosťou a ako je vidieť na grafe na obrázku 2.5 najčastejšie je touto zraniteľnosťou práve XSS.

[2]Je to prvá spoločnosť, ktorá vyvinula plne-automatický skener na webovú zraniteľnosť



Obr. 2.5: Porovnanie zraniteľností webových aplikácií s vysokou závažnosťou za rok 2020[1]

2.7 Detekcia chýb a zraniteľností vo webových aplikáciách

Zraniteľnosti, ktoré môžu viesť ku kompromisom s citlivými informáciami sa začínajú pravidelne objavovať a cena výslednej škody narastá. Hlavnými dôvodmi pre túto skutočnosť sú čas, finančné obmedzenia, limitované programovacie zručnosti a skúsenosti alebo nedostatok povedomia o bezpečnosti na strane vývojárov [17]. Aj kvôli tomuto a tiež prevalencii kódu vo web aplikáciách, ktorý manipuluje s reťazcami sa v dnešnej dobe kladie väčší dôraz na analýzu reťazcov. Existuje veľa výskumov zameraných na detekciu chýb a zraniteľností vo webových aplikáciách, možno ich rozdeliť podľa miesta zasiahnutia na zraniteľnosti na strane klienta a na strane serveru [6].

2.7.1 Analýza kódu na serverovej strany

Riešenia zraniteľností na strane serveru majú tú výhodu, že sú schopné odhaliť a zachytiť veľký rozsah zraniteľností. Ďalšou výhodou je, že keď provider vyrieši bezpečnostnú chybu, toto riešenie sa okamžite rozšíri medzi všetkých jeho klientov. Takéto techniky riešenia problémov, môžu byť klasifikované na dva prístupy: dynamický a statický[17].

Pre statickú analýzu existuje mnoho nástrojov hlavne keď sa jedná o PHP alebo JavaScriptové webové aplikácie. Napríklad Pixy[17] používa rôzne techniky statickej analýzy na vytváranie grafov závislostí, ktoré reprezentujú tok údajov zo zdrojov do *sinkov* vo we-

bovej aplikácií. Takáto analýza skenuje webovú aplikáciu a teda jej zdrojový kód zatiaľ čo nástroje pre dynamickú analýzu sa snažia zachytiť útoky počas vykonávania programu[6].

Táto práca sa však zameriava hlavne na klientskú stranu aplikácie keďže sa zameriava na zraniteľnosť, ktorá sa nachádza práve na strane klienta.

2.7.2 Analýza kódu na klientskej strany

Rovnaké dva typy analýzy sa využívajú aj na strane klienta, t.j. statická a dynamická.

Statická analýza sa v JavaScriptovom programe využíva na zisťovanie vlastností programu, ako je napríklad zisťovaní zraniteľnosti. Prechádza vždy celý kód zhora dole, nezáleží pritom, či sa naozaj tento kód spustí počas behu aplikácie alebo nie. V praxi by však mal byť tento druh analýzy vždy doplnený (napríklad) dynamickou analýzou a to z jednoduchého dôvodu: môže dochádzať k strate presnosti alebo nedostatku škálovateľnosti. Vyplýva to práve zo samotnej podstaty JavaScriptu, keďže je to dynamický jazyk a dynamické vlastnosti tohto jazyka sú často využívané. Pri používaní hybridnej analýzy (čo znamená kombináciu statickej a dynamickej) sa výrazne zredukuje počet falošných poplachov a tiež sa zvýši presnosť. Môže byť vhodná a veľmi užitočná napríklad pri kontrol reťazcov, keďže v JavaScripte sa využíva mapovanie z reťazca na reťazec pri objektoch či poliach.

Dynamickou analýzou môžeme skontrolovať napríklad overenie zraniteľnosti validácie na strane klienta a to tak, že extrahujeme validačný kód, ktorý súvisí s určitým *sinkom* a potom využijú náhodný *fuzzing*[3] na otestovanie[6].

Príklad nástroja, ktorý využíva práve klientskú stranu na zabraňovanie útokom XSS je Noxes. Tento nástroj vznikol pre prostredie Microsoft Windows ako osobný firewall a ponúka ochranu v prípade podozrenia na XSS útoky, ktorými sa útočník snaží ukradnúť užívateľove osobné údaje a beží na ako služba na pozadí[18].

2.7.3 Útoky na klientskú stránku webových aplikácií

Komponenty klientovej stany webovej aplikácie zahrňujú hlavne:

1. **HTML kód**, ktorý definuje elementy webovej stránky a ich štruktúru;
2. **CSS kód**, ktorého úlohou je tvoriť štýl týmto elementom;
3. **JavaScript kód**, ktorý zabezpečuje funkcionalitu na klientskej strane.

Tieto časti webovej aplikácie môžu byť písané priamo programátorom alebo automaticky generované kódom na strane serveru. [24]

[3]Fuzzing alebo Fuzzy testovanie je technika testovania softvéru *Black Box*, ktorá používa nesprávne alebo čiastočne poškodené údaje na automatické vyhľadávanie chýb.

Kapitola 3

Cross–Site Scripting (XSS)

Podľa rebríčka zverejneného nadáciou OWASP je aktuálne tento typ útoku na siedmom mieste. Za posledných pár rokov sa posunul v tomto rebríčku na nižšie miesto, avšak nie je to z dôvodu lepšej ochrany pred týmto typom útoku ale skôr sa objavili nové relatívne nebezpečnejšie hrozby. Obrázok 3.1 ukazuje, že početnosť útokov sa za posledné roky zvyšuje a tiež to, že útoky typu XSS sú častejšie ako ostatné typy útokov[25]. Pri XSS zraniteľnosti sa jedná o webový útok, ktorý nainjektuje webovú stránku v prehliadači škodlivým kódom. K XSS útoku môže dôjsť ak[20]:

1. webová aplikácia prijíma dáta z nedôveryhodného zdroja, najčastejšie cez webovú požiadavku
2. dáta sú zahrnuté v dynamicky načítateľnom obsahu, ktorý je poslaný užívateľovi bez predošlej kontroly na škodlivý obsah

Keďže je skript poväčšine zahrnutý do obsahu odpovede webovej aplikácie, je vykonaný aj s rovnakým prístupom, ako keby bol skript legitímny. Z tohoto dôvodu môže byť povolený prístup k tokenom relácie, cookies či dokonca k dôverným informáciám, ku ktorým má prehliadač prístup na tomto webe, prípadne aj k prepisovaniu obsahu stránky HTML[20].

3.1 Skriptovacie jazyky v HTML[21]

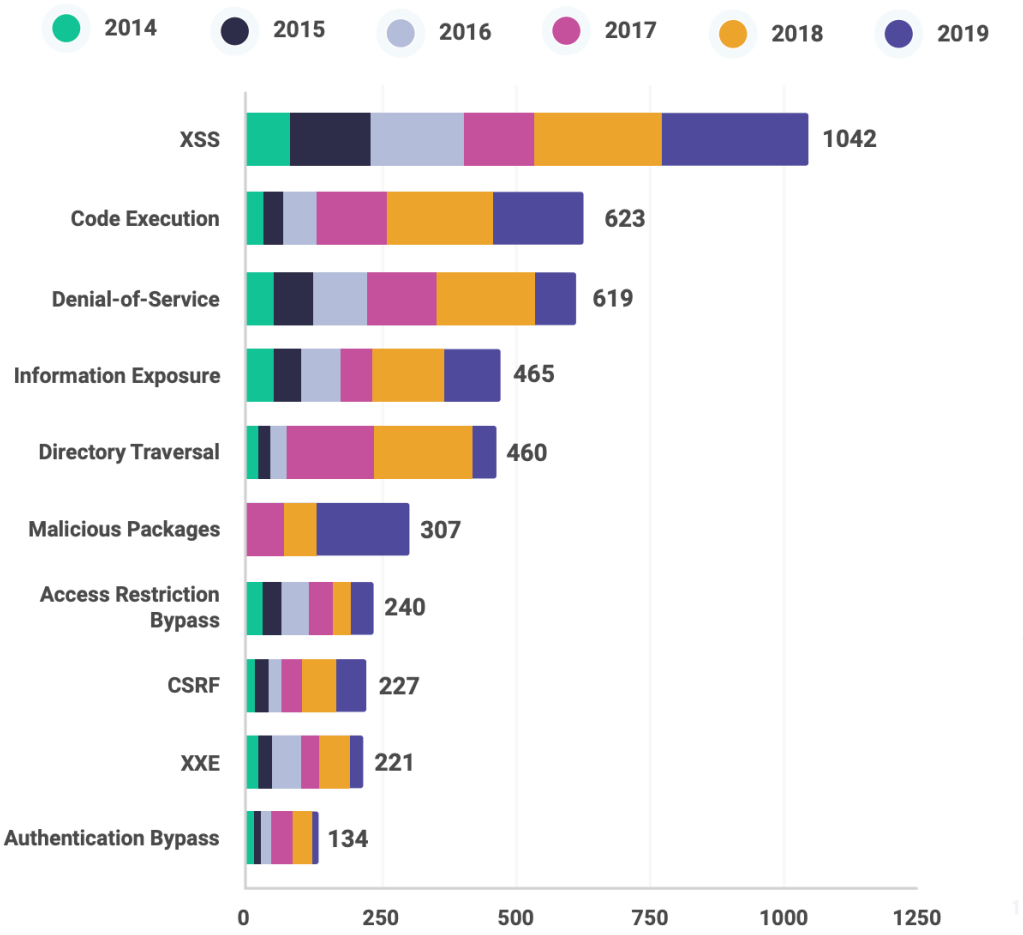
Zraniteľnosť typu XSS je založená na skriptovacích jazykoch na strane klienta. Práve vďaka skriptovacím jazykom, ktoré sú implementované do webových prehliadačoch, môžeme dynamicky načítavať obsah webových stránok v reálnom čase. Jeden z najznámejších a všeobecne najrozšírenejších skriptovacích jazykov je JavaScript, s ktorým prišla ako prvá spoločnosť Netscape. Postupne ho prevzali aj ďalšie webové prehliadače ako napríklad Internet Explorel, Mozila Firefox či Google Chorme.

Okrem JavaScriptu[1] je treba poznať aj jazyk HTML, ktorý tvorí obsah webových stránok.

3.2 Typy XSS zraniteľností

Útoky XSS je možné rozdeliť na dve kategórie. Prvá kategória obsahuje XSS útok typu *Reflected* - neperzistentný a *Stored* - perzistentný. Druhú kategóriu tvorí o dosť mladšia

[1] JS je multiplatformový, potrebujeme aby bol kompatibilny aj z ostatnými „nástrojmi“, ktoré vývojáry používaju pri tvorení webov



Obr. 3.1: Porovnanie početnosti zraniteľností za roky 2014 - 2019[22]

zraniteľnosť a to typu *DOM–based*. V tabuľke 3.1 môžeme vidieť stručný prehľad rozdielov medzi týmito kategóriami. Hlavný rozdiel medzi týmito kategóriami je v tom kde dochádza k útoku v aplikácií:

- pri útokoch *Reflected* a *Stored* XSS dochádza k injektovaniu aplikácie na strane serveru počas spracovávania požiadaviek, kde nedôveryhodný vstup je dynamicky pridávaný do HTML
- pri *DOM–based* je to pri behu aplikácie na strane klineta (t.j. v prehliadači)

3.2.1 Stored XSS

Perzistentný (Persistent) alebo útok XSS Typ I nastáva keď príjme aplikácia dáta z nedôveryhodného zdroja a zahrnie ich do neskorších HTTP odpovedí nezabezpečeným spôsobom. Príkladom takého útoku môže byť útočníkom napísaný komentár:

```
<script>/* Nejaký škodlivý kód od útočníka... */</script>
```

Takýto komentár by bol zakódovaný do URL adresy a každý ďalší užívateľ by ho dostal do odpovede, takto prijatý skript sa vykoná v rámci relácie s aplikáciou v prehliadači užívateľa.

Keď teda útočník môže spravovať skript, ktorý je uložený v užívateľovom prehliadači, má prístup ku všetkým akciám užívateľa (niektoré sú vypísané ako príklad v nasledujúcej podkapitole 3.2.2). Tento typ útoku je však zákernejší ako Typ II, lebo útočník nemusí hľadať externú cestu aby užívateľ zaslal HTTP požiadavku (napr. klikol na URL adresu), škodlivý skript sa uloží do používateľovej odpovede, ktorú dostane od aplikácie a tým je zaručené, že ho útok postihne. Útokom sa dá však vyhýbať napríklad používaním skeneru zraniteľností webu. Avšak je to náročnejšie ako pri Reflected XSS lebo musíme brať do úvahy všetky možné vstupné body do webovej aplikácie a tiež výstupy kde sa môžu škodlivé dáta nachádzať.

3.2.2 Reflected XSS

Tiež označovaný ako Typ II alebo perzistentný (Non-Persistent) nastáva keď aplikácia príjme dáta cez HTTP požiadavku a zahrnie tieto dáta nezabezpečenou cestou do okamžitej odpovede. Napríklad útočník môže kontrolovať útok ako tento:

```
https://insecure-website.com/status?message=
<script>/*nejaky+skodlivy+kod+od+utocnika...+*/</script>
```

V prípade že si obeť otvorí takúto URL v prehliadači umožňuje útočníkovi úplný prístup napríklad k jeho účtu. Príkladov možných hrozieb, ktoré z tohoto útoku plynú je viacej:

- Spravovať aplikáciu namiesto užívateľa
- Zobrazovať si informácie ktoré môže vidieť užívateľ
- Upravovať akékoľvek informácie, ktoré môže užívateľ upravovať
- Iniciovať interakcie s inými užívateľmi aplikácie, čo zahŕňa aj posielanie vírusov (bude sa to javiť ako by to zaslal užívateľ sám)

Existuje veľa spôsobov ako podhodiť užívateľovi škodlivú URL adresu (poslaním linku na mail alebo inou správou či ponechať odkaz na stránke kontrolovanej útočníkom) a tým spustiť útok tohoto typu. Môže sa potom jednať aj o cielený útok na konkrétneho užívateľa alebo náhodný výber užívateľov nejakej aplikácie. Veľká väčšina takýchto útokov sa našťastie dá ľahko odhaliť pomocou použitia skeneru na zraniteľnosť napríklad od Burp Suite. Dopad *Reflected XSS* útoku je vo všeobecnosti menší ako pri útoku *Stored XSS*.

3.2.3 DOM–based XSS

Tento typ zraniteľnosti XSS bol identifikovaný až v roku 2005 Amitom Kleinom a je tiež označovaný ako Typ 0. Princíp tejto zraniteľnosti je v tom, že klientská časť webovej aplikácie je injektovaná z hodnôt z parametrov a to pomocou JavaScript. JavaScript spracováva dáta z nedôveryhodného zdroja (napr. vyhľadávajúce okno v prehliadači) nezabezpečenou cestou a zapisuje ich späť do DOM.

Napríklad existuje aplikácia, ktorá používa vlastnosť `innerHTML` na dynamické upravenie svojho rozhrania. Ak sa do URL dostane `element script` od útočníka, ktorý aplikácia neošetrí, dochádza k jeho exekúcií ako náhle obeť tento odkaz navštívi. Tu prichádza rozdiel, ktorý robí túto zraniteľnosť samostatným typom a síce, že tento útok vôbec neprejde cez server. Z toho vyplýva, že bude oveľa náročnejšie detektovať alebo rovno odfiltrovať takýto útok. Kód, ktorým sa útočník snaží injektovať webovú aplikáciu pritom vôbec nemusí byť len v URL. Do objektového modelu sa môže dostať[12]:

- cez presmerovanie z útočníkovej stránky pomocou `document.referrer`, kde sa nachádza URL útočníkovej stránky alebo
- cez `window.name` cez HTML atribút `target` pri odkaze, ktorý obeť navštívi

Bližšie sa tejto zraniteľnosti venuje kapitola 4.

	XSS Typu I a II	**XSS Typu DOM–Based**
Príčina	Nezabezpečené vloženie vstupu od klienta do výstupnej HTML stránky.	Nezabezpečené odkazy a použitie DOM (na strane klienta), ktoré nie sú úplne kontrolované serverovou stránkou.
Zodpovedná osoba	Web developer (CGI[A])	Web developer (HTML)
Načítanie stránky	len dynamicky (CGI script)	Staticky (HTML) – nie nevyhnutne
Odhalenie zraniteľnosti	– manuálna alebo automatická injektáž chyby – kontrola kódu - potrebný je prístup k zdrojovému kódu stránky	– manuálna injektáž chyby – kontrola kódu (aj na diaľku)
Efektívna ochrana	Validácia dát na strane serveru nástroje na predchádzanie útokom (IPS[A], firewally aplikácií)	Validácia dát na strane klienta - v JavaScripte

Tabuľka 3.1: Rozdiely medzi DOM–based XSS a XSS Typu I a II

3.2.4 Server XSS vs. Klient XSS

Aj keď sa dlhé roky pristupovalo k tomuto rozdeleniu ako k trom rozličným typom, v realite nemusia byť striktne oddelené. Napríklad môžeme mať perzistentný Non-DOM–based XSS tak isto, ako aj perzistentný DOM–based XSS a to isté platí aj pre neperzistentný typ (viď. 3.2). V roku 2012 sa vo vedeckej komunite navrhlo a začalo používať nové rozdelenie týchto typov:

- Server XSS – Nedôveryhodné dáta sa dostávajú do vygenerovaného HTML/JavaScriptu a najlepšia ochrana je kontextové kódovanie.
- Klient XSS – Nedôveryhodné dáta sa pridávajú do DOM cez nezabezpečené volanie JavaScriptu a najlepšou ochranou je používannie bezpečného JavaScriptového API.

XSS	Server	Klient
Perzistentý	Perzistentý Server XSS	Perzistentý Klient XSS
		Perzistentý DOM–based Klient XSS
Neperzistentný	Neperzistentný Server XSS	Neperzistentný Klient XSS
		Neperzistentný DOM–based Klient XSS

Tabuľka 3.2: Miesto kde sa objavia nedôveryhodné dáta [32]

3.3 Prevencia proti XSS útokom[15]

Content Security Policy (CSP^A) je pridaná vrstva zabezpečenia, ktorá pomáha odhaliť a zmierniť niektoré typy útokov, vrátane Cross Site Scripting (XSS) a injekčných útokov.Je to halvička HTTP odpovedi (HTTP response header), ktorú používajú prehliadače s cieľom vylepšiť bezpečnosť webovej aplikácie (alebo dokumentu). Princíp spočíva v tom, že tieto hlavičky umožňujú obmedziť čo môže prehliadač načítať (JavaScript, CSS,...). Tento spôsob ochrany pomáha primárne pri redukovaní XSS útokov ale môže poskytovať ochranu napríklad aj proti útokom ako je Click Jacking[2] alebo iným útokom, ktoré injektujú kód. Podporu má u všetkým hlavných prehliadačov ako napríklad Chrome, FireFox, Safari,... Najnovšia verzia vyšla v roku 2021 a je označovaná ako *Level 3*. Tento produkt je pod správou *W3C*^{A}. Aj keď sa primárne používa ako hlavička HTTP odpovede, je možné ju použiť aj prostredníctvom meta tagu.

3.3.1 Meta tag

Metadáta HTML dokumetu sú definované pomocou `<meta>` tagu. Metadáta sú dáta, ktoré informujú o dátach stránky. Tento tag sa vždy nachádza na elemente `<head>` a používajú sa na upresnenie používanej znakovej sady, popisu stránky, kľúčových slóv, autora dokumentu či nastavenia časti stránky, ktorá sa zobrazuje užívateľovi. Metadáta sa nezobrazujú pri vykreslovaní stránky ale keďže sú zahrnuté v zdrojovom kóde stránky, dajú sa parsovať. Používajú ich prehliadače na vyhľadávanie podľa kľúčovych slóv, či iné webové služby [3].

Jedným zo spôsobov ako pridať CSP do hlavičky HTTP response, môže byť práve použitím meta tagu (3.1).

```
1 <meta http-equiv="Content-Security-Policy" content="default-src 'self'">
```

Príklad 3.1: Príklad vloženia CSP do meta tagu [16]

[2]Tiež známe ako „UI redress attack“ nastáva keď útočník prinúti užívateľa kliknúť na tlačítko/link, tým že používa priehľadné alebo nepriehľadné vrstvy.

Kapitola 4

Analýza DOM–based XSS

Útok DOM–based je jedinečná forma útoku na strane klienta. Document Object Model (DOM) je špecifikácia W3C, ktorá definuje objektový model na reprezentáciu XML a HTML štruktúr. Jazyk XML používa dva hlavné parsery:

- **SAX** je výrazne rýchlejší a menej pamäťovo náročný ale nie je veľmi intuitívny, pretože mechanizmus analyzátoru je jednosmerný čiže nie je ľahké sa vrátiť späť po uzloch dokumentu.
- **DOM** načíta celý dokument ako objektovú štruktúru vďaka tomu sa môže pohybovať po jednotlivých uzloch dokumentu a jednoducho, za chodu, meniť hodnoty, premenné, atď. Prehliadače pracujú práve s týmto typom parseru. Bližšie je tento model popísaný v podkapitole 2.3.

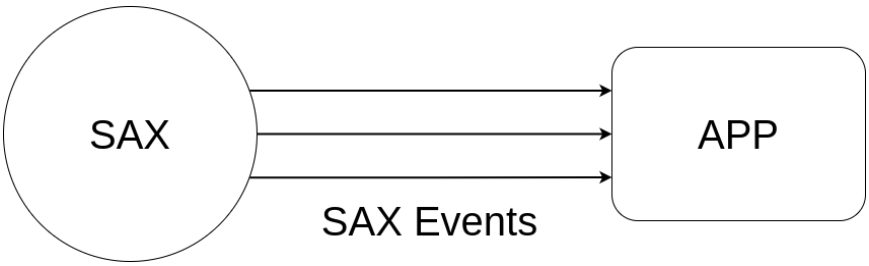
Pri DOM–based XSS nie je dôvodom pre zraniteľnosť časť skriptu na servery, ale nesprávne zaobchádzanie s dátami, ktoré pochádzajú od užívateľa na klientskej strane JavaScriptu.

4.1 Princíp

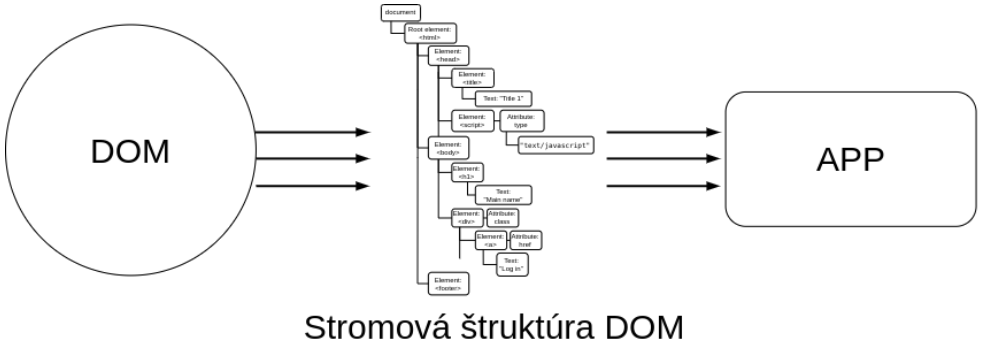
Na rozdiel od XSS typu I alebo II sa *DOM–based* spolieha výlučne na JavaScript na klientovej strane a na jeho nezabezpečené používanie dynamicky načítateľných dát do DOM štruktúry. Nevyhnutným predpokladom pre túto zraniteľnosť je mať HTML stránku, ktorá používa dáta načítané z `document.location` alebo `document.URL` alebo `document.referrer` (alebo podobného objektu ku ktorému má útočník prístup) nezabezpečeným spôsobom.

```
1  <HTML>
2      <TITLE>Welcome!</TITLE>
3      Hi
4      <SCRIPT>
5          var pos=document.URL.indexOf("name=")+5;
6          document.write(document.URL.substring(pos,document.URL.length));
7      </SCRIPT>
8      <BR>
9      Welcome to our system
10 </HTML>
```

Príklad 4.1: Príklad zraniteľnosti DOM–based ukázaný Amitom Kleinom v jeho článku "Dom Based Cross Site Scripting or XSS of the Third Kind"[19]



Obr. 4.1: Typológia SAX



Obr. 4.2: Typológia DOM

Predpokladané správanie tejto webovej aplikácie je že užívateľ zadá svoje meno a objaví sa uvítacia stránka. Ale čo v prípade, že útočník zadá do premennej kód napríklad:

```
<script>alert(...)</script>
```

Vznikne požiadavka s URL:

`http://www.vulnerable.site/welcome.html?name=<script>alert(...)</script>`,

ktorá keď sa začne parsovať v užívateľovom prehliadači do DOM, do objektu `document` do položky `URL` bude skopírovaná URL aktuálnej stránky. Keď sa potom parser dostane ku kódu JavaScrite, vykoná ho a tým zmení pôvodnú HTML stránku.

4.2 Kontext vykonávania (angl. execution context)[26]

Kontext vykreslenia (angl. *rendering context*) HTML obsahu (a iných obsahov stránky ako napríklad CSS alebo JavaScript,...) je spojený s analýzou HTML *tagov* a ich atribútov. Parser hovorí, ako majú byť dáta prezentované a kde na stránke sa majú nachádzať. Ďalej ich delí na štandardné kontexty HTML, atribúty HTML, URL a CSS.

Každý syntaktický analyzátor pre JavaScrit má odlišný prístup k spôsobu, akým môže byť vykonávaný skriptovací kód t.j. sémantická analýza. Táto skutočnosť sťažuje vytváranie konzistentných pravidiel, ktoré sú potrebné na zmierňovanie zraniteľností v rôznych kontextoch. Ďalší problém pri kompilácií môže spôsobovať kódovanie hodnôt. Rôzne významy a zaobchádzanie s takýmito hodnotami v rámci pod-kontextov (atribúty HTML, URL, CSS...) a v rámci vyhľadávania kontextov[1]. V príklade 4.2 je vidieť použitie HTML pod-kontextu v kontexte JavaScriptu.

[1]HTML, HTML atribúty, URL a CSS sú tu popisované ako pod-kontexty lebo každý môže byť použitý a aj nastavený v rámci JavaScritového kontextu

```
<script>
    var x = '<\%= taintedVar \%>';
    var d = document.createElement('div');
    d.innerHTML = x;
    document.body.appendChild(d);
</script>
```

Príklad 4.2: Príklad kódu JavaScriptu s použitím HTML pod-kontextu

4.2.1 Pod-kontexty kontextu vykonávania

Podľa OWASP[26] existuje niekoľko pod-kontextov kontextu vykonávania ktoré ovplyvňujú zraniteľnosť stránky na DOM–based XSS .

HTML pod-kontext

Existujú isté metódy a atribúty do ktorých sa dá priamo vkladať JavaScript a môžu byť použité na priame vykonanie HTML obsahu:

- ATRIBÚTY: element.innerHTML, element.outerHTML, ...
- METÓDY: document.write, document.writeln, ...

Tomuto sa dá predchádzať zakódovaním nedôveryhodného vstupu HTML ako aj JavaScriptu pred vložením do stránky, napríklad:[2]

```
element.outerHTML="<%=Encoder.encodeForJS(Encoder.encodeForHTML(...))%>";
```

Pod-kontext HTML atribútov

Pravidlo na kódovanie HTML atribútov je nevyhnutné a veľmi dôležité pre zmiernenie útokov aj keď môže byť odlišné od štandardných kódovacích pravidiel. Tieto útoky môžu byť zamerané buď na ukončenie HTML atribútov alebo vloženie dodatočného atribútu. V kontexte vykonávania DOM, potrebujeme aby JavaScript zakódoval HTML atribúty, ktoré nevykonávajú kód (t.j. iné ako event handler, CSS a URL atribúty). HTML atribút, ktorý nevykonáva kód nastavuje hodnotu priamo v rámci atribútu objektu v HTML elemente čo znamená, že tu nemôže dôjsť k injektáži. Všeobecné pravidlo je že nedôveryhodné dáta v HTML atribútoch sa zakódujú. Tento krok môže byť výhodný keď sa dáta dostávajú na výstup pri vykresľovaní ale pri používaní kódovania HTML atribútov v kontexte vykonávania to spôsobí prerušenie zobrazovania dát v aplikácií. Príklad 4.3 ukazuje riešenie, ktoré je síce bezpečné ale je prerušené, narozdiel od príkladu 4.4.

```
var x = document.createElement("input");
x.setAttribute("name", "company_name");
x.setAttribute("value",'<%=Encoder.encodeForJS(Encoder.encodeForHTMLAttr(companyName))%>');
var form1 = document.forms[0];
form1.appendChild(x);
```

Príklad 4.3: Bezpečné ale prerušené riešenie

[2]Encoder.encodeFor... sú len vymyslené kodéry

```
var x = document.createElement("input");
x.setAttribute("name", "company_name");
x.setAttribute("value", '<%=Encoder.encodeForJS(companyName)%>');
var form1 = document.forms[0];
form1.appendChild(x);
```

Príklad 4.4: Bezpečné a správne funkčné riešenie

`Event Handler` atribút a pod-kontext JavaScript kódu

Vkladanie dynamických dát do kódu JavaScriptu je obzvlášť nebezpečné, pretože JavaScriptové kódovanie dát má rozdielnu sémantiku ako napríklad jazyk HTML. V mnohých prípadoch kódovanie JavaScriptu nedokáže zastaviť útoky v kontexte vykonania, preto je najlepšie sa úplne vyhnúť takémuto vkladaniu dát.

```
var x = document.createElement("a");
x.href="#";
x.setAttribute("onclick", "\u0061\u006c\u0065\u0072\u0074\u0028\u0032\u0032\u0029");
var y = document.createTextNode("Click To Test");
x.appendChild(y);
document.body.appendChild(x);
```

Príklad 4.5: Metóda `setAttribute`

Kódovanie JavaScriptu nezabraňuje útoku DOM XSS lebo metódy, ktoré používa berú kód ako premennú typu *string* (napr. metóda setAttribute, setTimeout, setInterval, new Function,...). V príklade 4.5 je implicitne hodnota *value_string* zapísaná do DOM atribútu dátového typu *name_string*. Atribút typu *name_string* je JavaScriptový event handler to znamená, že atribút *value_string* sa implicitne konvertuje do kódu JavaScriptu a vyhodnocuje.

Na rozdiel od toho pri kódovaní JavaScriptu, ktoré sa používa pre atribút HTML *tagu* event handler je efektívne proti XSS. Riešenie problému a teda aby kódovanie JavaScriptu bolo účinné voči DOM XSS je nastaviť *event handler* priamo (viď. 4.6). Avšak ani tento spôsob je väčšinou spojený s nebezpečenstvom útoku.

```
var s = "\u0065\u0076\u0061\u006c";
var t = "\u0061\u006c\u0065\u0072\u0074\u0028\u0031\u0031\u0029";
window[s](t);
```

Príklad 4.6: Kódovanie JavaScript, ktoré je akceptované ako validne spustiteľný kód

Takéto môžnosti dáva JavaScriptu fakt, že tento programovací jazyk je založený na medzinárodnom štandarde *ECMAScript* a preto kódovanie JavaScriptu umožňuje podporu medzinárodných znakov v programovacích konštrukciách a premenných k alternatívnym zastúpeniam reťazcov (string escapes). Čo sa týka kódovania HTML je to naopak. Elementy HTML *tagov* sú dobre definované a nepodporujú alternatívne zobrazenia toho istého *tagu* (napríklad kódovanie HTML nemožno použiť na to, aby vývojár mal alternatívnu reprezentáciu `<a>` *tagu*).

Nedôveryhodné dát v atribútoch pod-kontextu CSS

Ako popisuje príklad 4.7 spustenie JavaScriptu cez CSS kontext vyžaduje v CSS metóde `url()` použitie `javascriptu:attackCode()` musí byť však isté, že dáta, ktoré vchádzajú do tejto metódy sú kódované. Kedysi sa používala aj iná CSS metóda a to `expression()` avšak už nie je ďalej podporovaná a to aj z dôvodu bezpečnosti.

```
1 document.body.style.backgroundImage = "url(<%=Encoder.encodeForJS(Encoder.encodeForURL(
      companyName))%>)";
```

Príklad 4.7: CSS metóda `url()`

Logika, ktorá je za parsovaním URL pre kontext vykresľovania a vykonávania sa zdá ako rovnaká ale nie je to úplne tak. V kódovacích pravidlách pre URL atribúty pre vykonávací kontext (*DOM kontext*) je istý rozdiel:

```
1 var x = document.createElement("a");
2 x.setAttribute("href", '<%=Encoder.encodeForJS(Encoder.encodeForURL(userRelativePath))%>');
3 var y = document.createTextElement("Click Me To Test");
4 x.appendChild(y);
5 document.body.appendChild(x);
```

Príklad 4.8: Vykonávací - DOM kontext

Pri použití plne kvalifikovaných URL adries sa tým odkazy rozbijú lebo dvojbodka v identifikátore protokolu (`http:` alebo `javascript:`) bude zakódovaná pomocou URL adresy a zabráni tak tomu aby protokoly HTTP a JavaScript boli vyvolané.

Používaním bezpečných JavaScriptových funkcií a vlastností v DOM

Najzákladnejšia bezpečná cesta ako naplniť DOM s nedôveryhodnými dátami je používať vlastnosti bezpečného priradenia `textContent`.

```
1 <script>
2   element.textContent = untrustedData;
3 </script>
```

Príklad 4.9: Príklad bezpečného použitia - nevykoná kód

Napravenie DOM–based XSS zraniteľností

Všetko závisí od použitia správnej výstupnej metódy - tzv. *sink*. Napríklad ak potrebuje programátor vstup od užívateľa na zapísanie do `div tag` tak použije `innerText` alebo `textContent` namiesto `innerHtml`. Naopak najhoršia možnosť akú môže programátor zvoliť je použiť vstup kontrolovaný užívateľom bez toho aby tento vstup nejakým spôsobom ošetril.

V príklade ktorý je uvedený vyššie (príklad 4.2) teda stačí použiť `element.textContent`:

```
1 <b>Current URL:</b> <span id="contentholder"></span>
2 ...
3 <script>
4   document.getElementById("contentholder").textContent = document.baseURI;
5 </script>
```

Príklad 4.10: Príklad bezpečného použitia

Takéto riešenie síce robí to isté ale v tomto prípade sa redukuje DOM–based XSS zraniteľnosť.

4.3 Hrozby plynúce z útoku DOM–based

Podobne ako pri ostatných typoch XSS útokov, je cieľom ukradnutie dôverných informácií, celý užívateľský účet, cookies užívateľa alebo iné citlivé informácie uchovávané prehliadačom v súvislosti s danou webovou stránkou.

4.4 Navrhovaná efektívna ochrana pred DOM–based XSS

1. Asi najjednoduchší spôsob je úplne sa vyhnú používaniu metód na prepisovanie, presmerovanie alebo iných citlivých operácii na strane klienta s použitím dát od klienta.

2. Analyzovanie kódu v JavaScripte na klientovej strane (tzn. kontrolovať objekty v DOM, ktoré môžu byť ovplyvnené užívateľom - útočníkom):

   - `document.URL`
   - `document.URLUnencoded`
   - `document.referrer`
   - `document.location` a jeho vlastnosti
   - `window.location` a jeho vlastnosti
   - ...

   Zvláštnu pozornosť vyžaduje situácia, kde je DOM modifikovaný:

   - písaním rovno do HTML napr. `document.write(...)`
   - priamym upravovaním DOM napr. `document.attachEvent(...)`
   - nahradzovaním URL dokumentu napr. `document.URL=...`
   - otváraním alebo modifikáciou okien napr. `document.open(...)` alebo `window.open(...)`
   - skriptom, ktorý sa hneď vykoná napr. `window.execScript(...)`

   Jednou z možných efektívnych ciest na ochranu, pre príklad 4.1, Amit Klein[19] uvádza:

```
1 <SCRIPT>
2     var pos=document.URL.indexOf("name=")+5;
3     var name=document.URL.substring(pos,document.URL.length);
4     if (name.match(/^[a-zA-Z0-9]$/)) {
5        document.write(name);
6     } else {
7         window.alert("Security error");
8     }
9 </SCRIPT>
```

Príklad 4.11: Príklad bezpečného použitia

3. Zaviesť IPS politiku, pri ktorej je napríklad pre stránku `welcome.html` len jediný argument nazvaný `name` a jeho hodnota sa kontroluje. V prípade akéhokoľvek neregulárneho vstupu (napr. prázdny reťazec alebo naopak veľmi dlhý), tento výsledok nie je prenesený na pôvodnú stránku. Avšak v niektorých prípadoch ani toto nezaručuje úplné prekazenie útoku.

Kapitola 5

Návrh riešenia

Cieľom tejto kapitoly je navrhnúť nástroj na analýzu JavaScriptu pre detekciu DOM–based XSS zraniteľností vo webových aplikáciách. Bežná komunikácia prebieha z webovej aplikácie na sieťový server. Komunikáciu možno kontrolovať pomocou proxy serveru, ktorý je na ceste medzi webovým prehliadačom a webovým serverom. Implementovaný nástroj, s názvom *DOM–basedXSS catcher*, ktorého účelom je evaluácia webových aplikácií na zraniteľností DOM–based XSS, je navrhnutý ako rozšírenie pre proxy server Burp Suite (obr. 5.1).

V komunikácií, ktorá prechádza proxy serverom vieme analýzou JavaScriptu identifikovať nedôveryhodné vstupy (angl. sources) a výstupy (angl. sinks). Avšak pre detekciu a následnú exploatáciu DOM–based XSS zraniteľností je potrebná interpretovaná odpoveď prehliadačom. Z toho dôvodu si nástroj odchytí odpoveď, ktorú dostáva proxy od serveru. Výsledkom analýzy odpovede sú nájdené nedôveryhodné vstupy a výstupy. Následne sa nástroj snaží do nájdených vstupov injektovať sadu *payloadov*, ktoré by mohli byť potenciálne interpretované. Pomocou *headless* prehliadača nástroj skontroluje, či sa niektorý z injektovaných *payloadov* vykonal a či identifikovaná chyba v odpovedi predstavuje hrozbu pre užívateľa. V takomto prípade upozorní užívateľa na možné nebezpečenstvo. Výstupom je textový súbor so sumarizáciou výsledkov pre danú URL.

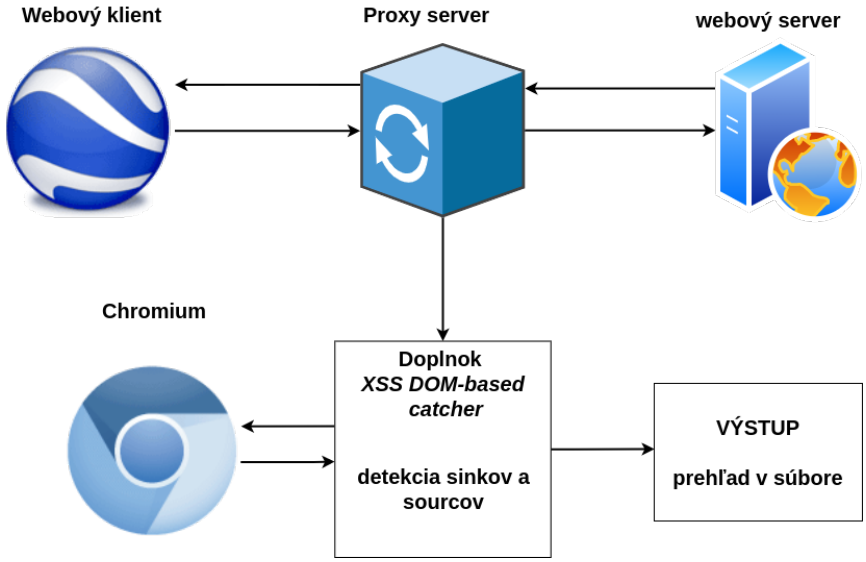
5.1 Existujúce riešenia

Existujú rôzne riešenia ako odhaliť Cross-side scripting zraniteľnosť vo webových aplikáciách. Niektoré z nich sa zameriavajú priamo na DOM–based, iné riešia XSS zraniteľnosť ako celok. Dostupné nástroje sú implementované pre rôzne webové rozhrania a v porovnaní s navrhovaným riešením pre odlišný spôsob použitia.

DOMinátor

Tento nástroj bol vyvinutý v roku 2011 Stefanom Di Paolom [1] pre internetový prehliadač *Firefox*. Používal sa na penetračné testovanie a má na starosť identifikovať zraniteľnosť reflected DOM–based XSS. Dynamicky vyhodnocuje dáta počas behu programu a pomáha identifikovať problémy na strane klienta v relatívne krátkom čase.

[1] https://blog.mindedsecurity.com/2011/05/dominator-project.html



Obr. 5.1: Schéma návrhu riešenia

DOMGoat

DOMGoat je open-source aplikácia, ktorá bola primárne vyvinutá na pomoc testerom pre pochopenie rozličných problémov bezpečnosti na strane kódu klienta, ktorá sa prejavuje v DOM. Tiež sa môže použiť na získanie informácií o problémoch, ktoré môžu nastať. Autorom tohto nástroja je Lavakumar Kuppan.

NoXSS[2]

Skener XSS zraniteľností, ktorý detekuje reflected a DOM–based XSS. Výhodou tohto skeneru je rýchlosť, ktorá je zabezpečená práve tým, že využíva len 8 payloadov pre testovanie. Takýmto spôsobom dokáže overiť aj milión URL adries (rýchlejšie ako fuzzing). Vyvinutý bol pre internetový prehliadač Chrome, používa Python verziu 2.7 a je naprogramovaný pre operačný systém Linux.

XSSvalidator

Bol naimplementovaný Johnom Poulinakom ako rozšírenie pre Burp Suite od spoločnosti PortSwigger pre automatizáciu a validáciu XSS zraniteľnosti[3]. Je to jediné dostupné rozšírenie v burpe, ktoré rieši XSS zraniteľnosť. Funguje na princípe posielania odpovede na lokálne bežiaci server na detekciu XSS. Server detekcie XSS používa technológiu Phantom.js a/alebo Slimer.js.

DOM Snitch

Pasívny prieskumný nástroj DOM Snitch vo vnútri DOM je experimentálny nástroj vyvinutý ako rozšírenie pre prehliadač Chrome. V reálnom čase môžu developer alebo tester sledovať ako sa DOM modifikuje bez toho aby museli debugovať JavaScriptový kód alebo

[2]https://github.com/lwzSoviet/NoXss
[3]https://portswigger.net/bappstore/98275a25394a417c9480f58740c1d981

pozastaviť vykonávanie ich aplikácie. Testerom pomáha identifikovať bežne používané zlé postupy pri vytváraní kódu na strane klienta a bezpečnostným testerom pomáha lepšie porozumieť transformáciám, ku ktorým v rámci DOM dochádza. Dokáže relatívne ľahko exportovať a zdieľať zachytené úpravy DOM.

5.2 *Headless* prehliadače

Kvôli rýchlosti a lepšej možnosti ho programovo ovládať, používam v navrhovanom riešenie headless prehliadač. Pod pojmom *headless* prehliadač rozumieme internetový prehliadač, ktorý nepoužíva grafické rozhranie. To zabezpečuje dostatočnú rýchlosť pre vykonávanie potrebných operácií. Prístup k ovládaniu *headless* prehliadačov je primárne cez príkazový riadok alebo prostredníctvom sieťovej komunikácie.

Použitie

Využíva sa predovšetkým pri testovaní webových aplikácií cez automatizované testy (klikanie myšou, potvrdzovanie formulárov,...). Tiež na získavanie dát z webu, tesetovanie layoutu či na rýchle otestovanie výkonu webu.

Výhody

Jednou z najväčších východ je rýchlosť, napríklad pri testovaní sa ignorujú všetky vykresľovanie operácie. Ďalšia dôležitá výhoda je rýchle odhalenie problému ak nejaká aktivita na webe neprebieha plynule a tiež využíva menej pamäte.

Nevýhody a obmedzenia

Pri využívaní *headless* prostredia si developer musí dať pozor, aby neopravoval chyby, ktoré sa neprejavia pri spúšťaní v normálnom užívateľskom prostredí. Keďže sa stránky bez UI môžu načítavať veľmi rýchlo, môže to spôsobiť sťaženie ladenia nekonzistentných zlyhaní pri lokalizácii prvkov.

Chromium

Chromium[4] (obr. 5.2a) patrí pod vývojára Google Chrome a je to open-source webový prehliadač. V aplikácií Chrome je toto *headless* prostredie dostupné od verzie 59. Často sa využíva pre vytváranie PDF súborov, zhromažďovania informácií o webe alebo na prehľad DOM modelu.

Firefox

Na rozdiel od Chromia je Firefox (obr. 5.2b) prehliadač, ktorý má možnosť spustiť mode *headless*. Najčastejšie sa používa so Seleniom a na automatizované testy, čo robí testovací proces efektívnejším. Tento mode je v prehliadači podporovaný od verzie 56.

[4]https://www.chromium.org/

(a) Chromium (b) Firefox (c) PhantomJS

Obr. 5.2: Logá headless prehliadačov

Phantom JS

Síce sa už aktívne nepoužíva no Phantom JS[5] (obr. 5.2c) bol veľmi populárny open-source softvér, ktorý spravovali ho vývojári. Bol to multiplatformový WebKit skriptovateľný pomocou JavaScriptu.

5.3 Proxy servery

Pre efektívne odchytenie a následnú analýzu komunikácie využívam proxy server, ktorý je umiestnený medzi klientom a web serverom. Bežná hlavná úloha, ktorú má proxy server plniť je ochrana užívateľovho súkromia. Tvorí akúsi bránu alebo prostredníka medzi klientskou stranou a serverom a funguje ako istý webový filter prípadne firewall. Zabezpečuje pre užívateľa rôzne stupne ochrany, súkromia a funkcionality. Prechádza cez neho všetky požiadavky od klienta ako aj odpovede serveru (celý tok internetového prenosu). Pre rýchlejšie spracovanie a odosielanie budúcich požiadaviek si proxy server ukladá dáta do vyrovnávacej pamäti (angl. *cache memory*)[28].

5.3.1 Nevýhody proxy serverov

Napriek tomu, že je výhodou keď proxy zakrýva IP adresu užívateľa pred zvyškom internetu môže používanie proxy serverov mať aj isté nevýhody. Napríklad niektorí poskytovatelia proxy serverov monitorujú a ukladajú si históriu vyhľadávania alebo aj niektoré osobné údaje užívateľov. Užívateľa by si tiež mali dať pozor na zabezpečenie spojenia, ktoré proxy používa. Častokrát je toto spojenie nezašifrované lebo proxy server používa len SSL certifikát na šifrovanie prenášaných dát, čo v dnešnom svete častých útokov (napr. SSL stripping) nemusí byť plnohodnotná ochrana užívateľa a jeho dát. Aj fakt, že používaná webová stránka ukazuje užívateľovi SSL šifrovanie, nemusí nutne znamenať, že užívateľove dáta sú zašifrované aj vo chvíli, keď odchádzajú z proxy serveru na cieľový server[7].

5.3.2 Typy

- Transparentný = identifikujú sa ako proxy ale neskýva IP adresu užívateľa. Firmy, verejné knižnice a školy často používajú na filtrovanie obsahu transparentné servery proxy: ich nastavenie je jednoduché na strane klienta aj servera.

[5] https://phantomjs.org/

Obr. 5.3: Logo Burp Suite

- Anonymný = identifikuje sa ako proxy ale skryje IP užívateľa, bráni cieleným reklamám
- Skresľujúci = server poskytuje falošnú IP adresu užívateľovi pokým sa sam identifikuje ako proxy; sluzi na podobne ako anonymný server ale pomocou falosnej IP sa moze uzivatel prihlasit z inej lokacie a tym obist obmedzenia obsahu
- Proxy s vysokou anonymitou = periodicky menia IP adresy, ktorymi sa prezentuju na web serveroch a to stazuje treckovanie tokov a priradenie k zariadeniam. napr TOR Network, je najbezpecnejsia cesta ako komunikovat s internetom

5.3.3 Burp Suite[2]

Nástroj Burp Suite (obr. 5.3) je jeden z najviac používaných nástrojov pre penetračné testovanie, či hľadanie zraniteľností na webe. Využíva sa na kontrolu bezpečnosti webových aplikácií pomocou buď implicitného prehliadača, ktorý otvorí užívateľ cez *burp app* alebo pomocou použitia externého prehliadača do ktorého je nainštalovaný CA certifikát burpu. Okrem klasického odpočúvania toku umožňuje užívateľovi aj prešetrovať či modifikovať prechádzajúce toky dát oboma smermi. To znamená, že tento nástroj ukladá históriu komunikácie a okrem toho je možné vďaka tlačidlu *Intercept is on*, sledovať túto komunikáciu aj *real time*. Dôležitou súčasťou nástroja je aj *BApp Store*, ktorý ponúka škálu rozšírení čo naprogramovali samotný užívatelia. Prípadne sa dá neimplementovať vlastné rozšírenie s funkcionalitou, ktorá je aktuálne potrebná pre daného užívateľa.

Kapitola 6

Implementácia a etické testovanie

Obsahom tejto kapitoly je bližší popis implementácie navrhnutého rozšírenia s názvom *DOM–based XSS catcher*. Ďalej sú popísané nástroje a technológie použité pri vývoji. Programová časť riešenia je rozdelená na dva logické celky. V prvom celku sa detekujú *sources* a *sinks* a v druhom sa tieto miesta v kóde testujú na DOM–based XSS zraniteľnosť.

Rozšírenie *DOM–based XSS catcher* bolo vyvíjané na operačnom systéme Ubuntu 20.04.2. ale keďže nástroj Burp Suite, do ktorého sa toto rozšírenie nahráva je multiplatformový, tak aj toto rozšírenie je multiplatformové.

6.1 Použité nástroje pri implementácií rozšírenia *DOM–based XSS catcher*

Projekt je písaný v jazyku Python. Vzhľadom na to, že Burp Suite používa primárne jazyk Java, použitý interpreter je Jython verzie 2.7. Implementácia bola programovaná vo vývojovom prostredí PyCharm a testovanie funkčnosti nástroja prebehlo priamo cez Burp Suite.

6.2 Štruktúra a využité triedy

Diagram tried, ktorý je zobrazený na obrázku 6.1, opisuje štruktúru a tried využívaných v nástroji *DOM–based XSS catcher*.

Trieda `response_catcher.BurpExtender` obsahuje dve metódy, kde každá z nich využíva iné triedy či metódy tried z knižnice *burp*[1]. Prvá metóda `registerExtenderCallbacks`, ktorá zabezpečuje registráciu rozšírenia využíva triedu `burp.burp.IBurpExtender` a jej jedinú metódu `registerExtenderCallbacks` na načítanie objektu. Toto rozhranie je nutné pre každé rozšírenie vyvíjané pre Burp. A tak isto je nevyhnutné, aby bolo deklarované verejne a musí obsahovať predvolený konštruktor.

Druhá metóda `processHttpMessage` využíva inštanciu triedy `burp.burp.IHttpListener`. Príznak, ktorý má na starosti rozlišovanie či sa jedná o požiadavku alebo odpoveď je definovaný v objekte vytvorenom predchádzajúcou metódou. Obsah HTTP odpovedi prípadne požiadavku je načítateľný a uložený v `IHttpRequestResponse`.

Trieda `burp.burp.IExtentionHelpers` má na vstupe objekt, vytvorený na začiatku rozšírenia. Obsahuje mnohé pomocné metódy ako napríklad `bytesToString` na konvertova-

[1] https://portswigger.net/

nie bytového poľa do reťazca znakov. Rozšírenia využívajú volanie `IBurpExtenderCallbacks .getHelpers` pre získanie inštancie tohto rozhrania.

Trieda `object` je základná trieda hierarchie tried v knižnici *burp*. Pri zavolaní, neprijíma argumenty a vracia novú inštanciu bez vlastností, ktorá neobsahuje atribúty a žiadne nemôže prijímať. Na tejto triede sú založené všetky ostatné triedy a teda aj metódy využívané v rozšíreniach a nástrojoch.

6.3 Popis *DOM–based XSS catcher* a použité knižnice

Pre inicializáciu rozšírenia je využitá knižnica burp. Vytvorí sa inštancia triedy `BurpExtender` (bez konštruktora) a následne zavolá metódu `registerExtenderCallbacks`. Pomocou tejto funkcie sa nainicializuje listener pre zachytávanie komunikácie a iné vybrané pomocné funkcie. Týmto sa rozšírenie inicializuje a určí sa na čo je dané rozšírenie zamerané[2].

Metóda `processHttpMessage` slúži na zachytávanie HTTP požiadaviek ako aj odpovedí. Rozlíšiť či daný paket je požiadavka klienta alebo odpoveď od serveru má na starosti parameter `messageIsRequest`. Ak sa jedná o požiadavku klienta, získa z tohoto paketu presnú URL adresu. V prípade, že sa jedná o paket odoslaný serverom, odchytí sa celá odpoveď pomocou parametru `message` a preloží sa z binárnej podoby na reťazec.

Takto upravenú odpoveď skontroluje pomocou dvoch regulárnych výrazov kde každý sa zameriava na iný druh možnej zraniteľnosti. Príklad 6.1 je regulárny výraz použitý pre odhalenie zraniteľných miest vstupov v zdrojovom kóde stránky a príklad 6.2 odhaľuje zraniteľnosť miest výstupov v tej istej odpovedi zaslanej zo servera. Tieto RegExy vytvorili pre potreby dizertačnej práce napísanej Zdravkom Danailovom a Krassenom Deltchevom s názvom *DOM-based XSS Attacks*[3] [8], no zoznam zraniteľných vstupov a výstupov je veľmi veľký. Dôvodom je, že skoro všetko môže byť manipulovateľné JavaScriptom a tým pádom sa každé takéto miesto môže stať potencionálnou zraniteľnosťou [4]. Napríklad pre lepšiu detekciu je v RegExe na detekciu výstupov (*sinks*), použitom v tomto nástroji, pridané v druhej zátvorke kľúčové slovo `location`.

Výstupom sú dva zoznamy čísel, kde každé číslo reprezentuje index na miesto s možnou zraniteľnosťou v odpovedi.

```
1 {((src|href|data|location|code|value|action)\s*[\"\'\]]*\s*\+?\s*=)|((replace|assign|
      navigate|getResponseHeader|open(Dialog)?|showModalDialog|eval|evaluate|execCommand|
      execScript|setTimeout|setInterval)\s*[\"\'\]]*\s*\()}
```

Príklad 6.1: RegEx použitý pre odhalenie vstupov (*sources)*

```
1 {(location\s*[\[.])|([.\[]\s*[\"\']?\s*(location|arguments|dialogArguments|innerHTML|write(
      ln)?|open(Dialog)?|showModalDialog|cookie|URL|documentURI|baseURI|referrer|name|opener|
      parent|top|content|self|frames)\W)|(localStorage|sessionStorage|Database)}
```

Príklad 6.2: RegEx použitý pre odhalenie výstupov (*sinks*)

Po dokončení skenovania odpovede, vypíše najskôr štatistiku nájdených predpokladaných zraniteľností daného webu a potom postupne kontroluje tieto miesta. Kontrola prebieha pomocou injektáže payloadami za použitia *headless* prehliadača Chromium. Bližšie je táto kontrola alebo testovanie zraniteľných miest popísané v sekcii 6.4.

[2] https://portswigger.net/burp/extender/writing-your-first-burp-suite-extension

[3] Na fakulte elektrotechniky a informačných technológií na univerzite Ruhr v Bochum v Nemecku

Funkcia `isAlertPresent` má na starosti kontrolu, či sa daný payload vykonal a teda potvrdí zraniteľné miesto. Do výpisu na výstupe sa pripíše dané miesto aj spolu s payloadom, ktorý potvrdil zraniteľnosť.

Stránka sa po tejto kontrole normálne načíta. Výsledný výpis skenovania je buď zobrazený v konzole nástroja Burp Suite alebo uložený do súboru.

6.4 Penetračné testovanie

Penetračné testovanie je proces tesovania zraniteľnosti webových aplikácií, ktorý sa skladá z dvoch hlavných častí. Prvá časť sa venuje nachádzaniu zraniteľných miest a druhá časť injektáži týchto miest:

- nastaviť útok,
- vykonať útok,
- zanalyzovať výsledky,
- oboznámiť s výsledkami pomocou výsledného reportu[9].

Práve prítomnosť druhej časti, tzv. etického hackerstva odlišuje penetračné testovanie od tzv. *vulnerability scanu*. Penetračné etické testovanie webu na zraniteľnosti, či už všeobecné alebo konkrétny typ zraniteľnosti, musí spĺňať isté pravidlá a obmedzenia.

Metodika penetračného testovania zahŕňa tri typy a to Black Box, White Box a Gray Box Testing. Black Box Testing je spôsob testovania kde tester nepozná žiadnu internú špecifikáciu, dizajn či štruktúru testovaného objektu. Je to podobné ako tzv. *blind test* a najpodobnejšie reálnemu útoku. White Box Testing je opozitum k Black Box Testingu. V tomto prípade je tester plne informovaný o testovacom objekte a Gray box testing sa nachádza niekde medzi Black a White testovaním to znamená, že tester pozná niektoré informácie o testovanom objekte ale samozrejme nemá úplný prístup ku počítaču.

Existujú dve metódy na vykonávanie testovania a to pomocou manuálnych testov alebo automatizovaných testov. Oboje majú svoje výhody a nevýhody. Niektoré zraniteľnosti je veľmi náročné odhaliť pomocou automatizovaných testov a teda je nutné použiť manuálny sken. V takýchto prípadoch je preto potrebné poznať špecificky webovú aplikáciu, jej funkcionalitu a tak isto aj systém na akom je založená a na základe toho môže penetračný tester vyrobiť konkrétny a lepšie cielený útok. Na druhej strane mnohé spoločnosti vyvíjajúce webové stránky alebo aplikácie používajú automatické nástroje na kontrolu ich produktov. Takýto nástroj po automatickom oskenovaní každej stránky nájde rôzne slabiny a vygeneruje súhrnnú správu o stave produktu. Nástrojov na vykonávanie automatizovaných testov je niekoľko, jedným z nich je aj Burp Scanner.

Rozšírenie vytvorené a popísané v tejto práci používa práve automatizované testy. Nájdené zraniteľné miesta injektuje vopred definovanými payloadami. Payloady boli vyberané na základe príkladov z trénovacieho servera stránky PortSwigger[4] a tiež inšpirované nástrojom DomGoat[5]. Ten nepoužíva metódu testovania fuzzing čo znamená, že do nájdených zraniteľných miest sa neposielajú náhodné dáta a nehľadá sa nejaké zlyhanie aplikácie[13] ale posiela sa len pár konkrétnych payloadov na injektáž stránky.

[4] https://portswigger.net/web-security/cross-site-scripting/dom-based
[5] https://domgo.at/cxss/intro

Konkrétne príklady výstupov *DOM–based XSS catcher*

V príklade 6.3 je znázornený reálny výstup z vyvinutého rozšírenia pre nástroj Burp Suite zo skenovania stránky `https://brutelogic.com.br/tests/sinks.html`, ktorá bola použitá ako testovacia stránka. Táto stránka bola vyvinutá Rodolfom Assisom[4] a je určená na testovanie nástrojov na odhaľovanie zraniteľností XSS DOM–based.

V tomto konkrétnom prípade, nástroj identifikoval postupne 5 možných zraniteľných miest na výstupe stránky (*sinks*) a 2 možné zraniteľné miesta na vstupe do webu (*source*). Zraniteľnosť sa potvrdila z celkového počtu 7 v 4 prípadoch a to s 3 odlišnými payloadami: `=alert(1)'`, `=javascript:alert(1)'`, `=<img+src+onerror=alert(1)>'`.

```
##########       NEW URL      ##########

URL:
https://brutelogic.com.br/tests/sinks.html
HOST NAME:
brutelogic.com.br
##########   BEGIN RESPONSE SCANNER   ##########

Full context of exact sink:
var currentSearch = document.location.search;
SINK:
.location.search
This sink has index 373 in response.

Full context of exact sink:
var currentSearch = document.location.search;
SINK:
location.search
This sink has index 374 in response.

Full context of exact sink:
document.getElementById('p1').innerHTML = 'Hello, ' + username + '!';
SINK:
.innerHTML = 'Hello, ' + username + '!'
This sink has index 574 in response.

Full context of exact sink:
document.location = redir;
SINK:
.location = redir
This sink has index 719 in response.

Full context of exact sink:
document.getElementById('p1').innerHTML = 'Current market index is ' + market.index + '.';
SINK:
.innerHTML = 'Current market index is ' + market.index + '.'
This sink has index 971 in response.


***    END OF SINKS    ***

Full context of exact source:
document.location = redir;
SOURCE:
location = redir;
  }
```

```
47
48    /*** Execution Sink ***/
49
50    var nasdaq = 'AAAA';
51    var dowjones = 'BBBB';
52    var sp500 = 'CCCC';
53
54    var market = [];
55    var index = searchParams.get('index').toString();
56
57    eval('market.index=' + index);
58
59    document.getElementById('p1').innerHTML = 'Current market index is ' + market.index +
        '.';
60
61 </script
62 This source has index 720 in response.
63
64 Full context of exact source:
65 eval('market.index=' + index);
66 SOURCE:
67 eval('market.index=' + index);
68
69    document.getElementById('p1').innerHTML = 'Current market index is ' + market.index +
        '.';
70
71 </script
72 This source has index 908 in response.
73
74
75 ***    END OF SOURCE   ***
76
77 ##########    END RESPONSE SCANNER    ##########
78
79 Summary of inspecting URL  : https://brutelogic.com.br/tests/sinks.html
80           number of sinks  : 5
81           number of sources: 2
82
83 ##########    BEGIN OF TESTING VULNERABILITIES    ##########
84
85 Variable named name has vulnerability with script: =<img+src+onerror=alert(1)>'
86
87 Variable named redir has vulnerability with script: =javascript:alert(1)'
88
89 Variable named index has vulnerability with script: =alert(1)'
90
91 Variable named index has vulnerability with script: =alert(1)'
92
93 ##########     END OF TESTING VULNERABILITIES     ##########
```

Príklad 6.3: Výstup zo skenovania stránky za použitia rozšírenia *DOM–based XSS catcher*

V príklade 6.4 je znázornený ďalší reálny výstup z rozšírenia *DOM–based XSS catcher* pre nástroj Burp Suite zo skenovania stránky `http://www.insecurelabs.org/Task/Rule1`,

ktorá bola použitá ako testovacia stránka. Táto stránka bola vyvinutá nadáciou OWASP[6] a jej hlavný cieľ je naučiť ako exploitovať a chrániť pred útokmi XSS.

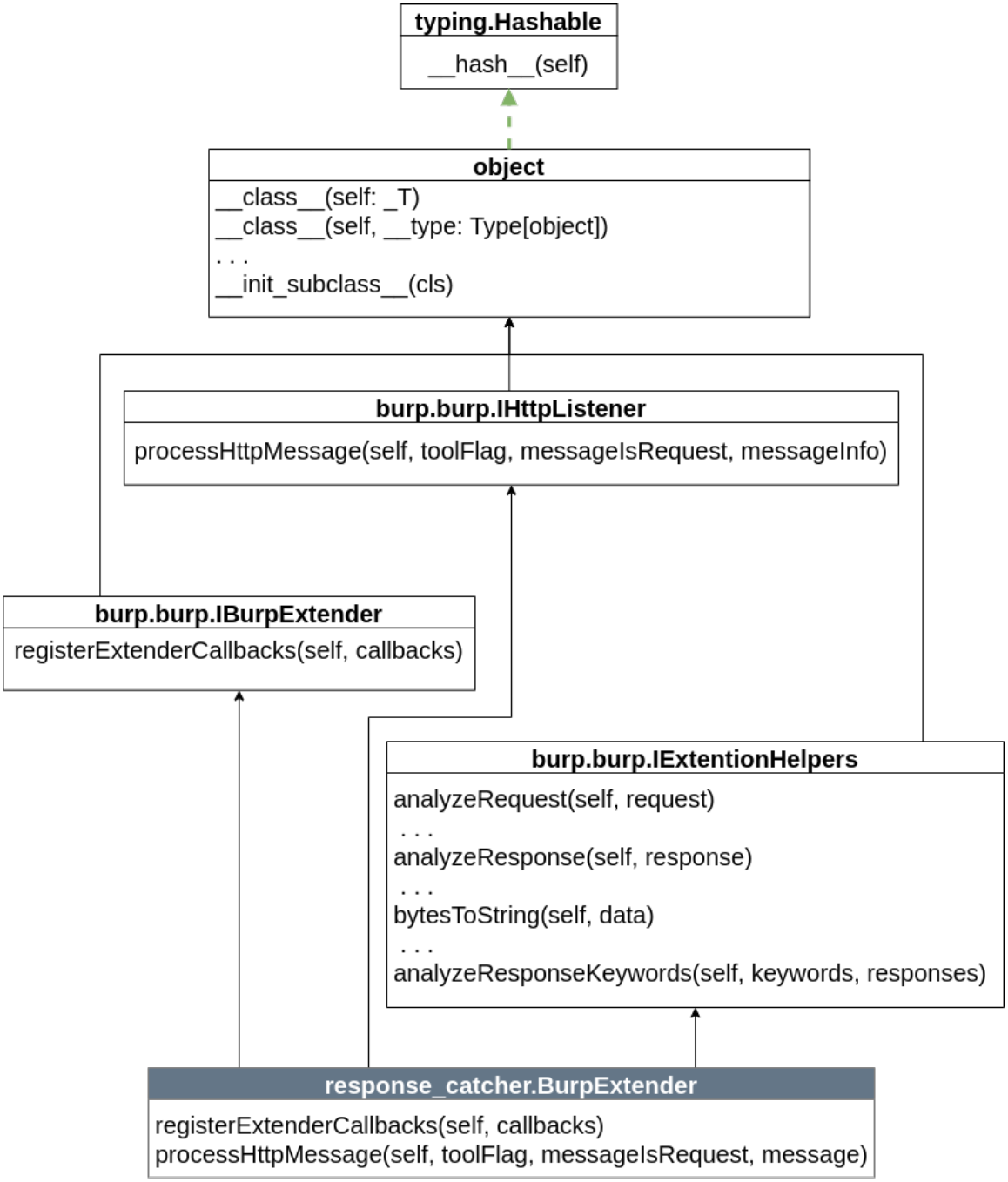
V tomto konkrétnom prípade, nástroj identifikoval iba zraniteľné miesta vstupe do webu (*source*) a to v počte 8. Zraniteľnosť sa potvrdila z celkového počtu len v 1 prípade a s payloadom `=<img+src+onerror=alert(1)>'`.

```
1  ##########       NEW URL      ##########
2
3  URL:
4  http://www.insecurelabs.org/Task/Rule1
5  HOST NAME:
6  www.insecurelabs.org
7  ##########   BEGIN RESPONSE SCANNER   ##########
8
9  Full context of exact source:
10 <link href="/Content/Task.css" rel="stylesheet" type="text/css" />
11 SOURCE:
12 href="/Content/Task.css" rel="stylesheet" type="text/css" /
13 This source has index 314 in response.
14
15 Full context of exact source:
16 <script src="/Scripts/jquery-1.5.1.min.js" type="text/javascript"></script>
17 SOURCE:
18 src="/Scripts/jquery-1.5.1.min.js" type="text/javascript"
19 This source has index 388 in response.
20
21 Full context of exact source:
22 <script src="../../Scripts/Task.js"></script>
23 SOURCE:
24 src="../../Scripts/Task.js"
25 This source has index 469 in response.
26
27 Full context of exact source:
28 <a href="/Task" class="back">Back to task list</a>
29 SOURCE:
30 href="/Task" class="back"
31 This source has index 593 in response.
32
33 Full context of exact source:
34 <form method="get" action="">
35 SOURCE:
36 action=""
37 This source has index 705 in response.
38
39 Full context of exact source:
40 Search: <input type="text" name="query" value=""/><input type="submit" value="Search" />
41 SOURCE:
42 value=""/
43 This source has index 765 in response.
44
45 Full context of exact source:
46 Search: <input type="text" name="query" value=""/><input type="submit" value="Search" />
47 SOURCE:
48 value="Search" /
49 This source has index 796 in response.
50
```

[6] http://www.insecurelabs.org/Task

```
Full context of exact source:
www.insecurelabs.org is an educational tool. It's hosted by <a href="http://appharbor.com">
    AppHarbor</a>
SOURCE:
href="http://appharbor.com"
This source has index 1308 in response.


***    END OF SOURCE   ***

##########    END RESPONSE SCANNER    ##########

Summary of inspecting URL  : http://www.insecurelabs.org/Task/Rule1
          number of sinks  : 0
          number of sources: 8

##########    BEGIN OF TESTING VULNERABILITIES    ##########

Variable named query has vulnerability with script: =<img+src+onerror=alert(1)>'

##########     END OF TESTING VULNERABILITIES     ##########
```

Príklad 6.4: Výstup zo skenovania stránky za použitia rozšírenia *DOM–based XSS catcher*



Obr. 6.1: Diagram tried pre nástroj *DOM–based XSS catcher*

Kapitola 7

Záver

V tejto práci bola opísaná Bezpečnosť aplikačnej vrstvy so zameraním na HTTP ako základný protokol webových aplikácií pre komunikáciu klient–server. Ďalej bol opísaný DOM (Document Object Model) ako dynamická, dátová, stromová štruktúra, tvoriaca rozhranie pre komunikáciu medzi jazykmi HTML a JavaScript. Na teoretickej úrovní práca analyzuje DOM, jeho uzly stromu, ktoré sú definované v HTML a tie zahrnuté v CSS kóde (Obr. 2.3).

Práca sa ďalej venuje práve útoku XSS typu DOM–based, preto kapitola 4 analyzuje tento typ útoku a jeho: princíp, kontext vykonávania, hrozby plynúce z útokov a navrhovanú ochranu.

Po navrhnutí detekcie XSS pomocou analýzy JavaScriptu som navrhla nastroj ako rozšírenie do proxy serveru Burp Suite. Burp Suite bol vybraný na základe požiadaviek zadávateľa témy práce (firma *AEC, spol. s r.o.*). Pre následnú detekciu zraniteľnosti bolo neimplementované rozšírenie, ktoré analyzuje JavaScript a úspešne detekuje zraniteľnosť DOM–based XSS. Následne nástroj vygeneruje report o zraniteľnosti skúmanej stránky (podkapitola 6.4). Vďaka Burp Suite bol nástroj *DOM–based XSS catcher* vyvinutý ako multiplatformový. Etické otestovanie zabezpečila vzorka webových stránok určených na testovanie zraniteľnosti, pomocou penetračných testov zahrnutých v implementácií rozšírenia. Praktickým výsledkom práce je teda nástroj na evaluáciu JavaScriptu vo webových aplikáciách zameraný na zraniteľnosť DOM–based XSS.

Literatúra

[1] Anonymous, A. *Web Application Vulnerability Report 2020* [online]. Acunetix, 2021 [cit. 2021-07-07]. Dostupné z: http://newtech.mt/wp-content/uploads/2021/01/Acunetix_2020_Web_Application_Vulnerability_Report.pdf.

[2] Anonymous, B. *Getting started with Burp Proxy* [online]. 2021 [cit. 2021–06-19]. Dostupné z: https://portswigger.net/burp/documentation/desktop/tools/proxy/getting-started.

[3] Anonymous, C. *HTML Tag* [online]. W3Schools, 2021 [cit. 2021–07-08]. Dostupné z: https://www.w3schools.com/tags/tag_meta.asp.

[4] Assis, R. *DOM-based XSS - The 3 Sinks* [online]. Nov 2018 [cit. 2021-07-05]. Dostupné z: https://brutelogic.com.br/blog/dom-based-xss-the-3-sinks/.

[5] Berners Lee, T., Fielding, R. a al. et. *RFC 3986, Uniform Resource Identifier (URI): Generic Syntax* [online]. 2005 [cit. 2021–07-07]. Dostupné z: https://datatracker.ietf.org/doc/html/rfc3986.

[6] Bultan, T., Yu, F., Alkhalaf, M. a Aydin, A. *String Analysis for Software Verification and Security*. Springer, Cham, 2017. 174 s. ISBN 978-3-319-68668-4. Dostupné z: https://doi.org/10.1007/978-3-319-68670-7.

[7] Buskirk, B. *The Advantages And Disadvantages Of Proxy Servers* [online]. Dec 2019 [cit. 2021–06-19]. Dostupné z: https://thinkcomputers.org/the-advantages-and-disadvantages-of-proxy-servers/.

[8] Danailov, Z. a Deltchev, K. *DOM-based XSS Attacks*. Bochum, Nemecko, 2012. M.Sc. Project thesis. Ruhr-University of Bochum, Faculty Of Electrical Engineering And Information Technology. Dostupné z: https://cupdf.com/document/dom-based-xss.html.

[9] Doshi, J. C. a Trivedi, B. H. *(PDF) Comparison of Vulnerability Assessment and Penetration Testing* [online]. Apr 2015. Dostupné z: https://www.researchgate.net/publication/276887993_Comparison_of_Vulnerability_Assessment_and_Penetration_Testing.

[10] Education, I. C. *What is Three-Tier Architecture* [online]. IBM, Oct 2020 [cit. 2021-07-07]. Dostupné z: https://www.ibm.com/cloud/learn/three-tier-architecture.

[11] Fielding, R., Gettys, J. a al. et. *RFC 2616, Hypertext Transfer Protocol – HTTP/1.1.* [online]. 1999 [cit. 2021–01-09]. Dostupné z: http://tools.ietf.org/html/rfc2616.

[12] Formánek, D. *Zranitelnosti typu injekce: XSS aneb cross-site scripting* [online]. root.cz [cit. 2021-05-22]. Dostupné z: https://www.root.cz/clanky/zranitelnosti-typu-injekce-xss-aneb-cross-site-scripting/.

[13] Goel, J. a Mehtre, B. Vulnerability Assessment & Penetration Testing as a Cyber Defence Technology. *Procedia Computer Science*. December 2015, zv. 57, s. 710–715. DOI: 10.1016/j.procs.2015.07.458. Dostupné z: https://www.researchgate.net/publication/283186664_Vulnerability_Assessment_Penetration_Testing_as_a_Cyber_Defence_Technology.

[14] Hyppönen, M. *Hypponen's law: Whenever an appliance is described as being "smart", it's vulnerable.* [@mikko].[Tweet]. (2016, December 12) [cit. 2021-07-03]. Dostupné z: https://twitter.com/mikko/status/808291670072717312.

[15] Inc., F. *Content Security Policy Reference* [online]. 2021 [cit. 2021–07-08]. Dostupné z: https://content-security-policy.com/.

[16] Inc., F. *Example a CSP header with a meta tag* [online]. 2021 [cit. 2021–07-08]. Dostupné z: https://content-security-policy.com/examples/meta/.

[17] Jovanovic, N., Krügel, C. a Kirda, E. Pixy: a static analysis tool for detecting Web application vulnerabilities. *2006 IEEE Symposium on Security and Privacy (S&P'06)*. 2006, s. 6 pp.–263. Dostupné z: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1624016.

[18] Kirda, E., Krügel, C., Vigna, G. a Jovanovic, N. Noxes: A client-side solution for mitigating cross-site scripting attacks. In:. Január 2006, s. 330–337. DOI: 10.1145/1141277.1141357. Dostupné z: https://sites.cs.ucsb.edu/~chris/research/doc/sac06_noxes.pdf.

[19] Klein, A. *DOM Based Cross Site Scripting or XSS of the Third Kind: A look at an overlooked flavor of XSS* [online]. 2005 [cit. 2021-01-12]. Dostupné z: http://www.webappsec.org/projects/articles/071105.shtml#r4.

[20] Kristen, S. *Cross Site Scripting (XSS)* [online]. The OWASP Foundation [cit. 2021-01-11]. Dostupné z: https://owasp.org/www-community/attacks/xss/.

[21] Kümmel, R. *XSS: Cross-Site Scripting v praxi*. Tigris, 2011. 332 s. ISBN 978-80-86062-34-1. Dostupné z: https://www.soom.cz/data/Cross-Site_Scripting_v_praxi__Roman_Kummel.pdf.

[22] MILLER, A. *Cross Site Scripting (XSS)* [online]. Snyk [cit. 2021-05-22]. Dostupné z: https://snyk.io/learn/cross-site-scripting/.

[23] NVD. *NATIONAL VULNERABILITY DATABASE* [online]. 2020 [cit. 2021-01-11]. Dostupné z: https://nvd.nist.gov/.

[24] Ocariza, F., Bajaj, K., Pattabiraman, K. a Mesbah, A. An Empirical Study of Client-Side JavaScript Bugs. In: *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*. 2013, s. 55–64. DOI: 10.1109/ESEM.2013.18. Dostupné z: https://ieeexplore.ieee.org/abstract/document/6681338.

[25] OWASP. *OWASP Top Ten* [online]. The OWASP Foundation, 2017 [cit. 2021-01-10]. Dostupné z: https://owasp.org/www-project-top-ten/.

[26] Owasp. *DOM based XSS Prevention Cheat Sheet* [online]. The OWASP Foundation, 2021. Dostupné z: https://github.com/OWASP/CheatSheetSeries/blob/6ffcbc5f3559b83d6cabf3a685d9a8aad22f0910/cheatsheets/DOM_based_XSS_Prevention_Cheat_Sheet.md.

[27] Pahi, N. *Client Server Database And Its Applications* [online]. Our Education | Best Coaching Institutes Colleges Rank, 2013 [cit. 2021–01-08]. Dostupné z: https://blog.oureducation.in/client-server-database-applications/.

[28] Petters, J. *What is a Proxy Server and How Does it Work?* [online]. May 2021 [cit. 2021–06-18]. Dostupné z: https://www.varonis.com/blog/what-is-a-proxy-server/.

[29] Rescorla, E., Gettys, J. a al. et. *RFC 2818, HTTP Over TLS* [online]. 2000 [cit. 2021–01-09]. Dostupné z: https://datatracker.ietf.org/doc/html/rfc3986.

[30] Škultéty, R. *JavaScript: programujeme internetové aplikace*. Computer Press, 2004. ISBN 9788025101445. Dostupné z: https://books.google.cz/books?id=YFQyAAAACAAJ.

[31] Stuttard, D. a Pinto, M. *The Web Application Hacker's Handbook: Finding and Exploiting Security Flaws*. Wiley, 2011. ISBN 9781118175224. Dostupné z: https://books.google.cz/books?id=jMkB4Q4lzpcC.

[32] Wichers, D. *Unraveling some of the Mysteries around DOM-based XSS* [online]. 2012 [cit. 2021–01-11]. Dostupné z: https://owasp.org/www-pdf-archive/Unraveling_some_Mysteries_around_DOM-based_XSS.pdf.

[33] Zakas, N. Z. *JavaScript pro webové vývojáře*. Computer Press, 2009. 832 s. ISBN 978-80-251-2509-0. Dostupné z: http://www.digitalniknihovna.cz/mzk/uuid/uuid:64cae000-e1cd-11e4-82a1-005056827e52.

Príloha A

Slovník

API	Application Programming Interface
CGI	Common Gateway Interface
CSP	Content Security Policy
DOM	Document Object Model
HTML	HyperText Markup Language
HTTP(S)	HyperText Transfer Protocol (Secure)
IPS	Intrusion prevention system policy
LDAP	Lightweight Directory Access Protocol
OWASP	nadácia Open Web Application Security Project
RegEx	Regular Expression
SSL	Secure Socket Layer
SQL	Structured Query Language
TCP	Transmission Control Protocol
TLS	Transport Layer Security
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
W3C	World Wide Web Consortium
XML	eXtensible Markup Language
XSS	Cross-Site Scripting

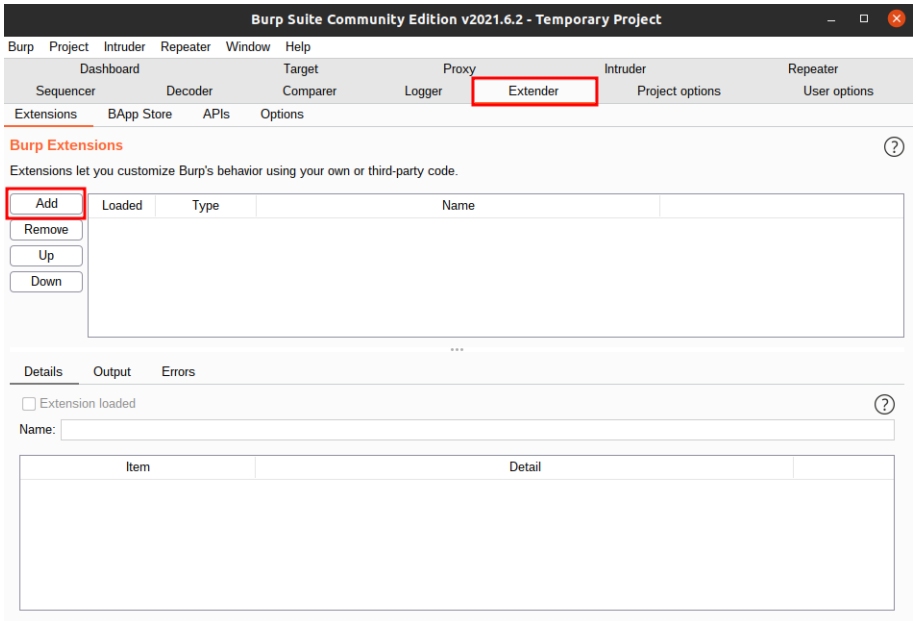
Príloha B

Manuál pre použitie nástroja `response_catcher.py`

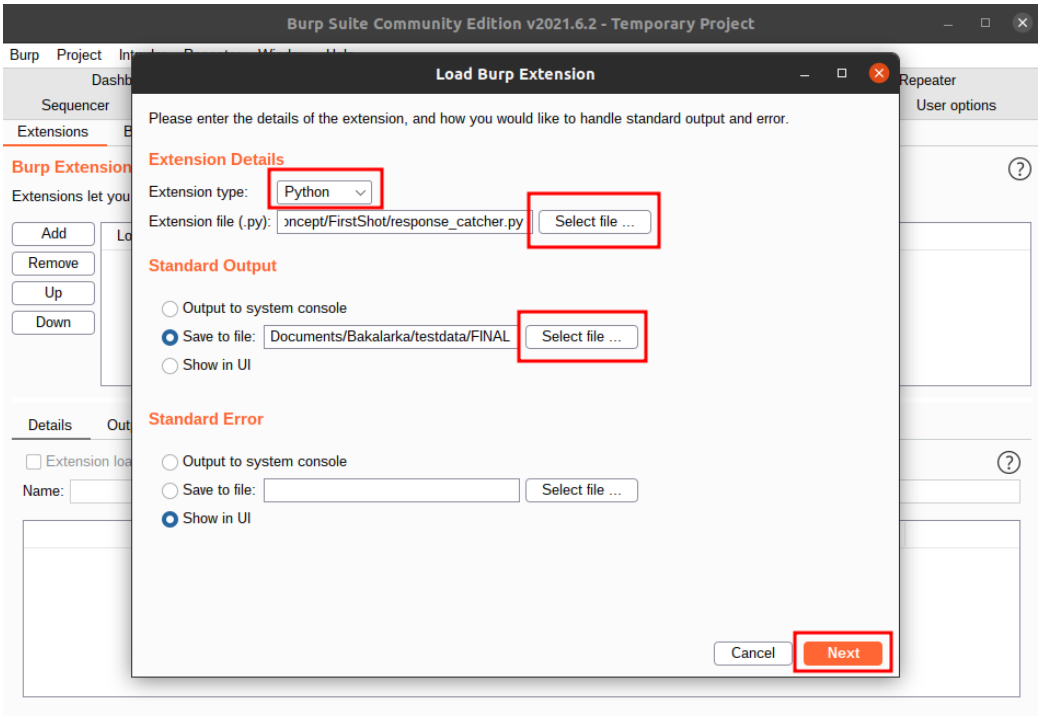
Po nainštalovaní Burp Suite a stiahnutí `response_catcher.py` otvoríme Burp Suite, prejdeme na záložku *Extender* a klikneme na tlačidlo *Add* (viz. obrázok B.1).

V okne, ktoré sa otvorí vyberieme v *Extention type* možnosť Python a v *Extention file* cez tlačidlo *Select file...* vyberieme súbor `response_catcher.py`. V *Standard Output* sú tri možnosti, kde sa zobrazí výpis z výsledku skenovania. Je odporúčané vybrať možnosť *Save to file* a cez tlačidlo *Select file...* vybrať cieľový súbor, v ktorom sa uloží tento výpis. Potvrdíme cez tlačidlo *Next* a tým sa rozšírenie načíta do Burp Suite (viz. obrázok B.2).

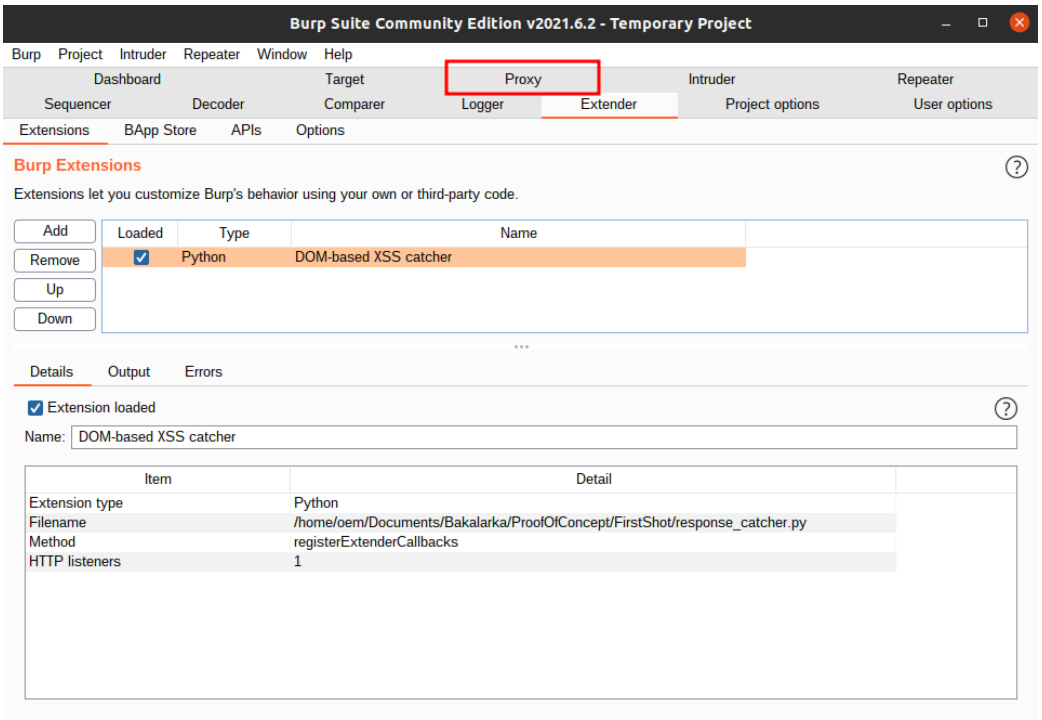
Po úspešnom načítaní je možné skontrolovať nastavenia nie len na uloženie výsledného výpisu ale aj prípadné chybové hlášky. Tie je možné nájsť pod záložkou *Errors*. Teraz vyberieme záložku *Proxy* na hornej lište (viz. obrázok B.3). Skontrolujeme tlačidlo *Intercept is off* či je správne nastavené na vypnuté a otvoríme si *headless browser* pomocou tlačidla *Open Browser*(viz. obrázok B.4). V novootvorenom okne zadané požadovanú url adresu a po jej načítaní máme výsledok detekcie v súbore prípadne v konzole.



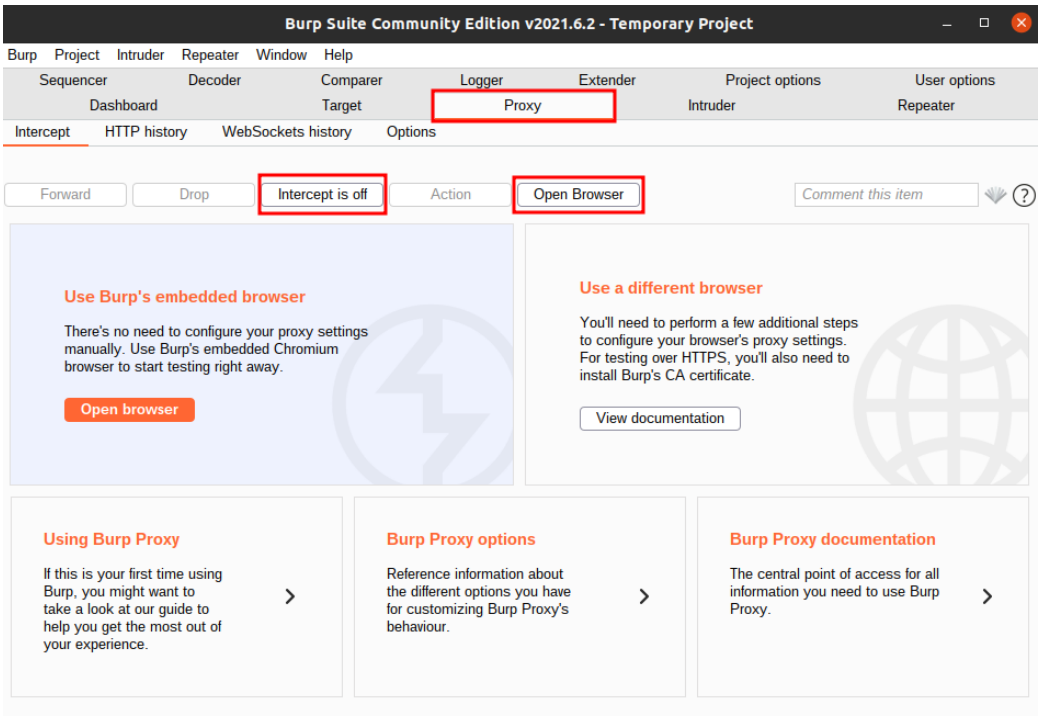
Obr. B.1: Pridanie rozšírenia do Burp Suite.



Obr. B.2: Pridanie rozšírenia do Burp Suite (pokrač.).



Obr. B.3: Pridanie rozšírenia do Burp Suite (záver).



Obr. B.4: Spustenie cez *headless browser*.

Príloha C

Obsah priloženého pamäťového média

Priložené CD obsahuje:

- DOM-based XSS catcher/response_catcher.py - zdrojové kódy nástroja,
- DOM-based XSS catcher/README.md - návod na spustenie nástroja,
- xbarno00.zip - zdrojové kódy pre LaTeX,
- xbarno00.pdf - text bakalárskej práce.