VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

DETEKCE PODEZŘELÝCH SÍŤOVÝCH POŽADAVKŮ WEBOVÝCH STRÁNEK

DETECTION OF SUSPICIOUS REQUESTS MADE BY WEB PAGES

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. PAVEL POHNER**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. LIBOR POLČÁK, Ph.D.**
SUPERVISOR

BRNO 2020

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav informačních systémů (UIFS) Akademický rok 2019/2020

Zadání diplomové práce

Student: **Pohner Pavel, Bc.**
Program: Informační technologie Obor: Bezpečnost informačních technologií
Název: **Detekce podezřelých síťových požadavků webových stránek**
Detection of Suspicious Requests Made by Web Pages
Kategorie: Bezpečnost

Zadání:

1. Seznamte se s projektem JavaScript Restrictor a používanou částí rozhraní WebExtensions a nastudujte API webRequest.
2. Seznamte se s politikami webového prohlížeče, které omezují přístupné zdroje pro načtené stránky (Same Origin Policy).
3. Popište metody dovolující webové stránce používat webový prohlížeč jako proxy při přístupu do lokální sítě obcházející Same Origin Policy.
4. Navrhněte mechanismus detekující podezřelé požadavky webových stránek s podezřením na obcházení Same Origin Policy. Mechanismus musí být vhodný pro nástroj JavaScript Restrictor.
5. Návrh implementujte kvalitním a komentovaným kódem, aby byl vedoucím práce akceptován do projektu JavaScript Restrictor.
6. Implementaci otestujte.
7. Práci vyhodnoťte a navrhněte možná zlepšení.

Literatura:

- TIMKO, Martin. *Vylepšení rozšíření pro omezení volání JavaScriptu*. Brno, 2019. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií.
- BERGBOM, John. Attacking the Internal Network From the Public Internet Using a Browser as a Proxy. Forcepoint Research Report, 2019. Dostupné online na https://www.forcepoint.com/sites/default/files/resources/files/report-attacking-internal-network-en_0.pdf.
- DI PAOLA, Stefano a FEDON Giorgio. Subverting Ajax. 23rd Chaos Communication Congress, 2006.

Při obhajobě semestrální části projektu je požadováno:

- Body 1 až 4 zadání.

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

Vedoucí práce: **Polčák Libor, Ing., Ph.D.**
Vedoucí ústavu: Kolář Dušan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2019
Datum odevzdání: 20. května 2020
Datum schválení: 29. října 2019

Abstrakt

Cílem této práce je zamezit webovým stránkám ve veřejné síti internet zneužívat webový prohlížeč uživatele jako proxy (prostředníka). V řešení tohoto problému byly využity znalosti o bezpečnostním mechanismu moderních prohlížečů – same-origin policy a možnostech implementace prohlížečových rozšíření pomocí WebExtensions. Navržené řešení využívá aplikační rozhraní WebRequest, které se zaměřuje na zachytávání HTTP požadavků, a rozšiřuje funkcionalitu již existujícího prohlížečového rozšíření JavaScript Restrictor o schopnost detekce a ochrany webového prohlížeče před zneužitím jako proxy pro skenování a vykonávání akcí ve vnitřní síti uživatele. Implementované řešení bylo otestováno a přijato jako součást JavaScript Restrictoru. Hlavním přínosem této práce je ochrana před možným zneužitím prohlížeče jako proxy, která v existujících rozšířeních chybí.

Abstract

The purpose of this thesis is to prevent websites located in public internet from accessing user's internal network through web browser. Acquired knowdledge about modern browser's security mechanism – same-origin policy and options of implementing the web browser extensions using WebExtensions, was used in the solution. Proposed solution is based on WebRequest API, which intercepts and modifies HTTP requests, and extends functionality of existing browser extension JavaScript Restrictor with the ability to detect and prevent the browser to be abused as a proxy for scanning and accessing user's internal network. The implemented solution was tested and accepted as a part of JavaScript Restrictor. The main benefit of this thesis is the protection from possible abusement of a web browser as a proxy, which is not present in existing extensions.

Klíčová slova

WebExtensions, WebRequest API, bezpečnost internetových prohlížečů, politika stejného původu, JavaScript, JavaScript Restrictor, zneužití prohlížeče jako proxy

Keywords

WebExtensions, WebRequest API, web browser security, same-origin policy, JavaScript, JavaScript Restrictor, abusement of the web browser as proxy

Citace

POHNER, Pavel. *Detekce podezřelých síťových požadavků webových stránek*. Brno, 2020. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Libor Polčák, Ph.D.

Detekce podezřelých síťových požadavků webových stránek

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Libora Polčáka Ph.D. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
Pavel Pohner
28. května 2020

Poděkování

Chtěl bych poděkovat Ing. Liborovi Polčákovi, Ph.D., za vedení práce, přínosné konzultace a cenné rady, které vedly k úspěšnému dokončení této práce.

Obsah

Kapitola 1

Úvod

V dnešní době internetové prohlížeče slouží jako nástroj k prohlížení webových stránek v síti internet a denně je k tomuto účelu využívá nespočet uživatelů. Prohlížeč ovšem nemusí vždy jen sloužit svým uživatelům, ale může být rovněž zneužit útočníkem. Přitom se nejedná o žádné „slabé místo“ implementace internetových prohlížečů, ale jejich návrhu, neboť prohlížeč jen v dobré víře vykonává to, k čemu je určený. Útočník může využít prohlížeč jako nástroj pro skenování lokální sítě uživatele, když bude požadovat po prohlížeči, aby načetl například obrázek z lokální IP adresy, ke které by sám neměl mít přístup [4]. Toto chování do jisté míry obchází veškeré bezpečnostní mechanismy prohlížeče, neboť pro prohlížeč se toto jeví jako validní akce, ze které ovšem profituje útočník.

Cílem této práce bylo nastudovat politiku stejného původu (angl. *same-origin policy*), což je bezpečností mechanismus internetových prohlížečů, a také aplikační prostředí WebExtensions, které slouží pro implementaci prohlížečových rozšíření. Následně se seznámit s existujícími prohlížečovými rozšířeními JavaScript Restrictor, uMatrix a NoScript a prostudovat možnosti útoku na lokální síť z veřejné sítě internet s využitím internetového prohlížeče jako proxy. Na základě získaných informací potom navrhnout mechanismus detekce a ochrany proti útokům, které využívají prohlížeč jako proxy. Tento navržený mechanismus následně implementovat pomocí aplikačního prostředí WebExtensions a to takovým způsobem, aby bylo umožněno jak samostatné použití řešení, tak i integrace do existujícího rozšíření JavaScript Restrictor, implementaci otestovat, vyhodnotit a navrhnout možnosti dalšího vývoje.

V úvodní kapitole této práce je představen fundamentální bezpečností mechanismus internetových prohlížečů – politika stejného původu, následuje kapitola věnující se prohlížečovým rozšířením a možnostem jejich implementace, v pořadí čtvrtá kapitola obsahuje analýzu útoku využívajícího internetový prohlížeč jako proxy, pátá kapitola je věnována existujícím rozšířením, jmenovitě JavaScript Restrictoru, uMatrixu a NoScriptu, a konečně v šesté kapitole je navrženo samotné řešení detekce a ochrany proti útoku, který zneužívá prohlížeč jako proxy. Kapitoly 7 a 8 jsou potom věnovány praktické části práce, tedy implementaci navrženého řešení, jeho otestování a konečnému vyhodnocení dosažených výsledků. V závěru práce jsou rovněž vytknuty nedostatky implementovaného řešení a možnosti jejich nápravy a dalšího vývoje.

Kapitola 2

Politika stejného původu

Politika stejného původu, v angličtině taktéž známá jako same-origin policy, je bezpečnostní mechanismus, který omezuje způsob, kterým můžou webové stránky přistupovat k datům jiných webových stránek nebo aplikací [3]. Bez politiky stejného původu by každá stránka mohla přistoupit k objektovému modelu dokumentu (DOM – Document Object Model) jakékoliv jiné stránky. To by mohlo vést k potencionálnímu vyzrazení citlivých dat, ale i k vykonávání různých akcí bez vědomí uživatele. Politika stejného původu se však netýká pouze přístupu a manipulace s DOM dané webové stránky, je to soubor bezpečnostních pravidel, který ošetřuje například i *XMLHTTPRequest* (HTTP požadavky vytvořené klientským JS – Ajax) a cookies [22].

Politika stejného původu není přímo internetovým standardem (dle [3] se jedná o tzv. *proposed standard*, tj. navrhovaný standard), ale spíše bezpečnostním konceptem, který využívají internetové prohlížeče. Její implementace se liší prohlížeč od prohlížeče a rovněž se aplikuje odlišně na různé entity [22]. Nicméně, původní myšlenka zůstává zachována – zabránit nepovolenému přístupu napříč webovými stránkami (angl. *cross-site*).

2.1 Původ

Pro politiku stejného původu je extrémně důležité jakým způsobem je definován původ (angl. *origin*). Ve většině případů je původ definován jako trojice – schéma, doménové jméno, port [12]. Přičemž platí, že každá entita vyskytující se na webu má touto trojicí určený svůj původ, který odvozuje ze své URL (Uniform Resource Locator).

- Schéma – protokol, HTTP a HTTPS jsou odlišné protokoly, proto je odlišné i schéma
- Host – doménové jméno daného původu, fit.vutbr.cz je odlišné od fsi.vutbr.cz, přestože doména je stejná
- Port – port daného zdroje, nemusí být explicitně uvedený, v každém případě vutbr.cz:80 je odlišný od vutbr.cz:443

Ve chvíli, kdy internetový prohlížeč zobrazuje určitou webovou stránku, dochází k vytvoření této trojice na základě URL navštívené webové stránky. Pokud webová stránka načítá zdroje z jiných URL (typicky se jedná o skripty, CSS dokumenty, obrázky aj.), je pro každou URL vytvořena nová trojice, která se porovnává s originální trojicí původní stránky [12]. Některé příklady porovnání jsou k nahlédnutí v tabulce 2.1, která zobrazuje výsledky porovnání různých URL s referenční URL: *http://example.com/dir/page.html*.

URL	Výsledek porovnání	Důvod
http://example.com/dir2/other.html	Stejný původ	Liší se pouze cesta
http://example.com/dir/inner/another.html	Stejný původ	Liší se pouze cesta
https://example.com/page.html	Odlišný původ	Liší se protokol
http://example.com:81/dir/page.html	Odlišný původ	Liší se port
http://new.example.com/dir/page.html	Odlišný původ	Liší se doména

Tabulka 2.1: Výsledky porovnání různých URL.

2.2 Přístup napříč původy

Přístup napříč původy (angl. *cross-origin*) specifikuje jakým způsobem spolu mohou interagovat dva elementy s různým původem. Kontrola původu je prováděna prohlížečem vždy, když se jedná o potenciální interakci mezi elementy s různým původem. Politika stejného původu interakci mezi elementy dvou odlišných původů nezakazuje, ale aplikuje na ni jistá pravidla. Každá interakce mezi elementy s různým původem může být potenciálně nebezpečná, nicméně je nezbytné, aby alespoň za určitých podmínek byla povolena. Z tohoto důvodu se typicky přístup napříč původy rozděluje do následujících kategorií [12]:

- Zápis napříč původy (angl. *cross-origin writes*) – zde se jedná o to, zda je možné odesílat data stránce jiného původu. Zápis bývá zpravidla povolený a to z toho důvodu, aby bylo například možné odesílat data z formulářů nebo využívat hypertextové odkazy na jiné webové stránky [3]. Pro webovou stránku, která je cílem zápisu napříč původy neznamená tato funkcionalita žádné riziko, neboť může sama rozhodnout o tom, jak s přijatými daty naloží.

- Čtení napříč původy (angl. *cross-origin reads*) – zde se jedná o to, zda stránka jednoho původu může volně přistupovat k DOM stránky jiného původu a číst její data. Čtení by mělo být ze zřejmých důvodů ve většině případů zakázáno. Za určitých podmínek jej lze však povolit. Například spouštění skriptů, vykreslování obrázků nebo aplikace kaskádových stylů (CSS – *cascade stylesheets*) z míst s odlišným původem je povoleno, ale pro tato čtení platí speciální pravidla. Skripty se spouští v kontextu stránky, která jej načetla, obrázky jsou pouze vykresleny, nelze přistupovat k jejich datové hodnotě a rovněž nelze číst obsah načtených CSS souborů [3]. Tento mechanismus se označuje též jako vkládání napříč původy (angl. *cross-origin embedding*).

2.3 Vkládání napříč původy

Pro lepší pochopení mechanismu vkládání napříč původy uvažujme následující situaci – vývojář vytváří webovou stránku, kde chce využít skript jQuery, webový prohlížeč na základě vývojářova kódu načte jQuery skript a vloží jej do dané webové stránky. Vývojář může spouštět tento skript, ale nemůže číst obsah vloženého `<script>` tagu, jelikož tam byl vložen interně prohlížečem, který ho načetl z jiného zdroje a pokud by prohlížeč čtení tohoto `<script>` tagu umožnil, došlo by k porušení politiky stejného původu, neboť čtení napříč původy, jak je uvedeno výše, by mělo být v každém případě zakázáno. Dalším příkladem může být vložení obrázku, který je volně dostupný na internetu, do webové stránky. Pokud vývojář do své webové stránky vloží `<img>` tag s příslušným URL onoho obrázku, prohlížeč přečte obrázek a zobrazí jej na webové stránce, nemůže však přistoupit k obsahu

obrázku (tedy například zapsat bytestream do souboru). Vývojář může typicky přistoupit k některým atributům tohoto obrázku, jako je výška nebo šířka, taktéž se dozví, zdali se obrázek v pořádku načetl, nebo zdali se vyskytla nějaká chyba. Pokud nastala chyba, nedokáže ovšem zjistit jaká, neboť prohlížeč mu tuto informaci z důvodu politiky stejného původu nemůže poskytnout.

Tento mechanismus značně zjednodušuje vývoj webových stránek i jejich udržování. Je zde taktéž z důvodu zpětné kompatibility s webovými stránkami z dob, kdy politika stejného původu ještě nebyla rozšířena. Článek [22] uvádí, že v dobách před politikou stejného původu, kdy používání různého obsahu (obrázky, CSS, skripty aj.) z jiných zdrojů nebyl žádný problém, bylo implementováno značné množství webových stránek využívajících externí obsah a kdyby mechanismus vkládání napříč původy nebyl zaveden, znamenalo by to kolaps těchto webových stránek, neboť by nemohly k těmto zdrojům vůbec přistupovat. Co se týče vývoje webových stránek, při plně striktní politice stejného původu bez vkládání napříč původy, by bylo nezbytné každý obrázek, skript i stylový dokument lokálně uložit na serveru a číst ho lokálně. Pokud by byl originální skript upraven, znamenalo by to znovu ho manuálně stáhnout a uložit lokálně na serveru.

2.4 Zmírnění politiky stejného původu

Politika stejného původu se aplikuje v každém případě, ve kterém dochází ke komunikaci dvou webových stránek s odlišným původem, je ovšem možné tuto politiku explicitně zmírnit pro stránky s konkrétním původem [16]. Toho lze například využít při správě webových stránek s odlišnými doménovými jmény, které ale obě patří stejnému vlastníkovi. Může si být tedy jistý, že ani jedna z těchto stránek nepředstavuje bezpečnostní riziko a sdělit prohlížeči, aby v tomto případě udělil výjimku a politiku stejného původu v tomto konkrétním případě neuplatňoval.

2.4.1 Nastavení domény

Nejjednoduší cestou jak toho dosáhnout, je změnit doménu stránky (a tím i její původ). Lze k tomu využít deklaraci v jazyku JavaScript:

```
document.domain = "vutbr.cz"
```

Dle bezpečnostní příručky od Michala Zalewskeho [22], je ovšem toto možné využít pouze pro stránky ve stejné doménové hierarchii, tedy například stránka *fit.vutbr.cz* může zobecnit své doménové jméno právě tímto příkazem na *vutbr.cz*. Pro stránky mimo doménovou hierarchii je toto zakázáno, jinak by si každá stránka mohla nastavovat doménu dle svého uvážení. Tato metoda tedy umožňuje zmírnit omezení politiky stejného původu pouze v ohledu doménového jména, omezení na schéma a port zůstávají zachována.

2.4.2 Sdílení zdrojů napříč původy

Sdílení zdrojů napříč původy (v angličtině *cross-origin resource sharing*, dále jen CORS) označuje HTTP mechanismus [20], který umožňuje explicitně povolit stránky s konkrétním původem. Využívá k tomu dodatečné HTTP hlavičky, které jsou často označovány jako *CORS headers* [16]. Pomocí těchto hlaviček lze informovat prohlížeč, že zdroje ze stránek s konkrétním původem mají právo přistupovat k datům této stránky. CORS přidává do hlaviček HTTP požadavků další hlavičku *Origin*, čímž sděluje serveru původ požadavku.

V dokumentaci prohlížeče Firefox [14] autoři rozlišují dva typy požadavků – jednoduché (angl. *simple*) a tzv. *preflight* požadavky, tedy takové, které před svým odesláním vyžadují ještě dodatečné ověření od serveru. Do kategorie jednoduchých požadavků spadají všechny požadavky splňující následující podmínky (pozn. nejsou uvedeny všechny podmínky, pro kompletní seznam viz [14]):

- Povolené metody GET, POST a HEAD
- Hlavičky, které je povoleno manuálně nastavit:
  - *Accept*
  - *Accept-language*
  - *Content-Type*
  - *DPR*
  - *Downlink*
  - *Save-Data*
  - *Viewport-Width*
  - *Width*
- Povolené hodnoty pro hlavičku *Content-Type*:
  - *application/x-www-form-urlencoded*
  - *multipart/form-data*
  - *text/plain*

Jednoduché požadavky jsou přímo odeslány serveru. V případě, že server přijme tento požadavek, vrátí odpověď obsahující pole *Access-Control-Allow-Origin*, čímž sděluje, které původy stránek akceptuje. Pokud odpověď obsahuje v tomto poli původ uvedený v požadavku, prohlížeč odpověď zpracuje.

Do kategorie *preflight* požadavků spadají všechny ostatní, které nesplňují kritéria jednoduchého požadavku. *Preflight* požadavky musí být nejprve schváleny serverem před tím, než mohou být odeslány. Před odesláním prohlížeč nejprve zasílá požadavek metodou OPTIONS, ve které sděluje svůj původ a dále přidává pole *Access-Control-Request-Method*, kde sděluje o jakou metodu se bude jednat, a *Access-Control-Request-Headers*, kde sděluje jaké hlavičky se budou vyskytovat v požadavku, pro který žádá potvrzení [16]. V odpovědi na tento požadavek server opět sděluje, které původy jsou povolené (*Access-Control-Allow-Origin*), které metody jsou povolené (*Access-Control-Allow-Methods*) a které hlavičky akceptuje (*Access-Control-Allow-Headers*) [20].

2.5 Zranitelnost politiky stejného původu

Jak bylo zmíněno výše, vkládání napříč původy sice umožňuje sdílení obsahu mezi stránkami s odlišným původem, ale dá se využít i pro nečestné účely. V podkapitole 2.3 bylo zmíněno, že politika stejného původu sice nezpřístupní obsah (například obsah tagu `<img>`) stránce s jiným původem, nicméně i informace o tom, že se tento obsah podařilo v pořádku načíst mnohdy útočníkovi stačí a je to tím pádem inkriminující informace, která by neměla být získatelná. Demonstrujme si tuto zranitelnost na následujícím příkladu – na výpisu 2.1

je k nahlédnutí krátký úsek JavaScript kódu, který je umístěný na serveru v síti internet. Tato část kódu načítá obrázek z privátní IP adresy (řádek 17), jedná se o výchozí IP adresu routeru v lokální síti s cestou k obrázku loga jeho výrobce. Pokud uživatel přistoupí na tento server ze své domácí sítě, prohlížeč bez jakýchkoliv problémů přistoupí na tuto privátní IP adresu a načte obrázek loga routeru. Útočník na straně serveru nemůže k tomutu obrázku přímo přistoupit a například si ho uložit, nicméně s využitím tohoto kódu (vlastnost *onload* – řádky 8-16) dokáže zjistit, zda se obrázek načetl, čímž automaticky zjistil, na které IP adrese ve vnitřní síti se router nachází a dokonce i od jakého výrobce je. Řádek číslo 12 demonstruje, že je možné přistoupit k rozměrům načteného obrázku i v případě aktivní politiky stejného původu. Na řádku číslo 15 se potom nachází pokus o vypsání obsahu obrázku, získaného ze stránky jiného původu, který skončí chybovým hlášením prohlížeče, protože se jedná o porušení zásad politiky stejného původu.

```
1  function get_image() {
2    var image = new Image();
3    image.onerror = function () {
4      if (!image) {
5        return;
6      }
7    };
8    image.onload = function (val) {
9      if (!image) {
10       return;
11     }
12     console.log(image.height);
13     document.getElementById("mycanvas").getContext('2d').drawImage(image, 0, 0);
14     //SOP viaolation
15     console.log(document.getElementById("mycanvas").toDataURL("image/png"));
16   };
17   image.src = 'http://192.168.1.1/images/New_ui/asustitle.png';
18 }
```

Výpis 2.1: Útržek JavaScript kódu načítající obrázek z lokální IP adresy.

Kapitola 3

Rozšíření prohlížeče – WebExtensions

Kniha pojednávající o tvorbě prohlížečových rozšíření [15] popisuje rozšíření prohlížeče jako programy, které běží v kontextu internetového prohlížeče a umožňují rozšířit, nebo modifikovat jeho chování. Dokáží například přebarvit webové stránky na vybranou barvu, skrývat reklamy nebo třeba modifikovat záložky. V dnešní době každý rozšířený internetový prohlížeč poskytuje vývojářům API (Application Programming Interface) pro programování těchto rozšíření. Zpočátku měl každý prohlížeč vlastní API, takže naprogramovaná rozšíření nebyla příliš kompatibilní s ostatními prohlížeči. S rostoucí popularitou prohlížeče Chrome od společnosti Google se však objevila snaha tato API určitým způsobem sjednotit a standardizovat. V současné době je WebExtensions API nejblíže tomu, co by se dalo nazývat standardizovaným API pro vývoj prohlížečových rozšíření. I zde však existují drobné rozdíly napříč prohlížeči, které budou popsány v následující podkapitole.

3.1 Kompatibilita WebExtensions

V současné době se prohlížeče dělí na tři hlavní skupiny, jsou to – prohlížeče založené na jádru Chromium (open-source projekt od společnosti Google), kam patří například Chrome, nebo Opera, dále Microsoft Edge, který je v aktuální verzi sice taktéž založený na Chromiu, ale je na tolik odlišný, že ho nelze zařadit do první skupiny, a poslední skupinou, opět pouze s jedním reprezentantem, je prohlížeč Firefox [10]. Mezi těmito třemi skupinami jsou drobné rozdíly v kompatibilitě. Všechny tyto rozdíly jsou zevrubně popsány v článku od vývojářů prohlížeče Firefox [10].

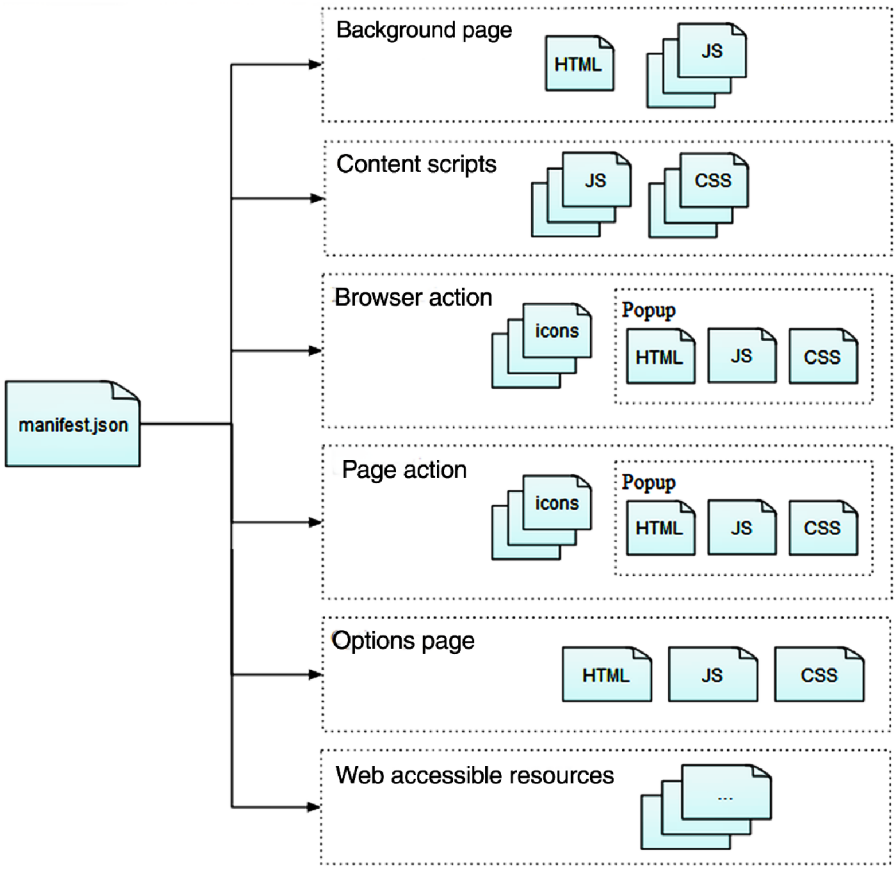
Prvním rozdílem je proměnná (přesněji prostor jmen), přes kterou se přistupuje k základním JavaScriptovým API. V prohlížečích založených na Chromiu se přistupuje přes proměnnou **chrome** (například tedy *chrome.webRequest*) naproti tomu ve Firefoxu se používá jméno **browser** (tedy *browser.webRequest*) [10].

Druhým rozdílem je způsob, jakým se implementují asynchronní funkce. V prohlížečích založených na Chromiu se asynchronní volání řeší tzv. **callbackem** [10] (zpětné volání) – asynchronní funkce očekává v parametru další funkci, kterou zavolá po svém dokončení. Naproti tomu Firefox používá modernější JavaScriptový objekt – **Promise** [10], který asynchronní funkce vrací a kterého je možné se dotázat, zda funkce dopadla úspěšně, či neúspěšně a taktéž k těmto případům navázat korespondující funkce. Taktéž se napříč prohlížeči může lišit dostupnost některých funkcí a API, například prohlížeč Firefox nabízí

DNS API, které umožňuje vytvářet dotazy na DNS server a překládat doménová jména na IP adresy, oproti tomu Chrome ani Opera toto, ani žádné obdobné API, vývojářům nenabízí, což vylučuje, aby rozšíření využívající toto API v prohlížeči Firefox bylo kompatibilní i s ostatními prohlížeči.

Posledním větším rozdílem je konfigurační soubor **mafinest.json**, který obsahuje základní informace o rozšíření a je využíván jako vstupní soubor rozšíření, kde prohlížeč hledá všechny důležité informace (např. jméno, verzi, cestu ke zdrojovým souborům atp.). Struktura tohoto souboru se liší od prohlížeče k prohlížeči, při vývoji kompatibilního rozšíření je tedy nutné zahrnout jeden tento soubor pro každý cílový prohlížeč, respektive skupinu prohlížečů.

3.2 Struktura rozšíření s podporou WebExtensions



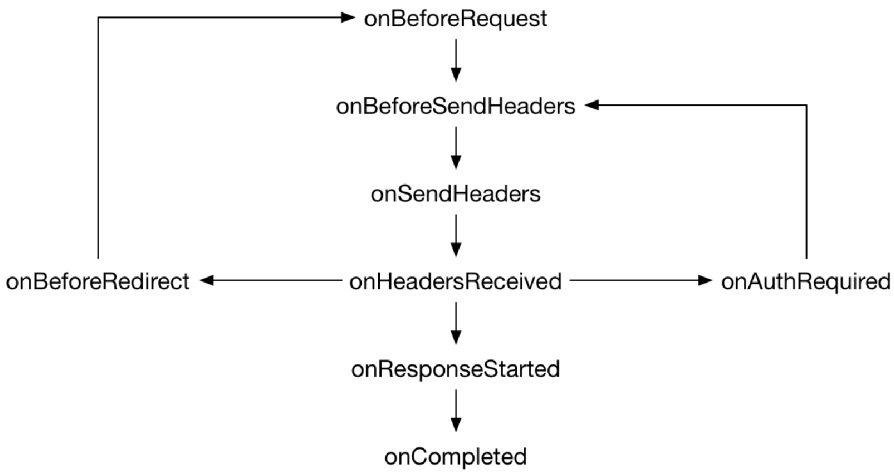
Obrázek 3.1: Struktura WebExtensions rozšíření (převzato z [9]).

Pro lepší pochopení toho, co vlastně tato technologie vývojářům poskytuje, je vhodné si popsat základní strukturu rozšíření vyvinutého s podporou WebExtensions (schéma k vidění na obrázku 3.1) tak, jak ji uvádí autoři dokumentace prohlížeče Firefox [9]. Jak bylo zmíněno v předešlé podkapitole, **manifest.json** je vstupním bodem každého rozšíření. Tento JSON soubor musí povinně obsahovat každé rozšíření. V tomto souboru jsou kromě jména, verze a dalších detailů uvedená požadovaná oprávnění a také cesty k jednotlivým zdrojovým

souborům rozšíření. Dalším prvkem rozšíření jsou skripty běžící na pozadí (angl. *background scripts*), které běží po celou dobu fungování rozšíření – od jeho načtení až po jeho vypnutí, nebo odinstalaci. Tyto skripty obsahují dlouhodobou logiku a vykonávají operace nezávislé na době běhu, konkrétní webové stránce, prohlížeči či okně, nemají však přístup k DOM navštívených webových stránek. Zde je povoleno využívat veškerá API, která WebExtensions poskytují, pro která jsou udělena oprávnění v `manifest.json`. Skripty, které mají přístup k obsahu stránky jsou dalším důležitém prvkem každého rozšíření, v angličtině se označují jako *content scripts*. Tyto skripty jsou načteny do každé webové stránky a mají možnost manipulovat s jejím DOM, jako každý jiný skript uvedený v `<script>` tagu. Navíc mají možnost využívat některá z WebExtensions API a taktéž dokáží komunikovat se skripty běžícími na pozadí. V neposlední řadě by každé rozšíření mělo obsahovat několik HTML a CSS dokumentů, které vytváří uživatelské prostředí. Ať se již jedná o uživatelské prostředí vyskakovacího okna (*browser action*) a nebo samotné stránky nastavení rozšíření (*options page*).

3.3 WebRequest API

WebRequest je jedno z mnoha rozhraní, která poskytuje WebExtensions. Zaměřuje se na zachycení, blokování a modifikaci HTTP požadavků a umožňuje na jednotlivé fáze zpracování požadavku navázat vytvořené funkce, čímž dává vývojáři do rukou nástroj, pomocí kterého lze zjistit detaily o zpracovávaných požadavcích, ale i toto zpracování kontrolovat. Jednotlivé události (angl. *events*) na které je možné navázat funkce jsou k viděni na obrázku 3.2. Pomocí těchto událostí lze kontrolovat zpracování požadavku v každém jednom kroku. Každé funkci, která obsluhuje kteroukoliv z těchto událostí je předán objekt *details*, který obsahuje detailní informace o zpracovávaném požadavku [13]. Na základě zmíněných událostí lze požadavky i modifikovat. Ve fázích *onBeforeRequest*, *onBeforeSendHeaders* a *onAuthRequired* lze požadavek úplně zrušit, ve fázích *onBeforeRequest*, *onHeadersReceived* lze požadavek libovolně přesměrovat a taktéž přistoupit k bezpečnostním informacím, pokud je využito TLS (Transport Layer Security), tyto informace však nelze modifikovat, ve fázi *onBeforeSendHeaders* lze modifikovat hlavičky požadavku a konečně ve fázi *onAuthRequired* lze poskytnout autentizační údaje [13].



Obrázek 3.2: WebRequest události při zpracování HTTP požadavku (převzato z [13]).

Obdobně lze modifikovat i odpovědi na tyto požadavky. Pro práci s odpověďmi WebRequest implicitně poskytuje události *onHeadersReceived* a *onResponseStarted*, v obou fázích jsou však k dispozici pouze hlavičky odpovědi, nikoliv celá odpověď. Odpověď lze zablokovat v první zmiňované fázi, druhá má pouze informativní charakter [13]. Pro zachycení celé odpovědi WebRequest neposkytuje žádnou událost, proto lze sledovat pouze odpověď na konkrétní požadavky. Pomocí funkce *webRequest.filterResponseData()*, které se předává ID odeslaného požadavku, lze vytvořit filtr, pomocí kterého pak WebRequest sleduje odpovědi a pokud narazí na odpověď reagující na požadavek s předaným ID, zachytí ho a zavolá funkci, kterou má navázanou ve vlastnosti *ondata* [13]. V této funkci je možné odpověď modifikovat a následně ji vrátit zpět prohlížeči, který s ní dále pracuje stejně jako by nebyla nikdy upravena.

Kapitola 4

Útok na vnitřní síť s využitím prohlížeče jako proxy

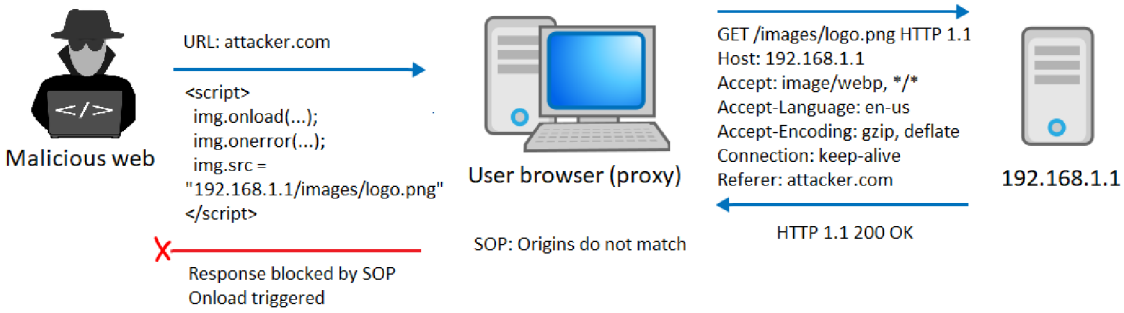
Motivací této práce byl článek [4], popisující útok na privátní síť uživatele z veřejné sítě internet s využitím prohlížeče jako proxy. V článku autoři zmiňují, že se nejedná o žádný nový útok, ale o útok, který není příliš známý a tento článek vznikl jako osvěta pro uživatele a vývojáře, aby se tímto útokem mohli dále zabývat. Na tomto útoku je zajímavé, že pro jeho provedení není nutná žádná konkrétní zranitelnost, nebo slabé místo. Každý krok tohoto útoku spoléhá na to, že jednotlivé věci fungují tak, jak by měly. Navíc obchází jak politiku stejného původu, tak uživatelský firewall. V této kapitole bude popsáno, v čem tento útok spočívá a jakým způsobem ho lze provést.

4.1 Využití prohlížeče jako proxy

Nejprve je nutné vysvětlit, co vlastně znamená výraz *proxy*. Jako *proxy* se v informatice běžně označuje server nebo počítač, který vykonává akce a požadavky na přání uživatele [7]. Slouží tedy jako sluha, nebo prostředník, který přijímá uživatelovi požadavky a vykonává je, jako by byl on sám jejich autorem a odpovědi přeposílá zpět uživatelskému zařízení. *Proxy* se běžně využívá pro zajištění anonymity a obcházení různých omezení (např. webové stránky přístupné jen pro uživatele z konkrétní země) [7]. Útok na vnitřní síť prostřednictvím prohlížeče využívá faktu, že cílový prohlížeč běží na zařízení v určité privátní síti a tím pádem, jako člen této vnitřní sítě, má přístup k ostatním zařízením v této síti. V podkapitole 2.5 byla nastíněna situace, kdy webová stránka požaduje po prohlížeči načtení obrázku z privátní IP adresy, čímž prohlížeč využívá právě jako *proxy*. Prohlížeče tomuto chování běžně nebrání a díky tomu existuje cesta, kterou můžou webové stránky ve veřejné síti posílat požadavky do vnitřní sítě prostřednictvím prohlížeče. Toto potenciálně umožňuje útočníkům rozpoznat zařízení ve vnitřní síti, do určité míry skenovat porty a také zjišťovat jaké služby na daných portech běží [4].

Politika stejného původu je proti tomuto chování rovněž bezmocná, a to hlavně z toho důvodu, že nedokáže zabránit samotnému odeslání požadavku (viz podkapitola 2.2 – zápis napříč původy). To tedy znamená, že zde není žádný ochranný mechanismus, který by zabránil škodlivému JavaScriptu odesílat požadavky prostřednictvím prohlížeče na zařízení ve vnitřní síti se kterými, by za normálních okolností, neměl být schopný komunikovat [4]. Implementace politiky stejného původu sice zabrání JavaScriptu na straně webové stránky, aby si přečetl odpověď (prohlížeč obdrží odpověď, ale nepředá ji dále webové stránce),

nicméně v podkapitole 2.5 bylo ukázáno, že i bez přístupu k odpovědi lze zjistit zda se daná akce úspěšně vykonala či nikoliv. Schematický příklad komunikace mezi webovou stránkou, prohlížečem a zařízením ve vnitřní síti, je k nahlédnutí na obrázku 4.1.



Obrázek 4.1: Schematický příklad využití prohlížeče jako proxy.

4.2 Zjištění IP adresy zařízení a odhadnutí struktury vnitřní sítě

Útok na vnitřní síť prostřednictvím prohlížeče spočívá v umístění škodlivé webové stránky do veřejné sítě internet a navedení uživatele, aby tuto stránku navštívil. Bez toho, aby uživatel navštívil webovou stránku a setrval zde po určitou dobu, není možné tento útok provést. Tuto webovou stránku tvoří klasický HTML dokument, který obsahuje JavaScript kód velmi podobný tomu, který byl prezentován na výpisu 2.1. Jediným rozdílem je, že tento kód nemusí obsahovat pouze jednu IP adresu s cestou k určitému obrázku, ale celé pole se stovkami takových adres, které může tento kód v cyklu testovat. Typicky útočník nepoužije zcela náhodné IP adresy, ale nejprve zjistí privátní IP adresu zařízení, ze kterého uživatel tuto webovou stránku navštívil a na základě této adresy se pokusí co nejlépe odhadnout IP adresy ostatních zařízení ve vnitřní síti [4].

Ke zjištění lokální IP adresy zařízení je možné zneužít WebRTC (Web Real-Time Communication)[1], což je volně dostupné API, podporované všemi rozšířenými prohlížeči. WebRTC poskytuje podporu pro telefonní hovory, streamování videa, sdílení souborů a další [1]. Důležité je zmínit, že WebRTC pro komunikaci využívá peer-to-peer architekturu, kde na rozdíl od standardní architektury klient-server, komunikují uživatelská zařízení přímo mezi sebou. Pro navázání peer-to-peer spojení může být v některých případech nutné znát lokální IP adresu komunikujících zařízení, což nahrává potenciálnímu útočníkovi, který může WebRTC využít pouze k získání této informace [1].

Na základě známé lokální IP adresy lze celkem dobře odhadnout v jakém rozmezí se pravděpodobně budou nacházet IP adresy ostatních zařízení v síti. Typicky útočník využije adresy v blízkosti zjištěné IP adresy, navíc se může pokusit odhadnout například IP adresu routeru, protože routery typicky využívají nejnižší IP adresu v dané podsíti, tedy například pro podsíť `192.168.1.0/24` je vhodné hledat router na adrese `192.168.1.1`, případně lze využít veřejně dostupný seznam výchozích adres routerů pro jednotlivé výrobce[2].

Toto je zásadní část celého útoku, od schopnosti útočníka odhadnout co možná nejvíce IP adres zařízení ve vnitřní síti, se přímo odvíjí úspěšnost útoku. Z toho vyplývá, že lokální

[1] Webová stránka demonstrující toto zneužití: http://net.ipcalf.com/
[2] Například zde: https://www.techspot.com/guides/287-default-router-ip-addresses/

sítě s atypickou strukturou IP adres mohou být méně náchylné k tomuto útoku. Útočník totiž nemůže zkoušet neomezený počet adres, protože je omezen časem, který uživatel stráví na jeho stránce [4]. Ze stejného důvodu rovněž není vhodné v této fázi testovat určité cesty k souborům tak, jak to bylo prezentováno na výpisu 2.1. Tento kód lze zjednodušit, aby stejným způsobem testoval jen IP adresy a konkrétní porty. Cesty ke konkrétním souborům budou testovány až v následujícím kroku a to jen pro ty IP adresy a porty, které dopadnou v tomto kroku pozitivně. Sníží se tím celkový počet odeslaných požadavků a tím pádem i čas potřebný k provedení útoku. Moderní prohlížeče blokují některé běžně známé porty (např. 20 – FTP, 22 – SSH, 25 – SMTP, atd.) a proto je vhodné se při testování těmto portům vyhnout [4]. Jako nejvhodnější se jeví testovat porty, které jsou určitě přístupné. V tomto případě se jedná o HTTP nebo HTTPS porty (80, 443, případně 8080).

Nyní je útočník v situaci, kdy otestoval několik stovek privátních IP adres a portů a zjistil, že některé porty testovaných IP adres jsou skutečně otevřené a tedy na těchto adresách fungují nějaká zařízení. Další fází útoku je rozpoznání o jaké zařízení se jedná a jaké služby běží na otevřených portech [4]. Pro tuto fázi útoku si může útočník připravit seznam cest k obrázkům, které by se mohly s jistou pravděpodobností vyskytovat na aktuálně testovaném zařízení. Využije k tomu opět kód, který byl prezentovaný na výpisu 2.1, který vyžaduje od prohlížeče, aby se pokusil načíst obrázek z konkrétní IP adresy a konkrétního portu s využitím útočníkem podstrčené cesty k tomuto souboru. A protože nic nebrání prohlížeči, aby tuto akci vykonal, odešle požadavek na danou IP adresu a port a očekává, že se mu vrátí požadovaný obrázek. Z důvodu politiky stejného původu prohlížeč ovšem nepředá odpověď, ať už byla jakákoliv, útočníkovi [4]. Útočník však dokáže zjistit, zda byl požadavek vyřízen kladně či záporně (tj. zda se obrázek našel, či nikoliv) a to na základě toho, která z JavaScriptových událostí byla spuštěna (*onload* nebo *onerror*) [4]. Pro každou dvojici – IP adresa a port, lze otestovat libovolný počet cest. Čím víc testovaných cest, tím větší šance, že útočník zjistí o jaké zařízení se jedná.

4.3 Kompromitace vybraného zařízení

Útočníkovi se tedy podařilo zjistit, některá zařízení ve vnitřní síti a nyní se může pokusit některá z nich kompromitovat. Díky tomu, že útočník dokázal určit alespoň výrobce zařízení, může se pokusit využít některá ze známých slabých míst. Navíc může využít toho, že pokud je oběť k této službě přihlášena, tak útočník nemusí znát autentizační údaje a může je nechat doplnit prohlížeč oběti (`request.withCredentials = true`). V článku [4] je popisován útok na Jenkins server, ke kterému je zrovna oběť útoku přihlášena. Autoři článku se tedy dokáží autentizovat prostřednictvím prohlížeče a následně pomocí požadavku odeslaného z prohlížeče oběti dokáží injektovat libovolný kód do skript konzole Jenkins serveru, který jej bez problému provede. Dále je zde zdůrazněno, že se útočník nedozví, zdali byla injekce daného kódu úspěšná a zda se kód vykonal, opět kvůli politice stejného původu. Ovšem i pro tento problém je zde navržené řešení. Útočník si může zřídit svůj DNS server a v kódu, který injektuje na Jenkins server oběti, zahrnout příkaz vyžadující DNS dotaz, který bude vyřízen DNS serverem útočníka. Útočník může následně zkontrolovat logy svého DNS serveru a tím pádem zjistit, zdali byl kód proveden či nikoliv. Obecně se útočník může pokusit kompromitovat libovolné zařízení, které v síti odhalil, zmíněný útok na Jenkins server slouží pouze jako příklad, že toho skutečně lze dosáhnout.

Kapitola 5

Existující rozšíření pro ochranu uživatele před útoky zneužívajícími prohlížeč

V této kapitole budou představeny některé existující rozšíření, které mají potenciál chránit uživatele před útokem popsaným v kapitole 4. Všechny tyto rozšíření jsou komplexními aplikacemi, které se nezaměřují pouze na jediný problém, ale snaží se uživatele chránit proti různým hrozbám, které jsou spojené s HTTP požadavky, načítání skriptů, sledování uživatele, fingerprintingu a tak podobně.

5.1 JavaScript Restrictor

JavaScript Restrictor[1] (dále jen JSR) je, jak uvádí Ing. Martin Timko ve své práci [19], rozšíření pro prohlížeč, které využívá WebExtensions API. Hlavním cílem tohoto rozšíření je zvýšení bezpečnosti, anonymity a ochrany soukromí uživatele při návštěvě různých webových stránek. Některé stránky totiž sbírají informace o uživatelích bez jejich vědomí. Shromážděné informace mohou být využity například ke sledování uživatele, zjištění typu a verze operačního systému, prohlížeče, ale i hardwaru počítače (tzv. *fingerprinting*). Tyto informace dobrovolně poskytuje samotný prohlížeč a uživatel nemá žádnou snadnou možnost kontrolovat jak a komu jsou poskytovány. V mnoha případech může být důvod poskytnutí informace naprosto čestný (například zjištění polohy pro lepší lokalizaci na mapě při hledání restaurace), ale taktéž se dají tyto informace v mnoha případech zneužít ke sledování uživatele. JSR blokuje některá JavaScriptová volání funkcí, která poskytují tyto informace, nebo záměrně poskytuje falešná data, aby byla zachována co možná nejvyšší anonymita uživatele při procházení webových stránek.

V současné verzi JSR blokuje, nebo modifikuje následující volání[2]:

- **window.date** a **window.performance.now** – Obě tyto vlastnosti jsou schopné poskytnout velmi přesné časové údaje, což může vést k lepší identifikaci uživatele. JSR zaokrouhluje oba tyto časové údaje, čímž efektivně podvrhuje tyto informace.
- **HTMLCanvasElement.toDataURL()** – HTML Canvas (plátno pro vykreslování obrázků, tvarů aj.) má přístup k hardwarové akceleraci zařízení, lze tedy pomocí

[1] Oficiální GitHub rozšíření: https://github.com/polcak/jsrestrictor
[2] Detailnější popis v diplomové práci Ing. Martina Timka [19].

určitých metod zjistit informace o zařízení uživatele. JSR na toto volání vrací stejně velký obrázek, jako byl původní, ale všechny pixely přebarví na bílou barvu, čímž snižuje nebezpečí vyzrazení informace.

- **navigator.deviceMemory** a **navigator.hardwareConcurrency** – První vlastnost prozrazuje kapacitu paměti RAM uživatelova zařízení a druhá počet logických jader procesoru. JSR nastavuje obě tyto hodnoty na globálně nejčastější hodnotu.
- **window.XMLHttpRequest** – Tato vlastnost může obsahovat osobní data, které jsou sbírány a odesílány na server po tom, co byla stránka již načtena. V JSR lze všechny tyto požadavky blokovat anebo ponechat volbu na uživateli samotném.

5.2 NoScript

NoScript[3] je volně dostupné prohlížečové rozšíření pro prohlížeče Firefox, Google Chrome a další. Zaměřuje se na blokování aktivního (tj. spustitelného) obsahu webových stránek. Kromě toho, umožňuje rovněž sledovat HTTP požadavky a na základě pravidel vymezit hranice pro každou z navštívených stránek, které definují kam stránka může a nemůže přistupovat a jaké zdroje může požadovat [5]. NoScript pracuje v podobném režimu jako klasický uživatelský firewall, to znamená, že vše, co není explicitně povoleno v seznamu výjimek (*whitelist*) je implicitně zablokováno [6]. Povolovat lze konkrétní webové stránky, konkrétní doménu (například *fit.vutbr.cz*), nebo i celou doménovou hierarchii (například *vutbr.cz*).

Přestože byl NoScript vyvinut s cílem blokovat spouštění nebezpečného JavaScript kódu, NoScript efektivně blokuje i aktivní prvky implementované v Javě, dále například Flash plugin, Silverlight, ale i vložené HTML video/audio prvky [6]. Všechny tyto elementy jsou zablokovány dříve, nežli započne jejich načítání, čímž nejenom chrání uživatele, ale rovněž snižuje využití internetového připojení. Zablokované prvky, které jsou viditelné na výsledné stránce jsou nahrazeny ikonou (angl. *placeholder*) a uživatel má možnost, kliknutím na tuto ikonu, povolit načtení každého prvku zvlášť [6]. Mimo to, jsou jednotlivé zablokované prvky zobrazeny ve vyskakovacím okně rozšíření (viz obrázek 5.1, kde je může uživatel dočasně nebo úplně povolit, anebo naopak zakázat.

Je důležité zmínit, že NoScript nerozhoduje o zablokování prvku pouze na základě jeho původu, ale provádí i kontrolu rodičovské stránky, do které je tento obsah načítán. To znamená, že obsah na stránce, která se nenachází ve *whitelistu*, je blokován i v případě, že pochází z důvěryhodné stránky, tedy takové, která se na *whitelistu* vyskytuje [6]. Toto chování by mělo uživatele chránit proti útoku označovaném jako Flash XSS (*cross-site scripting through Flash*[4]). Kromě Flash XSS, poskytuje NoScript ochranu i proti běžnému XSS (*cross-site scripting*), který spočívá v tom, že se útočníkovi podaří injektovat nebezpečný kód ze své stránky, do jiné, obvykle důvěryhodné, stránky s cílem určitým způsobem poškodit, nebo sledovat uživatele této důvěryhodné stránky [21]. NoScript však na základě porovnání původu injektovaného skriptu dokáže jeho načtení včas zablokovat, čímž efektivně ochrání uživatele před jeho spuštěním.

Kromě výše zmíněné funkcionality, obsahuje NoScript ještě další důležitou funkci, která je označována jako ABE (*Application Boundaries Enforcer*), což je modul zaměřující se

[3] Oficiální GitHub repozitář: https://github.com/hackademix/noscript/

[4] Podrobněji popsán v článku dostupném zde: https://www.acunetix.com/blog/articles/elaborate-ways-exploit-xss-flash-parameter-injection/

Obrázek 5.1: Vyskakovací okno rozšíření NoScript.

na vymezování hranic přístupu dané webové stránky [5]. Tyto hranice jsou definovány na základě pravidel definovaných uživatelem, vývojářem, nebo i důvěryhodnou třetí stranou. ABE na základě těchto definovaných pravidel vyhodnocuje, které operace jsou povoleny a které zakázány vzhledem k uvedenému původu stránky [8]. Například pravidlo pro webovou stránku *example.com*, které povoluje HTTP požadavky typu POST pouze ze stránky s původem *https://secure.example.com*, ale požadavky typu GET bez omezení, má následující tvar:

```
Site example.com
Accept POST from SELF https://secure.example.com
Accept GET
Deny
```

Tato pravidla může upravovat uživatel přímo ve svém nainstalovaném rozšíření, ale kromě toho, poskytuje NoScript i možnost pro vývojáře webových stránek, aby vytvořily soubor s těmito pravidly (přípona `.abe`) a umístili jej do kořenového adresáře své webové stránky [5]. NoScript potom při návštěvě takto připravené stránky načte soubor `.abe` a zaručí vývojářem požadovanou ochranu pro svého uživatele. NoScript je však aktuálně ve stádiu přenosu z původní implementace, do implementace využívající programové rozhraní WebExtensions a proto existují dvě verze – NoScript a NoScript Classic [6].

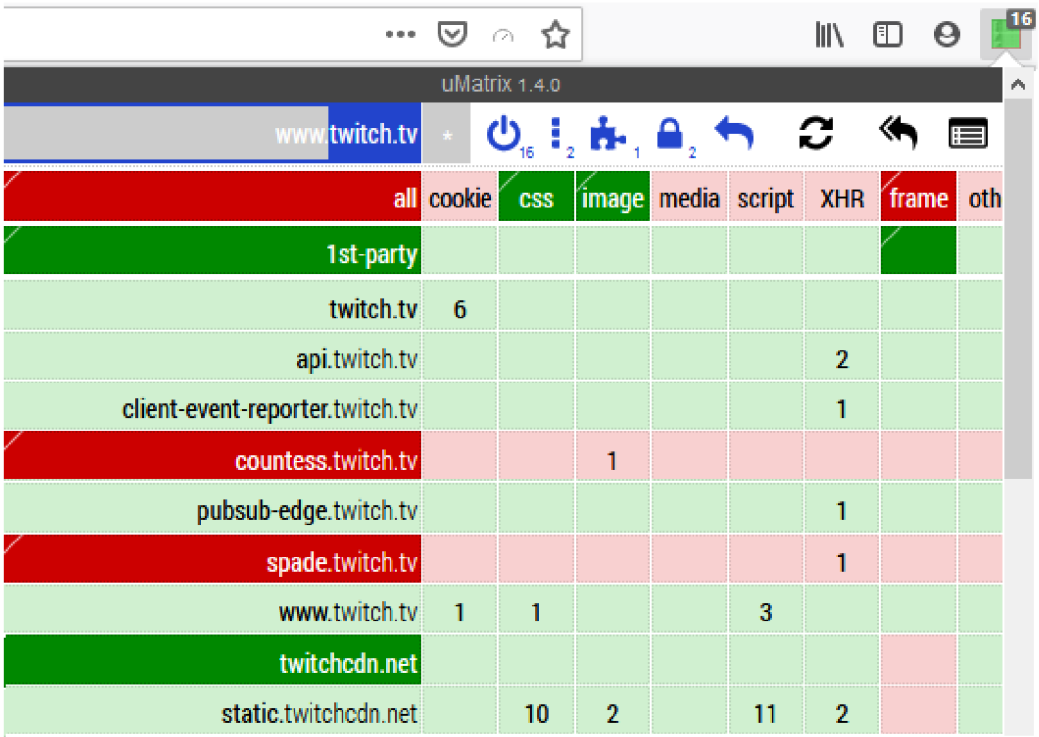
NoScript je verze, která je aktuálně publikovaná v obchodech pro prohlížeče Firefox a Chrome, je to tedy ta verze funugující na WebExtensions a poskytující kompatibilitu napříč všemi prohlížeči. V aktuální verzi ovšem neposkytuje výše zmiňované ABE[5], ostatní funkce jsou však již podporovány. Oproti tomu NoScript Classic je dostupný pouze z webové stránky vývojáře, jedná se o starou implementaci pro Firefox a další menší prohlížeče (Waterfox, Seamonkey a další). Tato verze obsahuje veškerou funkcionalitu včetně ABE, i když se jedná o starší a již nevyvíjenou verzi, které se nedostává tolik pozornosti co se týče například opravování chyb.

[5]Issue na oficiálním GitHub vývojáře: https://github.com/hackademix/noscript/issues/39

5.3 uMatrix

uMatrix[6] je, stejně jako noScript, volně dostupné rozšíření pro prohlížeče Firefox a Chrome. Toto rozšíření umožňuje uživateli sledovat jaké HTTP požadavky jsou prohlížečem odesílány a jaké zdroje jsou načítány [2]. V základním nastavení pracuje ve striktním režimu, kdy je blokováno vše, co není explicitně povoleno a načítání obrázků a CSS souborů je povoleno pouze ze stejného původu [2]. Kromě toho uMatrix rovněž sleduje ukládání cookies a vytváření XHR požadavků (*XMLHttp Requests*[7]). uMatrix je v podstatě velmi sotisfikovaný a nastavitelný filtr pro rozličné aktivity na internetu, jeho nastavení ovšem není úplně jednoduché a striktní režim poškozuje vysoké množství navštívených stránek. Naštěstí lze zablokované zdroje jednoduše povolit. uMatrix umožňuje sledovat následující zdroje:

- Cookies
- Soubory kaskádových stylů (`.css`)
- Média – obsah HTML tagů `<audio>`, `<video>`, `<object>`, `<embed>`
- Skripty
- XMLHttp požadavky
- Vkládání jiných webových stránek pomocí `<frame>` nebo `<iframe>`
- Ostatní – vše, vymykající se výše popsanému



Obrázek 5.2: Vyskakovací okno rozšíření uMatrix.

[6]Oficiální GitHub repozitář: https://github.com/gorhill/uMatrix
[7]Vysvětleno zde: https://en.wikipedia.org/wiki/XMLHttpRequest

uMatrix odvozuje své jméno od matice (angl. *matrix*), pomocí které se toto rozšíření ovládá. Tato matice (zobrazena na obrázku 5.2) umožňuje blokovat a povolovat jednotlivé zdroje (ve sloupcích) nebo jednotlivé hostitele (v řádcích), nebo každou buňku matice zvlášť [18]. Blokování je založeno na podobném principu jako v případě NoScript. Pokud uživatel navštíví stránku s doménou *example.com*, všechny zdroje pocházející z této domény budou implicitně povoleny, ale všechny ostatní zablokovány. Uživatel může samozřejmě toto chování plně ovlivnit v matici, která je přítomna ve vyskakovacím okně rozšíření.

uMatrix tedy s vhodným nastavením rovněž poskytuje ochranu před XSS útoky, neboť podobně jako NoScript dokáže identifikovat původ injektovaného skriptu a pokud se nejedná o původ, který je aktuálně povolen, je načtení tohoto skriptu zablokováno. uMatrix rovněž poskytuje ochranu před sledováním uživatelů pomocí různých trackerů a podobně. Na stránce internetového obchodu Chrome tohoto rozšíření[8], dokonce autoři zmiňují, že je možné nastavit uMatrix například tak, aby povoloval stránku *facebook.com* jen tehdy, pokud se uživatel skutečně na této stránce nachází, ale blokoval ji na všech jiných stránkách. Toto nastavení zabrání stránce *facebook* sledovat a shromažďovat informace o pohybu uživatele na internetu.

[8] Odkaz na stránku obchodu: https://chrome.google.com/webstore/detail/umatrix/ogfcmafjalglgifnmanfmnieipoejdcf

Kapitola 6

Návrh řešení

Cílem této práce je najít způsob detekce a ochrany proti využití prohlížeče jako *proxy* pro kompromitaci vnitřní sítě uživatele. Ideálním místem pro implementaci detekce i ochrany je prohlížeč samotný a protože prohlížeče samotné tomuto útoku nemohou nijak zamezit, jsme nutně odkázáni na využití prohlížečových rozšíření, které byly popsány v kapitole 3. Z tohoto důvodu bude výsledek této práce implementován jako dodatečné rozšíření již existujícího prohlížečového rozšíření JavaScript Restrictor (viz podkapitola 5.1), které se rovněž zaměřuje na bezpečnost a anonymitu uživatele při prohlížení stránek ve veřejné síti internet a je tedy ideálním místem pro integraci výsledků této práce. V této kapitole bude postupně představen způsob detekce a ochrany proti výše popsanému útoku, nejprve se zaměříme na identifikaci hlavního důvodu zranitelnosti, následně na samotné zachycení požadavků a jejich klasifikaci a na závěr bude představen návrh vyskakovacího okna upozorňujícího uživatele na potenciálně nebezpečné požadavky.

6.1 Identifikace hlavního důvodu zranitelnosti

Útok popsaný v kapitole 4 zásadně spoléhá na schopnost donutit prohlížeč odesílat požadavky na zařízení ve vnitřní síti. Pro provedení útoku není důležité, zdali útočník obdrží odpověď na tyto požadavky a z toho důvodu je celý mechanismus politiky stejného původu do jisté míry bezmocný. Samozřejmě, kdyby politika stejného původu nebyla implementována vůbec, bylo by provedení útoku o to jednodušší, ale ani plně striktní politika stejného původu bez jakékoliv možnosti jejího zmírnění, například pomocí *CORS*, útočníkovi v provedení útoku nezabrání. Hlavním cílem bude tedy detekovat chování prohlížeče, které by vedlo k odeslání požadavků, pocházejících z veřejné sítě internet, na lokální IP adresy. Ve chvíli, kdy se podaří zamezit prohlížeči, aby takovéto požadavky odesílal do vnitřní sítě, ztratí tím útočník zásadní nástroj pro provedení útoku a útok se tím pádem stane neproveditelným.

Ideálním prostředkem pro zachycení požadavku je WebRequest API (viz podkapitola 3.3), které umožňuje zachytit požadavky v různých fázích zpracování. Jako nejzajímavější se jeví fáze *onBeforeRequest* a *onBeforeSendHeaders*, které nám obě umožní pracovat s požadavkem dříve, než jej prohlížeč odešle. Fáze *OnBeforeRequest* je úplně první fází ve zpracování požadavku, kdy ještě prohlížeč nezná veškeré detaily o zachyceném požadavku. Zejména nejsou k dispozici všechny hlavičky a z toho důvodu je pro navržené řešení vhodnější využít fázi *onBeforeSendHeaders*, která těsně předchází samotnému odeslání hlaviček a následně i samotného požadavku. Ve fázi *onBeforeSendHeaders* jsou tedy k dispozici

již veškeré informace o požadavku, včetně HTTP hlaviček. Data z HTTP hlaviček jsou pro navržené řešení obzvlášť důležitá, neboť obsahují informace o původci a cíli požadavku. Veškeré dostupné informace o požadavku jsou předány z WebRequest API ve formě objektu reprezentujícího zachycený požadavek. Ze všech dostupných vlastností tohoto objektu jsou pro navržené řešení relevantní zejména tyto:

- `request.originUrl` – URL serveru ze kterého požadavek pochází,
- `request.url` – URL zdroje, který má prohlížeč požadovat, jinými slovy – cíl požadavku,
- `reuqest.type` – obsahuje informaci o jaký typ zdroje se jedná (skript, obrázek, css dokument, atd.),
- `request.requestId` – ID požadavku (unikátní v rámci sezení prohlížeče).

6.2 Klasifikace požadavku

Z informací předaných od WebRequest API lze zjistit, zdali se jedná o potenciálně nebezpečný požadavek, který směřuje z veřejné sítě do privátní, či nikoliv. Z obou URL lze extrahovat doménová jména a ta následně přeložit pomocí DNS dotazu na IP adresu. Překlad doménového jména na IP adresu lze provést pomocí *dns* API, které je rovněž součástí WebExtensions a obsahuje jedinou funkci s názvem `resolve`, která odešle DNS dotaz na překlad doménového jména předaného v parametru této funkce [11]. Získané IP adresy pak lze klasifikovat podle prvních dvou oktetů jako privátní (mezi privátní IP adresy patří například následující – *10.*.*.*, 192.168.*.*, 172.16-31.*.**), nebo veřejné, obdobným způsobem lze samozřejmě klasifikovat i adresy IPv6. V našem případě nás budou obzvlášť zajímat takové požadavky, které přichází z veřejných IP adres, ale směřují na adresy privátní. Takové požadavky znamenají potenciální hrozbu a protože ve většině případů neexistuje žádný relevantní důvod k tomu, aby server z veřejné IP adresy požadoval nějaké zdroje z privátních adres, můžeme tyto požadavky klasifikovat jako nebezpečné.

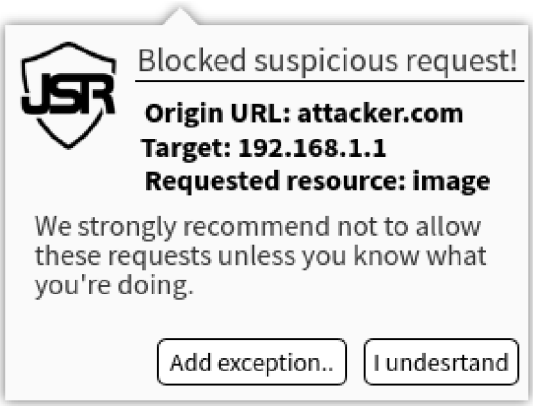
Na základě informací o kompatibilitě shrnutých v podkapitole 3.1, je evidentní, že tento přístup bude fungovat pouze v prohlížeči Firefox. Opera ani Chrome (tj. další prohlížeče, pro které je v současné době publikován JavaScript Restrictor) nenabízí žádné API umožňující práci s DNS dotazy. V tomto případě je třeba pozměnit návrh řešení tak, aby se dal využít i v těchto prohlížečích. Jednou z možností je spolehnutí se na fakt, že útočník nemůže dost dobře odhadnout doménová jména zařízení ve vnitřní síti a proto bude ve většině případů zkoušet přímo IP adresy, které lze rovnou klasifikovat jako veřejné, nebo privátní bez potřeby DNS dotazů. Tento přístup má však minimálně jednu nevýhodu. Některá zařízení ve vnitřní síti mohou mít odhadnutelné doménové jméno, v článku [4] byl prezentován příklad, kdy útočník využil znalosti domény společnosti (např.: *example.com*) a testoval některé subdomény (např.: *printer.example.com*, *wiki.example.com*). V takovém případě by v prohlížečích Chrome a Opera nedošlo k rozpoznání a klasifikaci nebezpečného požadavku. Zde se pak nabízí jiný přístup, z popisu útoku v kapitole 4 je zřejmé, že bude docházet k odesílání velkého množství HTTP požadavků v krátkém časovém intervalu a na většinu z nich budou přicházet negativní odpovědi. Lze tedy stanovit počet požadavků z jedné zdrojové URL s negativními odpověďmi a pokud tohoto počtu některá z uživatelem navštívených URL dosáhne, implicitně klasifikovat všechny požadavky z této URL jako nebezpečné.

6.3 Nebezpečné požadavky

Zde se nabízí několik možností jak naložit s požadavky, které budou klasifikovány jako nebezpečné. První z možností je nechat uživatele rozhodnout jak má být naloženo s každým nebezpečným požadavkem, což dává uživateli plnou kontrolu nad tímto chováním za cenu toho, že bude muset opakovaně povolovat, nebo zamítat nějaké vyskakovací okno. Tato možnost s sebou ovšem nese i riziko, že uživatel nemusí vždy přesně vědět o co se jedná a mohl by nevědomě některé z těchto požadavků povolit. Stojí tedy za zvážení, jestli není bezpečnější striktně blokovat všechny nebezpečné požadavky a jen upozornit uživatele, že k něčemu takovému došlo. Toto chování by se ovšem mohlo ukázat jako omezující pro uživatele, který by mohl chtít v některých krajních případech takové požadavky povolit, protože by si byli jisti, že nejsou nebezpečné. Jako nejlepší volba se tedy jeví kompromis těchto dvou řešení. Nebezpečné požadavky implicitně blokovat, ale dát uživateli možnost toto blokování pozastavit například pro vybrané URL.

6.4 Návrh vyskakovacího okna

Pokud tedy dojde k zablokování požadavku, bude uživatel upozorněn formou vyskakovacího okna, které ponese základní informace o blokovaném požadavku a taktéž možnost přidání výjimky pro danou URL. Vyskakovací okno bude rozděleno na tři části. V záhlaví bude umístněno logo JSR společně s nadpisem tohoto vyskakovacího okna, ve střední části okna budou zmiňované informace o zablokovaném požadavku: zdrojová URL, cíl a o jaký typ zdroje se jedná. Ve spodní části bude potom umístěno textové upozornění vyzývající uživatele, aby nepovoloval požadavky u kterých si není jistý zda jsou nebezpečné či nikoliv, a samotná tlačítka pro uzavření okna a přidání výjimky pro danou URL. Návrh popisovaného vyskakovacího okna lze vidět na obrázku 6.1.



Obrázek 6.1: Vyskakovací okno upozorňující uživatele na podezřelé požadavky.

Kapitola 7

Implementace

V této kapitole bude podrobně popsán a vysvětlen způsob implementace navrhnutého řešení. Nejprve bude představena struktura řešení jako celku a bude nastíněno jakým způsobem fungují jeho jednotlivé části. Následně budou tyto části do detailů vysvětleny, Bude popsáno jakým způsobem se zachytávají a klasifikují HTTP požadavky, jakých nástrojů a funkcí se k tomu využívá a i to, jakým způsobem to ovlivní navštívené webové stránky. Rovněž zde budou uvedeny některé z problémů, které se objevily až v průběhu implementace a kvůli kterým bylo nutné v určitých ohledech pozměnit návrh řešení. Při implementaci se také ukázalo, že pro lepší přehlednost bude nutné implementaci rozdělit do dvou verzí a to do verze pro prohlížeč Firefox, ve kterém je k dispozici DNS API a do verze pro prohlížeč Chrome, ve kterém toto API k dispozici není. Z tohoto důvodu budou tyto dvě verze popisovány odděleně.

7.1 Integrace s JavaScript Restrictorem

Implementované řešení bylo od počátku míněno jako rozšíření JavaScript Restrictoru a i když bylo vyvíjeno odděleně (*standalone* verze[1] se rovněž nachází na přiloženém paměťovém médiu), bude výsledek integrován v rámci prohlížečového rozšíření JavaScript Restrictor. Z toho důvodu zde nebudou popisovány soubory a implementační detaily, které se nachází v původním JavaScript Restrictoru, ale pouze ty, které chování JavaScript Restrictoru rozšiřují. Implementované řešení se skládá z několika souborů, které jsou buď převzaty z původního JavaScript Restrictoru a více či méně rozšířeny o novou funkcionalitu, nebo nově vytvořeny. Jedná se o následující soubory:

- `manifest.json` – obsahuje potřebná metadata, aby prohlížeč dokázal rozšíření v pořádku načíst, zde se, o proti původnímu JSR, změnily zejména povolení, která rozšíření musí vyžadovat od uživatele k přístupu k určitým API (viz podsekce 7.1.1).
- `http_shield_firefox.js` – soubor obsahující implementační logiku blokování HTTP požadavků specifickou pro prohlížeč Firefox, soubor byl nově vytvořen a přidán do JavaScript Restrictoru.
- `http_shield_chrome.js` – soubor obsahující implementační logiku blokování HTTP požadavků specifickou pro prohlížeč Chrome, soubor byl nově vytvořen a přidán do JavaScript Restrictoru.

[1] Standalone verze rovněž zde: https://github.com/IanNov/http-webRequest-shield

- `http_shield_common.js` – soubor obsahující podpůrné funkce, společné pro implementaci obou verzí navrženého řešení, soubor byl rovněž nově vytvořen a přidán do JavaScript Restrictoru.
- `options.html` a `options.css` – webová stránka nastavení všech funkcí rozšíření, byla rozšířena o zapnutí či vypnutí blokování požadavků a nastavení výjimek.
- `options.js` – obsahuje implementační logiku podporující stránku nastavení, soubor byl rozšířen o příslušné funkce týkající se zapínání a vypínání blokování požadavků, nastavování výjimek a komunikaci s `http_shield_*.js` (`http_shield_firefox` nebo `http_shield_chrome`).
- `popup.html` a `popup.css` – obsahuje definici uživatelského rozhraní vyskakovacího okna rozšíření, soubory byly rozšířeny o přepínač umožňující přidání a odebrání navštívené stránky do a z výjimek.
- `popup.js` – obsahuje implementační logiku podporující vyskakovací okno rozšíření, soubor byl rozšířen o příslušné funkce týkající se přidávání a odebírání aktuálně navštívené stránky do a z výjimek.

Všechny ostatní soubory JSR byly ponechány beze změny, neboť nebyly potřebné pro implementaci nové funkcionality. Jak bylo předesláno v úvodu této kapitoly, implementace byla nakonec rozdělena do dvou verzí a to z toho důvodu, že DNS API, dostupné ve Firefox, je natolik zásadní, že by byla škoda jej nevyužít i za cenu toho, že implementace potom není kompatibilní s prohlížečem Chrome. Na druhou stranu, řešení pro Chrome stojí pouze na analýze jednotlivých požadavků a statistickém vyhodnocování počtu úspěšných a neúspěšných odpovědí na tyto požadavky. Tato verze je samozřejmě použitelná i ve Firefox, nicméně, na základě testů provedených v podkapitole 8.1, dosahuje horších výsledků než verze využívající DNS API.

7.1.1 Manifest rozšíření

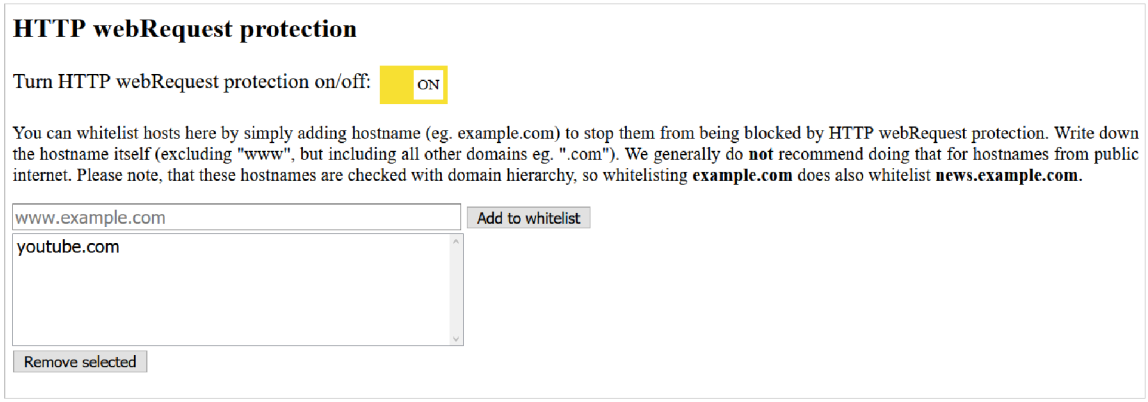
Soubor `manifest.json` obsahuje metadata, které využívá prohlížeč pro lokalizaci veškerých částí rozšíření a k jejich následnému načtení. Jsou zde rovněž uvedeny povolení k přístupu ke specifickým API z WebExtensions. Některá API je totiž nutné explicitně povolit, v implementovaném řešení se jedná zejména o povolení k přístupu pro *storage*, *webRequest*, *dns* a *notifications*.

- *Storage* – povoluje přístup k úložisti prohlížeče, které je synchronizováno napříč zařízeními spojenými s uživatelským účtem (`browser.storage.sync`), jedná se o persistentní úložiště, do nějž lze ukládat data potřebná pro chod rozšíření. V implementovaném řešení je využito pro uložení seznamu doménových jmen, pro které nemá být ochrana aktivní.
- *WebRequest* – povoluje přístup k API, které slouží k zachytávání HTTP požadavků. Toto API je základem pro implementované řešení, neboť umožňuje zachytit a povolit či zablokovat konkrétní požadavky.
- *Dns* – nachází se pouze v manifestu pro Firefox a povoluje přístup k API, které umožňuje překlad doménových jmen na IP adresy pomocí DNS dotazů. V implementovaném řešení je využito pro identifikaci a klasifikaci zachycených požadavků.

- *Notifications* – povoluje přístup k využití notifikací, které mají uživatele upozornit na nějakou událost. V implementovaném rozšíření je toto API využito pro upozornění uživatele na zablokování požadavku.

7.1.2 Stránka nastavení

Soubory `options.css` a `options.html` obsahují veškerý front-end, které implementované řešení využívá. Uživatelské prostředí stránky s nastavením bylo rozšířeno pouze o přepínač ON/OFF a o seznam domén, pro které nemá probíhat blokování (angl. *whitelist*, viz obrázek 7.1). Přepínač ON/OFF vypíná, respektive zapíná celou funkcionalitu. Toto je implementováno v souboru `options.js`, kde po přepnutí přepínače dochází k odeslání zprávy do `http_shield_*.js`. Na základě této zprávy je buď připojena příslušná funkce k události zachycení požadavku, nebo je naopak odpojena a zachytávání požadavků neprobíhá. Podobným způsobem probíhá i předávání seznamu výjimek, vždy, když uživatel vloží novou výjimku do seznamu domén, nebo některou z nich naopak odstraní, dochází k uložení celého seznamu do `browser.storage.sync` a k odeslání zprávy do `http_shield_*.js`, že došlo k aktualizaci tohoto seznamu. Skript běžící v `http_shield_*.js` v reakci na tuto zprávu aktualizuje svůj seznam výjimek opět z `browser.storage.sync`.



Obrázek 7.1: Přidané elementy ve stránce s nastavením.

7.2 Implementace pro prohlížeč Firefox

Implementace se mezi verzemi pro Firefox a Chrome příliš neliší, ale rozdělení na dvě verze odstranilo několik podmínek, které kontrolovaly, zda rozšíření běží v prohlížeči Firefox či nikoliv. Základem implementace pro Firefox je asynchronní funkce *beforeSendHeadersListener*, která se navazuje na WebRequest API podle toho, zda je kontrola a blokování požadavků zapnutá či nikoliv. WebRequest API následně této funkci, pro každý zachycený požadavek, předává objekt, který ho reprezentuje. V návrhu byly zvažovány dvě události – *onBeforeSendHeaders* a *onBeforeRequest*, ze kterých byla pro implementaci na konec zvolena událost *onBeforeSendHeaders* a to z toho důvodu, že některé informace z HTTP hlaviček nebyly ve fázi *onBeforeRequest* k dispozici.

7.2.1 Zpracování požadavku

Hned v prvním kroku implementace je z předaného objektu požadavku získána jeho zdrojová (vlastnost *originUrl*) a cílová URL (vlastnost *url*). Zdrojová URL je shodná s URL webové stránky, která je původcem tohoto požadavku. Pokud požadavek vznikl například ve vloženém HTML tagu `iframe`, ve kterém byla načtena odlišná webová stránka, potom je zdrojová URL shodná s URL webové stránky v `iframe`. Ihned po získání obou URL, je zdrojová URL vyhledána v asociativním poli výjimek (v implementaci označeno jako `doNotBlockHosts`) a pokud je nalezena, je požadavek ihned povolen a funkce ukončena. Asociativní pole výjimek je aktualizováno na základě uživatelského vstupu skrze stránku s nastavením a vyskakovací okno rozšíření. Pokud zdrojová URL nebyla nalezena mezi výjimkami, je požadavek postoupen do další úrovně zpracování.

Zde se zdrojová a cílová URL kontrolují pomocí regulárních výrazů. Nejprve je ověřeno zda URL neodpovídá regulárnímu výrazu pro IPv4 adresu, pokud ano, je tato URL ihned postoupena ke klasifikaci, v opačném případě je ještě ověřen regulární výraz pro adresu typu IPv6. Pokud ani zde nenastane shoda, je jasné, že se v URL nenachází IP adresa v číselném formátu a je nutné přistoupit k DNS dotazu. Překlad doménového jména na IP adresu obstarává DNS API přístupné skrze `browser.dns.resolve()`. Po získání zdrojové a cílové IP adresy následuje samotná klasifikace.

7.2.2 Klasifikace požadavku

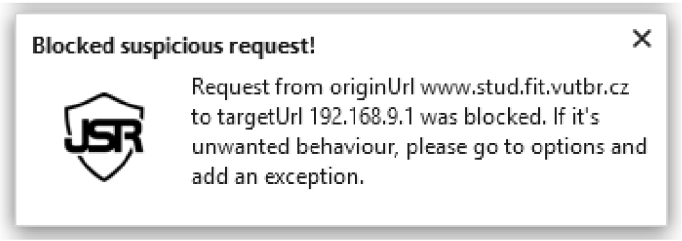
Získané IP adresy jsou nejdříve klasifikovány jako veřejné, nebo privátní. Pro tuto klasifikaci je využito CSV souborů vytvořených organizací **IANA**[2], které jsou staženy a připojeny k rozšíření při jeho překladu a sestavení. V těchto dvou CSV souborech (pro IPv4 a IPv6) se nachází všechny aktuálně platné rozsahy privátních IP adres. Tyto přibalené CSV soubory jsou lokálně zpracovány a načteny do pole při každém spuštění prohlížeče s aktivním rozšířením. Následně jsou v tomto poli udržovány po celou dobu běhu rozšíření. Získané IP adresy jsou tedy postupně kontrolovány oproti privátním rozsahům v tomto poli, pokud se u některého z rozsahů prokáže shoda, je tato IP adresa klasifikována jako privátní, pokud se u žádného shoda neprokáže potom jako veřejná. Potom, co je známa třída obou IP adres, je požadavek klasifikován na základě toho, zda pochází z veřejné IP adresy a míří na IP adresu privátní, či nikoliv. Pokud tomu tak je, požadavek je klasifikován jako nebezpečný a zablokován. V opačném případě je požadavek povolen a zpracování končí.

7.2.3 Notifikace uživatele

Pokud je požadavek zablokován, dojde k upozornění uživatele. V návrhu se počítalo s tím, že bude pro upozornění uživatele využito vyskakovacího okna, to se ovšem ukázalo jako problematické. Vyskakovací okno totiž nelze programově vyvolat bez předcházející uživatelské akce. To ovšem vůbec neodpovídá potřebám implementované funkcionality, kde je potřeba uživatele viditelně notifikovat vždy, když dojde k zablokování požadavku, a proto od něj jen těžko lze vyžadovat nějakou akci, která by umožnila aktivaci vyskakovacího okna. Zvolená alternativa využívá dalšího API z WebExtensions, které nese název *notifications*. *Notifications* dokáže vytvořit notifikaci (viz obrázek 7.2) velmi podobnou těm, které se běžně vyskytují v operačních systémech Windows. Notifikace se zobrazí v pravém dolním rohu obrazovky a nese logo JavaScript Restrictoru a informace o zablokovaném požadavku.

[2]Dostupné z: https://www.iana.org/assignments/locally-served-dns-zones/locally-served-dns-zones.xml

Konkrétně se jedná o zdrojovou a cílovou URL a informaci o tom, že pokud je toto chování nechtěné, lze jej omezit ve stránce s nastavením.



Obrázek 7.2: Notifikační zpráva upozorňující uživatele na zablokování požadavku.

7.3 Implementace pro prohlížeč Chrome

Prohlížeč Chrome nedisponuje DNS API, není tedy možné selektivně blokovat, nebo povolovat jednotlivé požadavky. Z tohoto důvodu pracuje verze pro prohlížeč Chrome se samotnými původci požadavků, umožňuje tedy blokovat všechny HTTP požadavky od původců, které budou vyhodnoceni jako nebezpeční. Pro vyhodnocování, zdali je daný původce nebezpečný či nikoliv, již nepostačují pouze informace získané z WebRequest události *onBeforeSendHeadersListener*, jako tomu bylo ve Firefox, ale je nutné zpracovávat i informace dostupné v HTTP hlavičkách odpovědí na požadavky.

Toto umožňuje WebRequest událost *onHeadersReceived*, která je spuštěna po přijetí HTTP hlaviček odpovědi a předává do navázané funkce objekt obsahující příslušná pole z těchto HTTP hlaviček. Zachycené HTTP hlavičky odpovědí poskytují, mimo jiné, informace o tom, zdali požadavek skončil úspěchem, nebo chybou. Nesou totiž hodnotu, která se běžně označuje jako *status code* na základě které lze poznat výsledek zpracování odpovídajícího požadavku (mezi nejznámější kódy patří například: kód 200 - OK nebo kód 404 - Not found) [17]. Tyto status kódy jsou v implementovaném řešení dále využity pro analýzu jednotlivých původců požadavků.

Kromě události *onHeadersReceived* se v implementaci pracuje ještě s událostí *onErrorOccured*, která se spouští ve chvíli, kdy WebRequest zachytí některou z chyb. Nejčastěji se jedná o chybu typu „*doba žádosti vypršela*“ (angl. *request timed out*), která je odeslána ve chvíli, kdy odpověď na daný požadavek nepřišla v daném časovém intervalu. Implementace pro prohlížeč Chrome se proto skládá ze tří hlavních částí, které tvoří právě funkce navázané na zde popsané události.

7.3.1 Zpracování požadavku

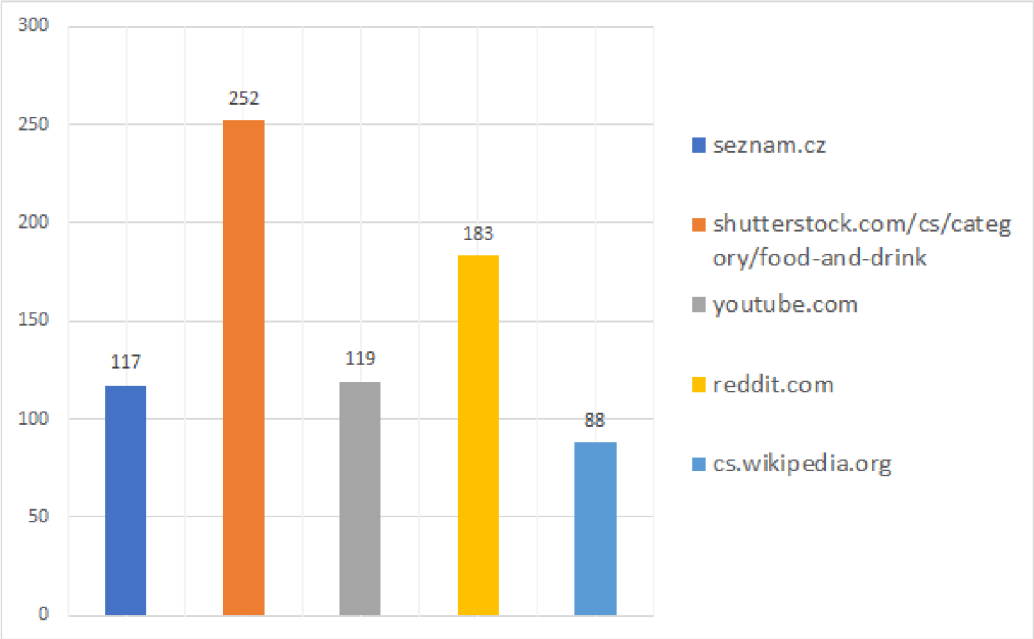
Obdobně jako ve verzi pro prohlížeč Firefox, i zde se nejprve získává zdrojová a cílová URL, rozdíl je pouze v názvu vlastnosti pro zdrojovou URL, ta je zde označena jako *initiator*. Opět se zde kontroluje nejprve přítomnost zdrojové URL v poli výjimek `doNotBlockHosts`. Kromě tohoto pole jsou v této verzi implementována ještě další dvě asociativní pole, jedná se o pole `blockedHosts`, které obsahuje již zablokované původce, a pole `hostStatistics`, které slouží pro ukládání statistických informací o jednotlivých původcích. Ihned po kontrole výjimek následuje ještě kontrola přítomnosti zdrojové URL v poli blokovaných původců `blockedHosts`. Pokud je zde zdrojová URL nalezena, je požadavek zablokován a zpracování končí, v opačném případě je požadavek postoupen do další fáze zpracování.

V další fázi zpracování dochází ke kontrole zdrojové a cílové IP adresy pomocí regulárních výrazů s cílem zjistit, zdali se jedná o IPv4, IPv6 adresu, nebo doménové jméno. Pokud se ukáže, že jsou obě URL IP adresami, zpracování pokračuje, obdobně jako ve verzi pro prohlížeč Firefox, klasifikací požadavku a jeho povolením, nebo zablokováním. Zajímavější je situace, kdy je alespoň jedna z URL skutečně doménovým jménem. V takovém případě mohou nastat následující tři scénáře:

1. Zdrojová URL je doménovým jménem a cílová IP adresou – cílová IP adresa je klasifikována jako veřejná, nebo privátní. Pokud požadavek směřuje na privátní IP adresu, tak je zablokován. V případě, že se jedná o veřejnou IP adresu, je povolen. Chování rozšíření je v tomto ohledu velmi striktní, tedy pokud nelze rozhodnout, zda je zdroj ve veřejné, nebo lokální síti a požadavek směřuje na privátní IP adresu, je zablokován.

2. Obě URL jsou doménovými jmény – v tomto případě nelze rozlišit požadavky mířící na privátní IP adresy od těch, které míří na IP adresy veřejné, je tedy nutné sledovat požadavky jednotlivých původů a shromažďovat o nich určité informace, na jejichž základě lze následně identifikovat potenciálně nebezpečné původce.

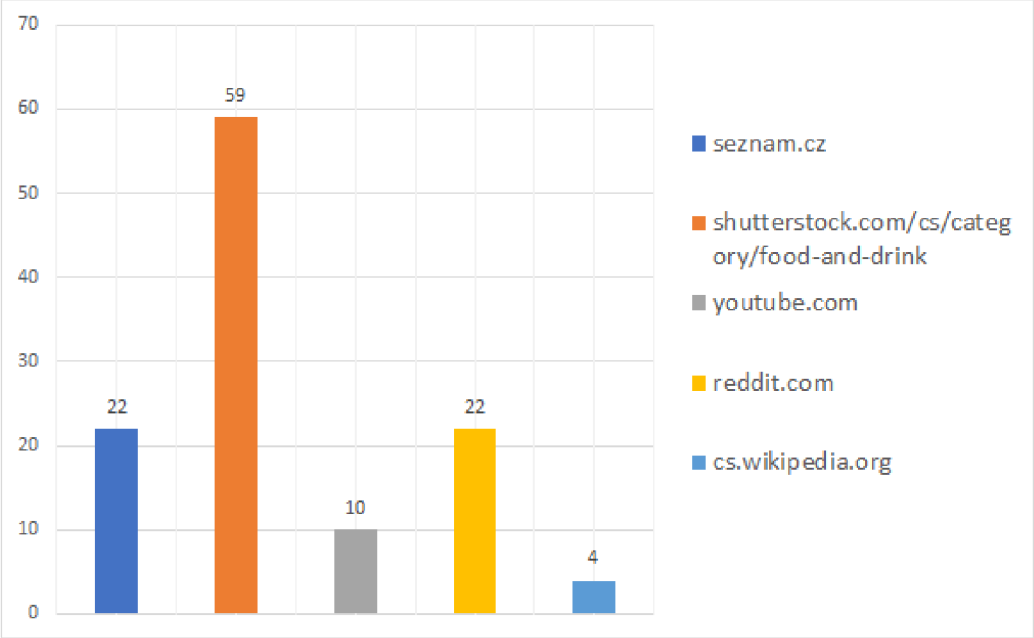
3. Zdrojová URL je IP adresou a cílová doménovým jménem – v tomto případě lze sice zjistit, zdali je zdrojová IP adresa veřejná či privátní, tato informace je však nedostatečná, neboť toto nelze říci o cílové URL. Z tohoto důvodu se v tomto případě rozšíření chová stejně, jako kdyby byly obě URL doménovými jmény, a tedy platí scénář označen číslem dva.



Obrázek 7.3: Graf udávající počet požadavků při načtení jednotlivých webových stránek (uvedená čísla jsou ilustrační, byla získána při jedné z návštěv daných webových stránek).

7.3.2 Identifikace nebezpečných původců

Návrh počítal s tím, že pro identifikaci nebezpečných původců bude dostatečné sledovat, pro každý původ, počet požadavků za určitou jednotku času. Nicméně se ukázalo, že i důvěryhodné webové stránky, při svém načítání, odesílají vysoké počty požadavků (viz graf na obrázku 7.3) a na základě této statistiky nelze nebezpečné původce odlišit.



Obrázek 7.4: Graf udávající počet unikátních cílů požadavků při načtení jednotlivých webových stránek (uvedená čísla jsou ilustrační, byla získána při jedné z návštěv daných webových stránek, při dalších návštěvách se mohou lišit).

Tato statistika byla tedy rozšířena o další metriku, kterou je počet různých cílových doménových jmen, na které daní původci odesílali své požadavky (viz graf na obrázku 7.4). Ani tato metrika se však neukázala být zcela prokazatelná, je sice o něco lepší nežli ta předchozí, nicméně z grafu je patrné, že i důvěryhodné webové stránky přistupují na relativně vysoký počet cílových doménových jmen. Původní myšlenka této metriky je však správná, neboť útočník, který nezná vnitřní strukturu sítě, na kterou se snaží zaútočit, bude velmi pravděpodobně potřebovat odesílat požadavky na vysoký počet cílových URL, aby v síti odhalil některá zařízení. Problém je však v tom, jak rozlišit požadavky legitimních původců od útočníkových. V tom by mohla pomoci další metrika, která by měřila počet požadavků, které skončí chybovou odpovědí, zvlášť pro každou cílovou doménu.

Důvěryhodné webové stránky totiž zcela určitě vědí, kde se nachází zdroje, které požadují a z toho důvodu bude docházet jen k velmi nízké chybovosti. Na druhou stranu, útočník, který nemá informace o dostupných zdrojích a struktuře vnitřní sítě, bude zkoušet a doufat, že něco odhalí, což vyústí ve zvýšený počet HTTP chyb v odpovědích na některé požadavky. Vyhodnocování bude probíhat tedy na základě toho, na kolika unikátních cílových doménách došlo k chybové odpovědi, pokud tedy například některá webová stránka požaduje zdroje z pěti unikátních cílových domén a ve čtyřech případech obdrží v odpovědi HTTP chybu, je to velmi podezřelé.

- `400 - Bad Request`
- `404 - Not Found`
- `405 - Method Not Allowed`
- `406 - Not Acceptable`
- `408 - Request Timeout`
- `410 - Gone`
- `413 - Payload Too Large`
- `414 - URI Too Long`
- `415 - Unsupported Media Type`
- `501 - Not Implemented`
- `503 - Service Unavailable`
- `505 - HTTP Version Not Supported`

Výpis 7.5: Seznam HTTP chyb, které jsou v rozšíření přičítány jako nebezpečné.

Na druhou stranu, pokud nastane situace, kdy z deseti cílových domén obdrží jen u jedné HTTP chybu, lze předpokládat, že se jedná o legitimní stránku, na které došlo pouze k nějaké chybě nebo výpadku. Rovněž třeba brát v potaz i to, zda jsou pro jednu cílovou doménu zachyceny i úspěšné odpovědi (tj. status kód například `200 - OK`), nebo pouze chyby. V prvním případě lze předpokládat, že se jedná o důvěryhodnou stránku, kde občas dojde k nějaké chybě, v druhém případě je velmi pravděpodobné, že jde o stránku nebezpečnou.

Množina HTTP chyb je relativně velká, ovšem ne všechny HTTP chyby budou považovány za potenciálně nebezpečné chování, neboť u některých nelze ani částečně rozhodnout, zda nemohly vzniknout legitimními požadavky (seznam HTTP chyb považovaných za podezřelé uveden na výpisu 7.5). Byly vybrány takové HTTP chyby, které mohou s větší pravděpodobností vzniknout při podezřelé aktivitě a rovněž takové, které jsou příliš specifické na to, aby vznikaly ve větším množství v obvyklých situacích. Vyloučeny byly ty chyby, které se tohoto problému přímo netýkají (například `402 Payment Required`), aby byla šance, že bude důvěryhodná stránka chybně vyhodnocena, co nejnižší.

Výše popsané metriky budou samozřejmě fungovat jen pro v síti existující zařízení. Pokud útočník bude odesílat požadavky na taková zařízení, která v síti neexistují, v popsané statistice se to neprojeví. Toto lze do jisté míry vyřešit zachytáváním chyb pomocí události *onErrorOccured*, kde se takové chování projeví zachycením chyby *„request timed out“*. Zpracování chyb je podrobněji popsáno v podkapitole 7.3.5.

Pokud nastane situace, kdy nelze o cílové URL říci, zdali se jedná o veřejnou nebo privátní IP adresu, dojde k vytvoření nového záznamu (tj. objektu) v asociativním poli `hostStatistics`. Klíčem této položky je doménové jméno zdrojové URL a do objektu pod tímto klíčem jsou uloženy jednotlivé statistické hodnoty. Jedná se o výše popsané metriky, zejména: dosavadní počet požadavků z dané zdrojové domény, počet požadavků s chybovou

odpovědí z dané zdrojové domény, počet unikátních cílových domén a počet chyb typu *„request timed out“*.

V tomto objektu jsou rovněž zaznamenány všechny cílové domény, na které tento původce odeslal alespoň jeden požadavek, ve formě asociativního pole a pro každou cílovou doménu jsou zaznamenávány další statistické hodnoty. Pro každou tuto doménu se uchovává počet požadavků, které na ni směřovaly, počet HTTP chyb, informace o tom, zda tato cílová doména již zaznamenala chybovou odpověď a seznam všech URL pod touto doménou, které zaznamenaly alespoň jednu kladnou odpověď.

7.3.3 Stanovení limitních hodnot

Stanovení limitních hodnot je zásadní pro korektní fungování rozšíření. Je důležité zajistit, aby rozšíření chybně nezablokovalo některou z důvěryhodných webových stránek (tzv. *false positive*) a zároveň je však třeba zachovat detekci a blokování potenciálně nebezpečných webových stránek. Z toho vyplývá, že prakticky neexistuje zcela správné nastavení limitních hodnot, vždy můžou existovat některé webové stránky, které budou chybně zablokovány, a naopak některé potenciálně nebezpečné, které nemusí být detekovány. Cílem je najít určitou rovnováhu mezi těmito dvěma extrémy. Lze brát v potaz i to, že uživatel má možnost chybně zablokované stránky manuálně povolit prostřednictvím výjimek, nicméně nesmí se to stávat příliš často, aby toto chování uživatele neodradilo od používání rozšíření. Na druhou stranu, běžný uživatel nemá možnost poznat, zdali není daná stránka potenciálně nebezpečná, z toho důvodu je lepší nastavit tyto limitní hodnoty o něco striktněji.

Jako rozhodující metrika byla zvolena metrika reprezentující počet HTTP chyb pro unikátní cílové domény. Pro rozšíření je důležitější na kolika cílových doménách dojde k chybě, nežli to, kolik chyb vznikne na jedné cílové doméně. Tento přístup by měl minimalizovat pravděpodobnost, že dojde k zablokování důvěryhodné stránky. Ukázalo se totiž, že požadavky pocházející z důvěryhodných webových stránek vyústí v odpověď nesoucí chybový kód jen zřídka, zatímco útočník bude jen stěží schopen se při své aktivitě HTTP chybám vyhnout. Toto ovšem nezabraňuje útočníkovi odesílat požadavky na jednu doménu. Útočník tohoto chování může využít a pokusit se například kompromitovat zařízení nacházející se na dané doméně. Z tohoto důvodu, je potřeba zavést ještě další limitní hodnotu, tentokráte pro počet chybových odpovědí z jedné cílové domény, která povolí útočníkovi jen určitý počet HTTP chyb.

Při implementaci tohoto limitu je brán ohled na to, zdali se z jedné cílové domény vrací nejen chybné, ale i úspěšné odpovědi. V případě úspěšných odpovědi, dojde k dekrementaci chybového počítadla pro tuto doménu a občasné chyby důvěryhodých stránek se tak v této metrice neprojeví, protože budou vyváženy větším množstvím úspěšných odpovědí. Naopak pro útočníka je obtížné získat větší množství úspěšných odpovědí z jedné domény, nežli těch chybových. Aby bylo zabráněno útočníkovi zneužít tento mechanismus tím, že objeví jednu fungující cílovou URL, pomocí které si bude generovat úspěšné odpovědi, je z každé cílové URL započítána pro dekrementaci chybového počítadla pouze první kladná odpověď.

Metrika udávající počet unikátních cílových domén je využita pro výpočet poměru mezi cíli, které zaznamenaly alespoň jednu chybovou odpověď a cíli, které nezaznamenaly žádnou chybovou odpověď. Zablokování původce je provedeno na základě tohoto poměru. Poměr by měl být lepším způsobem nežli fixní nastavení hranice, neboť pokud daná stránka přistupuje na více různých cílových domén, je zde větší pravděpodobnost, že by mohla překročit stanovený fixní limit.

Původce je tedy zablokován ve třech situacích: pokud přesáhne limit daný poměrem nechybových a chybových cílových domén, nebo přesáhne limit na počet chybových odpovědí na požadavky z jedné cílové domény, nebo přesáhne limit na počet chyb typu *request timed out*. V prvním případě byl limit stanoven na 10% chybových cílových domén z celkového počtu všech cílových domén pro jednoho původce, s tím, že v případech, kde tento limit nedosahuje ani hodnoty jedna, je vždy tolerována jedna chybová cílová doména. Aby bylo zamezeno útočníkovi donekonečna zvyšovat tento limit tím, že bude rozesílat požadavky na stovky různých cílových domén v síti internet a bude tím získávat nechybové cílové domény, bylo nutné stanovit i maximální hodnotu tohoto limitu, a to na hodnotu 20.

V druhém případě byl limit stanoven na pět chybových odpovědí pro jednu cílovou doménu, s tím, že zde funguje vyvažování, kdy dvě úspěšné odpovědi sníží počítadlo chyb o jedna. To znamená, že aby toto počítadlo chyb nerostlo, je nutné, aby byl v každém okamžiku počet chybových odpovědí alespoň dvakrát nižší nežli počet úspěšných odpovědí.

Ve třetím případě byl limit na počet chyb stanoven na hodnotu 10, byla zvolena o něco vyšší hodnota, neboť se počítadlo těchto chyb za celou dobu běhu prohlížeče neresetuje a vyšší hodnota by měla zamezit zablokování důvěryhodné stránky.

Stanovení limitních hodnot bylo provedeno postupným posouváním limitů a kontrolou, zda tyto limity neporušují funkčnost důvěryhodných stránek. Přesto je však nutné zdůraznit, že jsou tyto limitní hodnoty stanoveny jen hrubě a může se v určitých případech prokázat, že jsou limity nedostatečné pro detekci útočníka anebo porušují funkcionalitu některé z navšívených webových stránek.

7.3.4 Zpracování odpovědí na požadavky

Zpracování odpovědi probíhá ve funkci navázané na událost *onHeadersReceived*, do které je opět předaný objekt nesoucí veškeré informace o této odpovědi. Nejprve je z tohoto objektu zjištěn původce požadavku, na který reaguje tato odpověď, a rovněž cíl původního požadavku (tj. původce odpovědi). Tyto informace lze nalézt ve vlastnostech `initiator` a `url` předaného objektu. V této fázi se opět ověřuje, zdali je původce požadavku uložen v asociativním poli `doNotBlockHosts`, nebo `blockedHosts`, pokud se v některém z těchto dvou polí původce vyskytuje, je zpracování zastaveno a odpověď povolena (v případě příslušnosti v `doNotBlockHosts`), nebo zablokována (v případě příslušnosti v `blockedHosts`). Následně je přečten HTTP status kód odpovědi (vlastnost `statusCode`) a další zpracování je rozděleno do dvou větví na základě toho, zdali odpověď nese jeden z vybraných chybových HTTP kódů (viz výpis 7.5) či nikoliv.

Pokud odpověď nese jeden z vybraných chybových kódů, je nejprve ověřena příslušnost původce odpovědi k objektu, který reprezentuje původce požadavku a je uložen v asociativním poli `hostStatistics`:

```
if (hostStatistics[sourceUrl.hostname][targetUrl.hostname] != undefined)
```

V případě, že není původce odpovědi nalezen, znamená to, že tuto odpověď není potřeba zpracovávat a zpracování je ukončeno. V opačném případě, je zvýšeno počítadlo HTTP chyb pro tohoto původce odpovědi, pokud je to první chyba, které se vyskytla od tohoto původce odpovědi, je rovněž zvýšeno i počítadlo počtu HTTP chyb pro **původce požadavků** napříč všemi cíli. Následuje kontrola, zdali první počítadlo nepřekročilo stanovený limit (tj. pět chyb) a zdali druhé počítadlo nepřekročilo stanovený limit na počet chyb původce napříč všemi cíli (tj. 10% chybových cílů ze všech cílů daného původce). Pokud došlo k překročení

jednoho, nebo druhého limitu, je tento původce požadavku zablokován (přidán do pole `blockedHosts`) a zpracování je ukončeno.

Pokud však odpověď nenese chybový kód a zároveň je původce odpovědi nalezen v objektu uloženém v `hostStatistics`, je zde pouze sníženo počítadlo chyb pro daného původce odpovědi o hodnotu 0,5, což odpovídá tomu, že dvě úspěšné odpovědi, vyvažují jednu chybnou. Toto se děje ovšem pouze v případě, že je tato odpověď první nechybovou odpovědí z této cílové URL.

7.3.5 Zpracování chyb

Zpracování chyb probíhá ve funkci navázané na událost *onErrorOccured*, do které je předaný objekt nesoucí informace o zachycené chybě. Obdobně jako v předešlých funkcích se zde nejprve z předaného objektu zjišťuje původce (vlastnost `initiator`) a cíl požadavku (vlastnost `url`), který vyústil v zachycenou chybu. Po získání těchto dvou informací, které jsou zásadní pro přístup do pole `hostStatistics`, následuje kontrola samotné chyby. Ve vlastnosti `error` předaného objektu se nachází řetězec (angl. *string*) identifikující druh vzniklé chyby. Sledované jsou pouze chyby typu *request timed out*, které signalizují přístup na neexistující IP adresu, což lze vyhodnotit jako podezřelé chování. Není možné sledovat a počítat všechny zachycené chyby, neboť se ukázalo, že i některé požadavky důvěryhodných stránek vyústí v některou z chyb. Nejčastěji se jednalo o chybu typu *blocked by client*.

Po zachycení chyby typu *request timed out* je pro daného původce zvýšeno počítadlo `errors` a je zkontrolováno, zdali toto počítadlo nepřekračuje stanovený limit. Pokud ano, je původce zařazen mezi blokované původce (tj. umístěn do pole `blockedHosts`).

Aby byla zajištěna schopnost rozšíření odlišit chyby typu *request timed out* vznikající aktivitnou útočníka od těchto chyb vznikajících výpadky síťové konektivity, bylo nutné implementovat detekci síťové konektivity. Pro tento účel byly využity události *online* a *offline* objektu `window`, který reprezentuje okno prohlížeče. Prohlížeč aktivuje korespondující událost vždy, když detekuje změnu stavu připojení do sítě. V případě aktivace události *online* je funkce zodpovědná za zpracování chyb navázána na událost *onErrorOccured*. V případě aktivace události *offline*, je tato funkce odvázána od události *onErrorOccured* a zachytávání chyb vzniklých problémy se síťovou konektivitou tím pádem neprobíhá. Relevantní kousky kódu jsou k nahlédnutí na výpisu 7.1.

```
1  window.addEventListener("online", function()
2  {
3    browser.webRequest.onErrorOccurred.addListener(
4        logError,
5        {urls: ["<all_urls>"]}
6      );
7  });
8
9  window.addEventListener("offline", function()
10 {
11   browser.webRequest.onErrorOccurred.removeListener(logError);
12 });
```

Výpis 7.1: Útržek JavaScript kódu navázání a odebrání funkce k události *onErrorOccured* v závislosti na události *online*/*offline* objektu `window`.

Kapitola 8

Testování

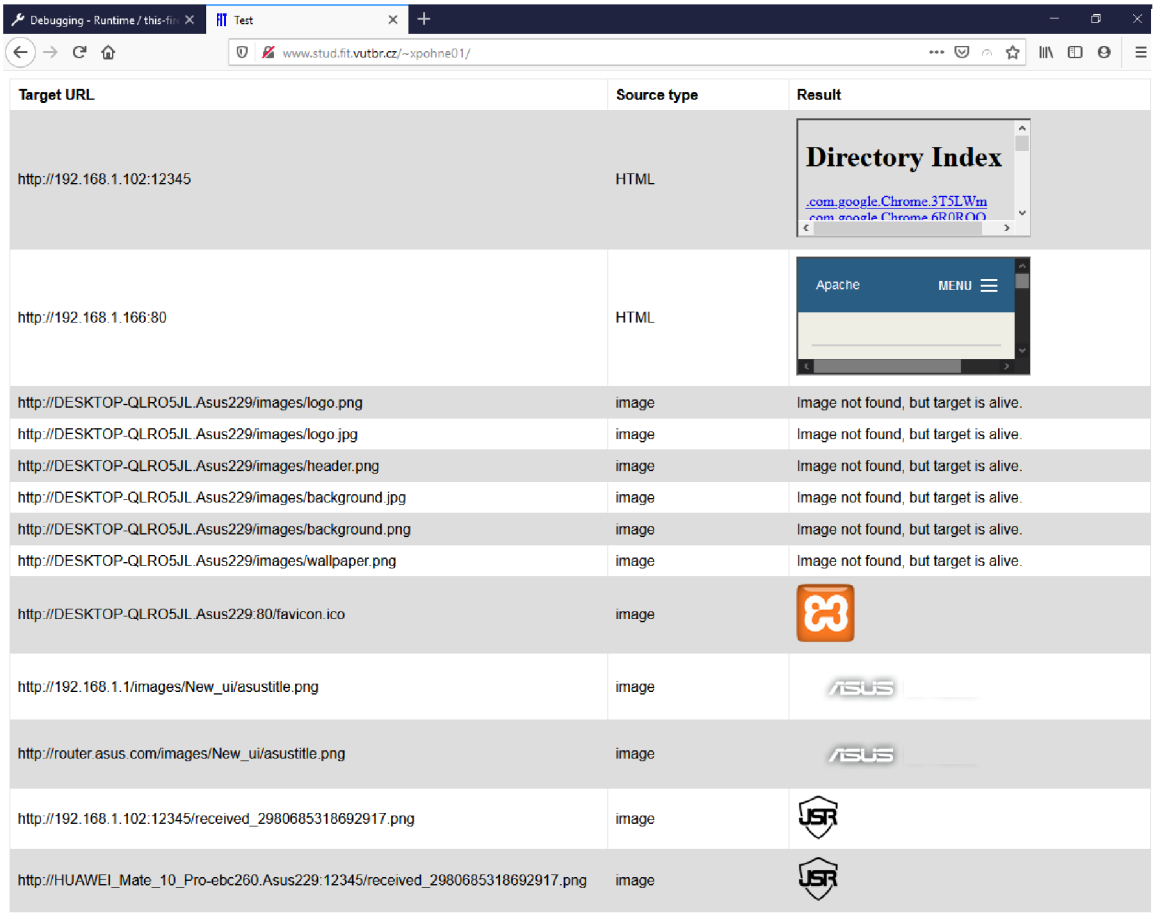
V této kapitole bude popsáno testování implementovaného řešení. Nejprve bude ověřena funkčnost implementace na webové stránce simulující aktivitu útočníka, následně bude změřen a vyhodnocen dopad implementovaného řešení na výkon jednotlivých internetových prohlížečů, rovněž zde budou porovnány obě verze řešení a budou zhodnoceny výsledky, kterých dosahují. V neposlední řadě dojde rovněž na porovnání implementovaného řešení s existujícími rozšířeními. Na závěr této kapitoly bude řešení zhodnoceno, budou identifikovány slabá místa a popsány možnosti vylepšení tohoto řešení.

8.1 Testování funkčnosti implementace

Řešení bylo samozřejmě testováno již v průběhu vývoje, nicméně pro demonstraci funkcionality se zdá býti vhodné to zahrnout i zde. Za účelem testování byla implementována jednoduchá webová stránka, která se inspiruje tou, která byla navržena v článku [4] a využívá například fragment kódu uvedený na výpisu 2.1. Tato stránka simuluje počínání útočníka ve veřejné síti internet, který se pokouší pomocí několika požadavků přistoupit k určitým zdrojům umístěných ve vytvořené lokální síti. V praxi by útočník využil více požadavků na více cílových adres, nicméně za účelem rozumné prezentace dosažených výsledků, byl počet požadavků omezen. Výsledky testování budou prezentovány prostřednictvím snímků obrazovky v následujících podkapitolách.

8.1.1 Výsledek testování bez aktivního rozšíření

Webová stránka simulující útočníka odesílá celkem třináct požadavků na lokální IP adresy a doménová jména příslušná zařízením ve vytvořené lokální síti. Všechny požadavky směřují na existující zařízení, nicméně ne všechny požadují existující zdroje. Výsledek návštěvy této webové stránky z vytvořené lokální sítě je k vidění na obrázku 8.1. Webová stránka je implementována formou tabulky, kde v prvním sloupci lze vidět cílové URL požadovaných zdrojů, ve druhém sloupci jejich typ a ve třetím potom samotný výsledek takového požadavku. Všimněme si, že přesto, že je stránka umístěna ve veřejné síti internet, podaří se jí bez problému načíst zdroje nacházející se na lokálních adresách. A to proto, že zneužívá internetový prohlížeč, spuštěný na zařízení v lokální síti, jako prostředníka. Toto chování bylo podrobněji vysvětleno v podkapitole 4.1. Prohlížeč si tedy tyto zdroje vyžádá od zařízení ve stejné síti, dokonce pomocí lokálního DNS serveru dokáže přeložit doménová jména zařízení na IP adresy a načíst zdroje i touto cestou.



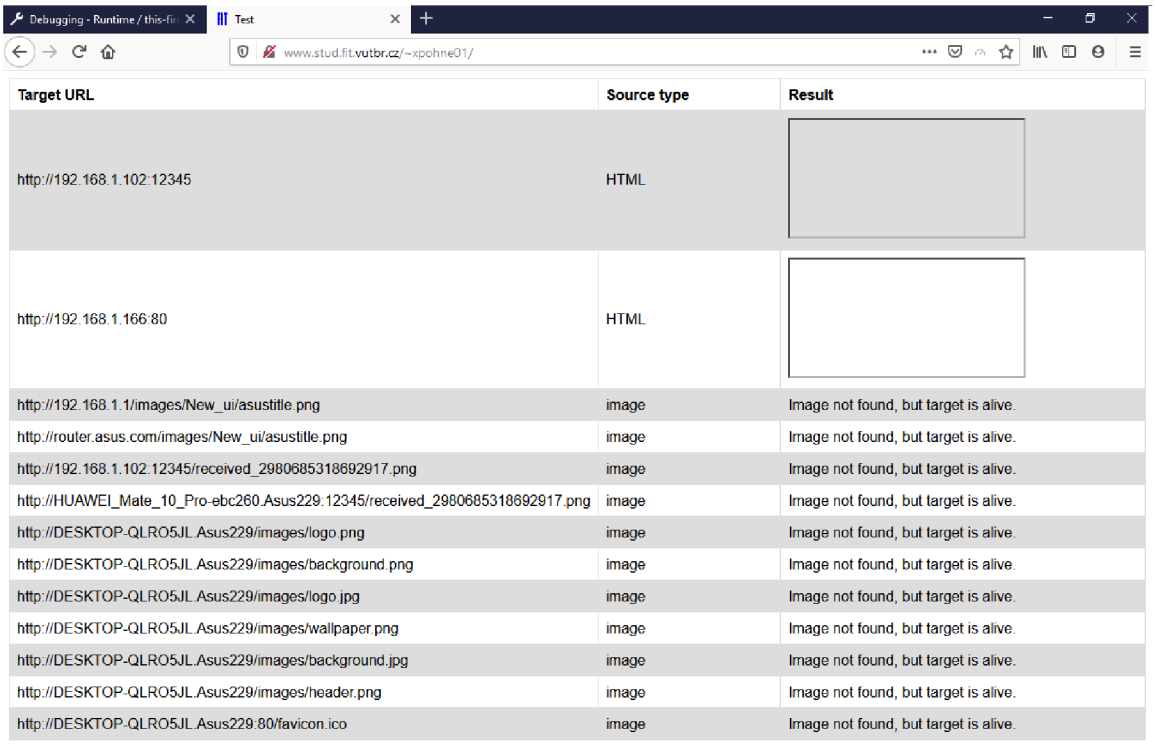
Obrázek 8.1: Snímek obrazovky zobrazující výsledek návštěvy nebezpečné webové stránky bez aktivního rozšíření.

8.1.2 Výsledek testování s aktivním rozšířením ve Firefoxu

Na obrázku 8.2 lze vidět, jak vypadá stejná webová stránka po aktivaci rozšíření v prohlížeči Firefox. Rozšíření správě rozpozná, že všechny požadavky generované touto webovou stránkou míří z veřejné sítě internet do lokální sítě a z toho důvodu je zablokuje. Doménová jména jsou přeložena pomocí DNS dotazu přímo v rozšíření, proto jsou zablokovány i ty požadavky směřující na URL obsahující doménové názvy. Rozšíření navíc vygeneruje třináct notifikačních oken, která upozorňují uživatele na podezřelou aktivitu, to však již na snímku obrazovky není vidět. Důležité však je, že se lokální síť pro útočnou webovou stránku jeví jednotně. V tomto případě obdrží útočník na všechny své požadavky odpověď s chybovým HTTP kódem, kterou automaticky vygeneruje prohlížeč po zablokování požadavku.

8.1.3 Výsledek testování s aktivním rozšířením v Chrome

Na obrázku 8.3 je pak k nahlédnutí stejná webová stránka s aktivním rozšířením v prohlížeči Chrome. Lze si povšimnout, že v tomto případě nejsou všechny požadavky správně zablokovány. Rozšíření sice správně rozpozná ty požadavky, které směřují na lokální IP adresy, ale už ne ty požadavky, které směřují na zařízení označené doménovým jménem. Aktivní limit na počet chybových odpovědí pro jedno cílové zařízení nakonec tuto webovou stránku



Obrázek 8.2: Snímek obrazovky zobrazující výsledek návštěvy nebezpečné webové stránky s aktivním rozšířením ve Firefox.

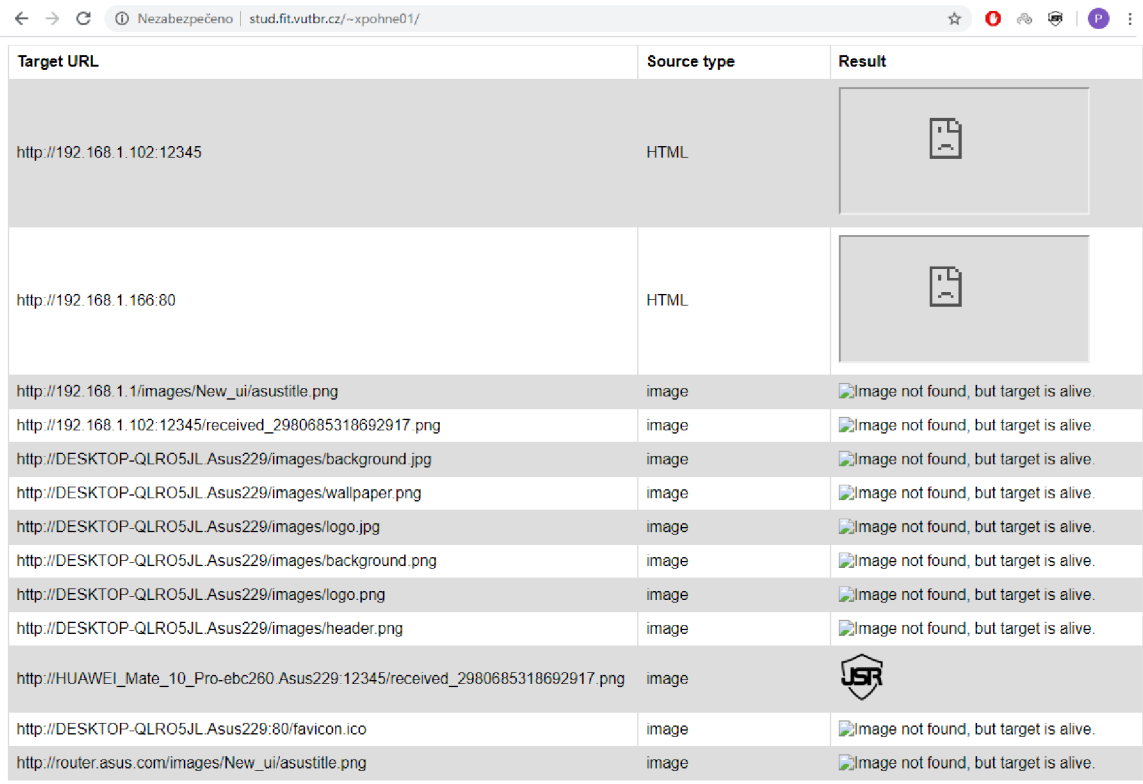
zablokuje, nicméně ne dostatečně včas na to, aby jedna z odpovědí nebyla kladně vyřízena. Z toho důvodu lze vidět, že se jeden z obrázků načetl, ale jiné ne. Překročení limitu je simulováno sekvencí požadavků, které požadují neexistující zdroje, tedy vyústí v chybovou odpověď (na obrázku tato sekvence začíná vyžádáním obrázku `background.jpg` a končí obrázkem `header.png`).

8.2 Vliv rozšíření na výkon prohlížeče

Pro rozšíření, které běží v podstatě neustále, je důležité zajistit, aby nějakým větším způsobem nezpomalovalo samotný prohlížeč. Hlavním cílem této fáze testování bylo zjistit a určit, o kolik implementované rozšíření zpomaluje běžné prohlížení webových stránek. Stanovení této hodnoty je důležité zejména pro prohlížeč Firefox, neboť v této verzi jsou využity DNS dotazy, které mohou být časově relativně náročné. V případě verze pro prohlížeč Chrome byla rovněž testována paměťová náročnost tohoto řešení, neboť zde se prakticky neustále ukládají statistické hodnoty do paměti a je důležité znát dopad tohoto chování na využití paměti zařízení.

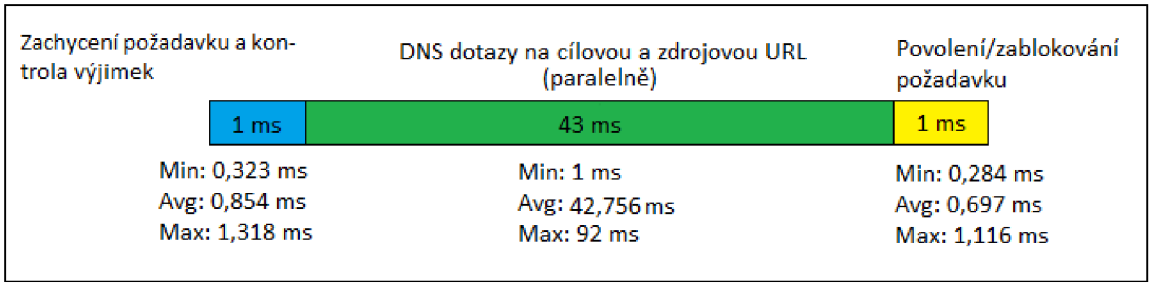
8.2.1 Výkon rozšíření v prohlížeči Firefox

Verze pro prohlížeč Firefox je specifická tím, že pro klasifikaci požadavku, který obsahuje doménové jméno ve zdrojové nebo cílové URL, provádí DNS dotaz (viz podkapitola 7.2). Při běžném prohlížení webových stránek bude takových požadavků s doménovými jmény ve zdrojové nebo cílové URL naprostá většina. Lze tedy předpokládat, že pro vět-



Obrázek 8.3: Snímek obrazovky zobrazující výsledek návštěvy nebezpečné webové stránky s aktivním rozšířením vprohlížeči Chrome.

šinu požadavků bude potřeba vykonat dva DNS dotazy – pro zdrojovou a pro cílovou URL. Na obrázku 8.4 je k nahlédnutí schéma s naměřenými hodnotami pro zpracování jednoho požadavku, který vyžaduje dva DNS dotazy. Měření bylo provedeno pomocí funkce `performance.now`, která vrací uplynulý čas od vytvoření interní reprezentace HTML dokumentu v prohlížeči. Měření probíhalo odděleně ve čtyřech fázích, aby se jednotlivá měření vzájemně neovlivňovala. Měřeny byly všechny požadavky vznikající při prvotním načtení webové stránky `seznam.cz`. Měření bylo opakováno celkem třikrát a z výsledných hodnot byl následně vypočten průměr.



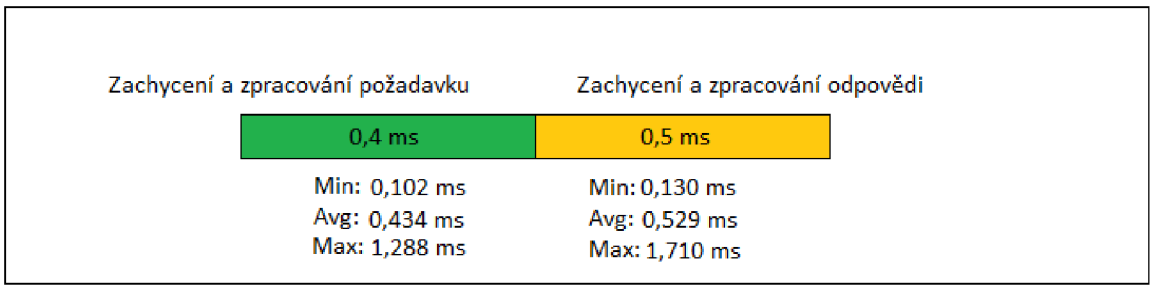
Obrázek 8.4: Časové schéma zpracování jednoho požadavku, ve verzi pro prohlížeč Firefox, s dvěma DNS dotazy.

Z výsledků tohoto měření lze vyčíst, že DNS dotazy zabírají majoritní část doby zpracování požadavku. Lze si rovněž povšimnout, že doba provedení DNS dotazu se liší na základě toho, zda byl záznam nalezen v DNS cache, či nikoliv (viz maximální a minimální naměřené hodnoty na obrázku 8.4). U všech požadavků v tomto měření bylo potřeba provést DNS dotazy dva, ty jsou však vykonávány paralelně a doba, kterou je nutné čekat na výsledek každého z nich se tím pádem překrývá. Z tohoto důvodu jsou na obrázku 8.4 oba DNS dotazy shrnuty do jedné hodnoty.

Naměřené hodnoty jsou samozřejmě ovlivněny rychlostí připojení i navštívenou webovou stránkou, nicméně lze říci, že implementované rozšíření má citelný dopad na dobu načítání webových stránek. Pokud vezmeme v úvahu hodnoty z proběhlého měření, každý požadavek se zdrží v průměru o 45 milisekund. Z grafu na obrázku 7.3 lze vyčíst počet požadavků při načtení webové stránky `seznam.cz` a díky tomu můžeme říci, že úplné načtení této stránky se zdrží, vlivem rozšíření, o 5,3 sekundy. Je důležité zdůraznit, že tato hodnota je platná opravdu pouze pro prvotní načtení, neboť při dalších načteních bude nepochybně více využita DNS cache. Dále u každé webové stránky závisí délka zdržení na tom, na kolik rozdílných doménových jmen přistupuje, tj. kolik DNS dotazů bude potřeba vyhodnotit. Demonstrovat to lze na grafu na obrázku 7.4. Přestože stránka `youtube.com` při svém načtení odesílá o dva požadavky více, nežli testovaná stránka `seznam.cz`, počet unikátních cílů je značně nižší, proto bude i celkové zdržení způsobené čekáním na rezoluci DNS dotazů kratší.

8.2.2 Výkon rozšíření v prohlížeči Chrome

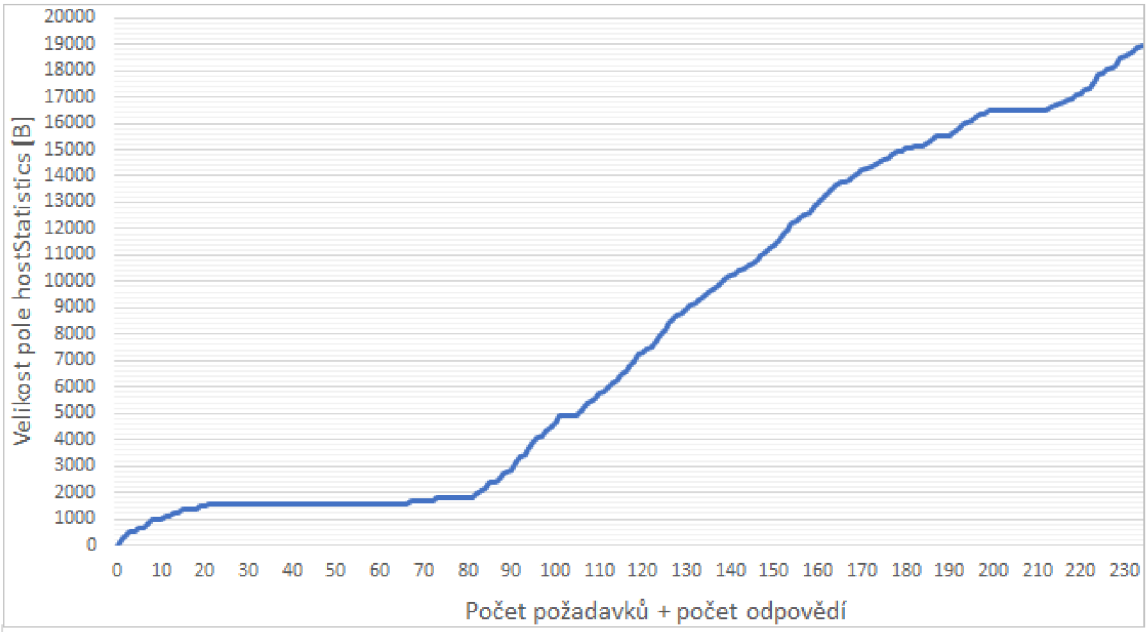
Narozdíl od verze pro prohlížeč Firefox, tato verze pro zpracování požadavků nevyužívá DNS dotazů. Z grafu na obrázku 8.4 tedy odpadá největší část, která odpovídá právě DNS dotazům. Na místo DNS dotazů je zde prováděna analýza (popsána v podkapitole 7.3), která je ovšem časově nesrovnatelně rychlejší než DNS dotazy. V podstatě se jedná o vytváření nových objektů a vyhledávání v asociativních polích, což jsou operace trvající, v nejhorších případech, jednotky milisekund. Měření, provedené opět pomocí funkce `performance.now` (viz graf na obrázku 8.5), ukázalo, že celkové zpracování jednoho požadavku netrvá déle nežli tři milisekundy. Tentokrát bylo měření rozděleno na dvě fáze – zachycení a zpracování požadavku, zachycení a zpracování odpovědi na tento požadavek. Doba zpracování požadavku i odpovědi je pochopitelně závislá na velikostech asociativních polí, která je nutné prohledávat. Měření bylo provedeno po navštívení čtyřiceti různých webových stránek, tak aby tato asociativní pole byla určitým způsobem obsazena. Měřeny byly opět všechny požadavky vznikající při prvotním načtení webové stránky `seznam.cz`.



Obrázek 8.5: Časové schéma zpracování jednoho požadavku ve verzi pro prohlížeč Chrome.

Z výsledků měření je patrné, že verze pro Chrome nemá žádný znatelný dopad na rychlost načítání navštívených webových stránek. Vezmeme-li v úvahu opět počty požadavků uvedené na grafu na obrázku 7.3, v nejhorším případě, tj. v případě fotogalerie `shutterstock.com`, nebude za těchto podmínek zdržení vyšší než jedna sekunda. V tomto ohledu je verze pro prohlížeč Chrome uživatelsky lépe přijatelná, protože neprodlužuje načítání webových stránek. Na druhou stranu však neposkytuje úplnou ochranu, jak bylo ukázáno v podkapitole 8.1.3. Kromě toho, má tato testovaná verze značně vyšší nároky na paměť, neboť statistická data, sbíraná za účelem analýzy a odhalení potenciálně nebezpečných stránek, zůstávají v paměti rozšíření po celou dobu běhu prohlížeče.

Graf, udávající závislost alokované paměti asociativního pole `hostStatistics` na počtu zachycených požadavků a odpovědí, je k vidění na obrázku 8.6. Toto měření bylo provedeno za pomocí knihovní funkce `sizeof`[1], která vrací přibližnou velikost objektu v bytech. Přibližnou velikost proto, že JavaScript nenabízí žádnou možnost, jak zjistit přesnou velikost daného objektu. Tato funkce byla volána vždy při zachycení požadavku i odpovědi na tento požadavek, protože v obou případech může dojít k zapsání statistických hodnot do měřeného pole `hostStatistics`. Z těchto hodnot byl potom vykreslen uvedený graf. Testováno bylo opět prvotní načtení webové stránky `seznam.cz`, zachyceno bylo 117 požadavků a stejný počet odpovědí a výsledná velikost pole, po načtení stránky, byla 18396 bytů.



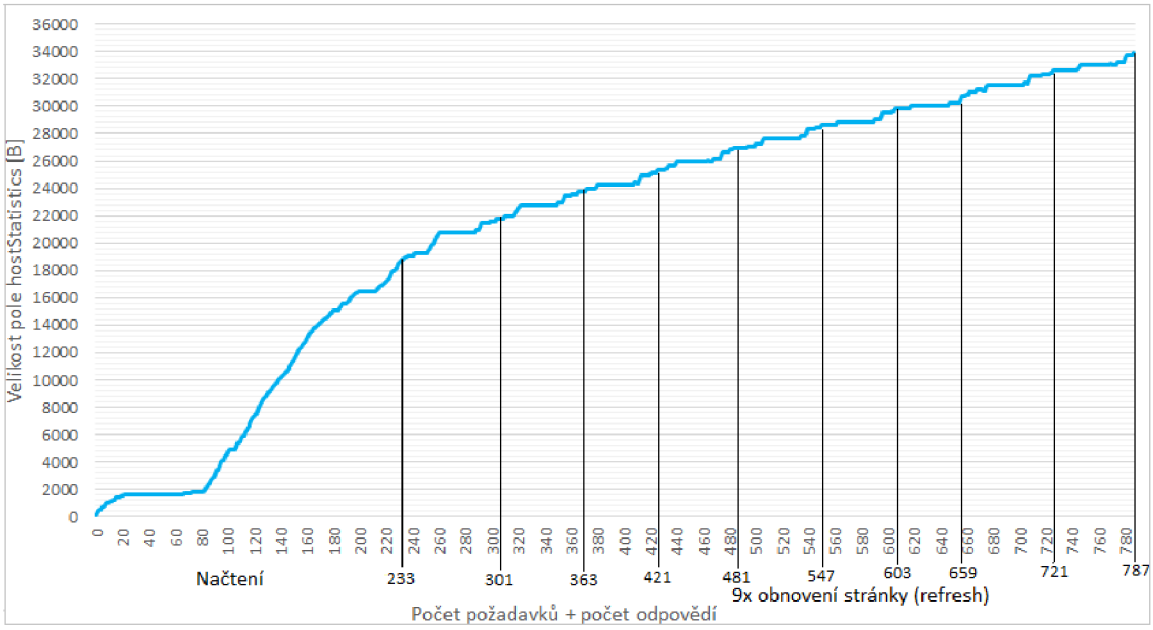
Obrázek 8.6: Graf růstu alokace paměti pole `hostStatistics` v závislosti na počtu požadavků a odpovědí při prvotní návštěvě stránky `seznam.cz`.

Je evidentní, že velikost alokované paměti tohoto pole má vzrůstající tendenci, nelze však říci, že by se alokovaná paměť zvětšovala s každým požadavkem nebo odpovědí. Záleží spíše na tom, na jaké URL požadavky směřují a do jaké míry jsou mezi sebou tyto URL unikátní. Toto chování lze pozorovat na grafu mezi požadavky 22-67, kde se alokovaná paměť téměř nemění. Lze předpokládat, že tyto požadavky mají shodné zdrojové doménové jméno a cílovou URL, jako již některý z požadavků v minulosti, což je důvodem, proč zde

[1] Celá knihovna dostupná zde: http://code.iamkate.com/javascript/finding-the-memory-usage-of-objects/.

nedojde k žádnému navýšení alokované paměti, neboť není třeba vytvářet nové položky pro uložení těchto hodnot. Jednoduše jsou pouze zvýšena příslušná počítadla. Z tohoto chování lze následně odvodit, že každé další načtení stránky `seznam.cz` zvýší alokovanou paměť jen minimálně (například vinou jiných zobrazených reklam/obrázků). I přesto však tvrzení, že alokovaná paměť má rostoucí tendenci, zůstává platné a pokud uživatel bude prohlížet různé webové stránky, alokovaná paměť neustále poroste.

Za účelem lépe poznat jakým způsobem se bude využití paměti vyvíjet v případě běžného používání rozšíření, byly provedeny ještě další měření. Každé z těchto měření pomůže odhalit vliv různých situací na využití paměti. První měření bylo zaměřeno na ověření tvrzení, že velikost alokované paměti pole `hostStatistics` poroste pomaleji v případě opakovaných návštěv stejné webové stránky. Předmětem měření byly opakované návštěvy webové stránky `seznam.cz`. Aby byly výsledky porovnatelné s grafem na obrázku 8.6, bylo měřeno prvních deset návštěv této stránky. Graf výsledků tohoto měření je zobrazen na obrázku 8.7. Z grafu lze vyčíst, že nárůst využité paměti při opakovaných návštěvách stránky je skutečně nižší nežli v případě první návštěvy. Při první návštěvě vzrostla využitá paměť o 18,6 kB, zatímco při dalších návštěvách rostla paměť v průměru jen o 1,7 kB. Do jisté míry za to může i fakt, že prohlížeč určité množství zdrojů ukládá do interní paměti cache a při znovunačtení této stránky nepotřebuje odesílat takové množství požadavků. Nicméně, z naměřených hodnot lze soudit, že úplně první návštěva stránky má na velikost alokované paměti větší dopad nežli zbývajících devět opakovaných návštěv. I přesto však nelze říci, že se využitá paměť ustálí a již dále neporoste, alespoň ne na vzorku deseti opakovaných návštěv.



Obrázek 8.7: Graf růstu alokace paměti pole `hostStatistics` v závislosti na počtu požadavků a odpovědí při deseti návštěvách stránky `seznam.cz`.

Poté, co předchozí měření prokázalo, že dopad opakovaných návštěv stejné stránky na velikost alokované paměti je skutečně nižší, bylo potřeba zjistit, jaký dopad mají návštěvy různých webových stránek. Bylo vybráno deset různých webových stránek s důrazem na to, aby byly tyto stránky co možná nejodlišnější v tom, jaké zdroje využívají. Toto měření

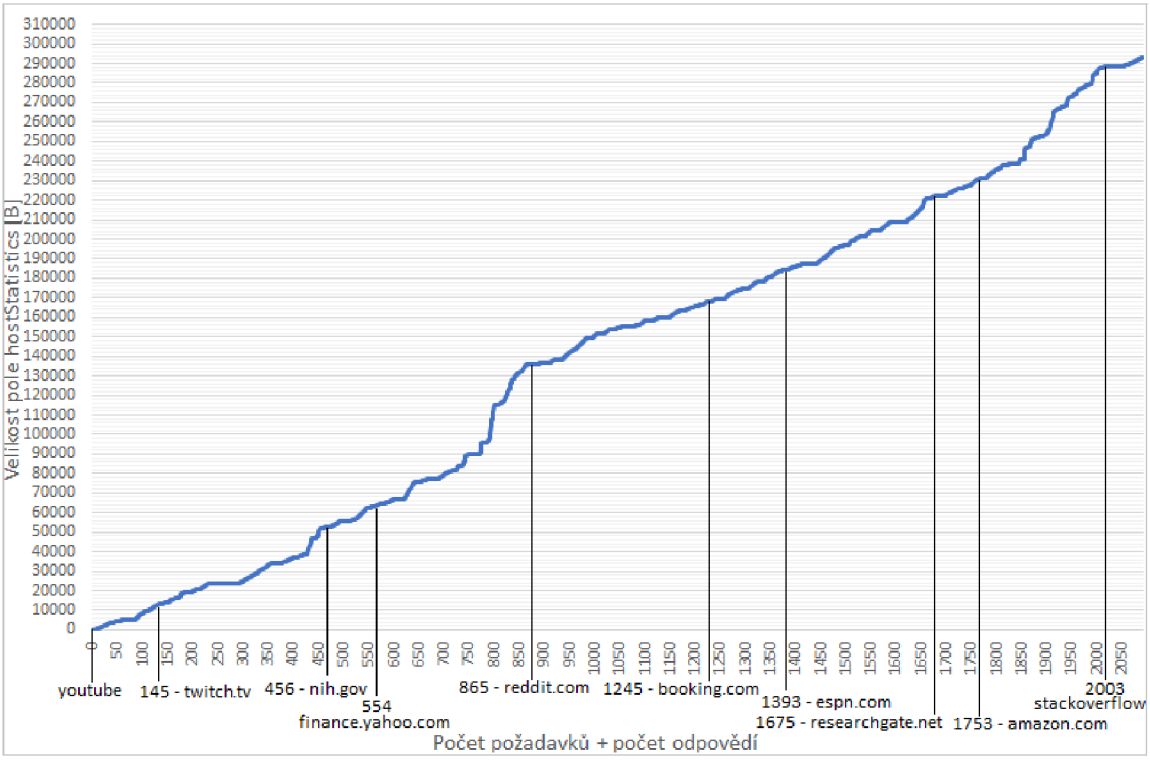
mělo testovat ten nejhorší případ, kdy jednotlivé stránky nevyužívají stejné zdroje. Webové stránky byly vybrány z žebříčků globálně nejnavštěvovanějších stránek dle kategorií obsahu[2], každá z těchto stránek se nacházela v prvních deseti nejnavštěvovanějších stránek ve své kategorii. Seznam vybraných stránek včetně jejich kategorií (pozn. kategorie jsou uvedeny v anglickém jazyce):

- Arts – `youtube.com`
- Games – `twitch.tv`
- Health – `nih.gov`
- Home – `finance.yahoo.com`
- News – `reddit.com`
- Recreation – `booking.com`
- Sports – `espn.com`
- Science – `researchgate.net`
- Shopping – `amazon.com`
- Computers – `stackoverflow.com`

Měření probíhalo stejným způsobem jako v případě opakovaných návštěv stejné stránky, tedy všechny stránky byly postupně navštíveny a z naměřených hodnot byl vykreslen graf, který je zobrazen na obrázku 8.8. V grafu jsou vyznačeny čísla prvních požadavků následující načítané webové stránky. Z intervalu mezi značkami lze tedy vždy odvodit o kolik vzrostla využitá paměť na jedné webové stránce a také počet zachycených požadavků a odpovědí. Z grafu je evidentní, že alokovaná paměť roste relativně rychle. V průměru každá z těchto navštívených stránek přispěla k nárůstu paměti o 29,3 kB, v největší míře se na nárůstu podílela stránka `finance.yahoo.com` s nárůstem využití paměti o 72,7 kB, v nejmenší míře potom stránka `stackoverflow.com` s nárůstem o 6,3 kB. Po návštěvě těchto deseti stránek celková velikost alokované paměti pole `hostStatistics` vzrostla o 293 kB. To by mohlo být pro dlouhodobé využití rozšíření relativně problematické, neboť není přijatelné, aby se rozšířením využívaná paměť vyšplhala do desítek megabytů.

Protože v předchozím měření byla testována skutečně nejhorší situace, bylo za účelem porovnání provedeno ještě jedno další měření. Tentokrát byla testována situace, kdy jednotlivé stránky mezi sebou sdílejí většinu zdrojů. Pro tuto situaci bylo využito webové stránky `novinky.cz`, kde byla postupně navštívena domovská stránka a následně devět různých článků. Vzhledem k tomu, že jsou tyto stránky umístěny pod jednou doménou, je zde vysoká pravděpodobnost, že budou částečně využívat stejných zdrojů. Výsledný graf na obrázku 8.9 toto tvrzení potvrzuje. Návštěva domovské stránky serveru `novinky.cz` měla za následek nárůst alokované paměti o 9,5 kB, nicméně návštěvy stránek pod touto doménou (tj. různých článků) měly za následek nárůst alokované paměti v průměru jen o 6,2 kB. I přesto, že paměť roste rychleji než v případě grafu na obrázku 8.7, který zobrazuje opakované návštěvy stejné stránky, je zde viditelná tendence pomalejšího růstu velikosti alokované paměti. Obzvlášť porovnáme-li to s grafem na obrázku 8.8. V tomto měření jsou

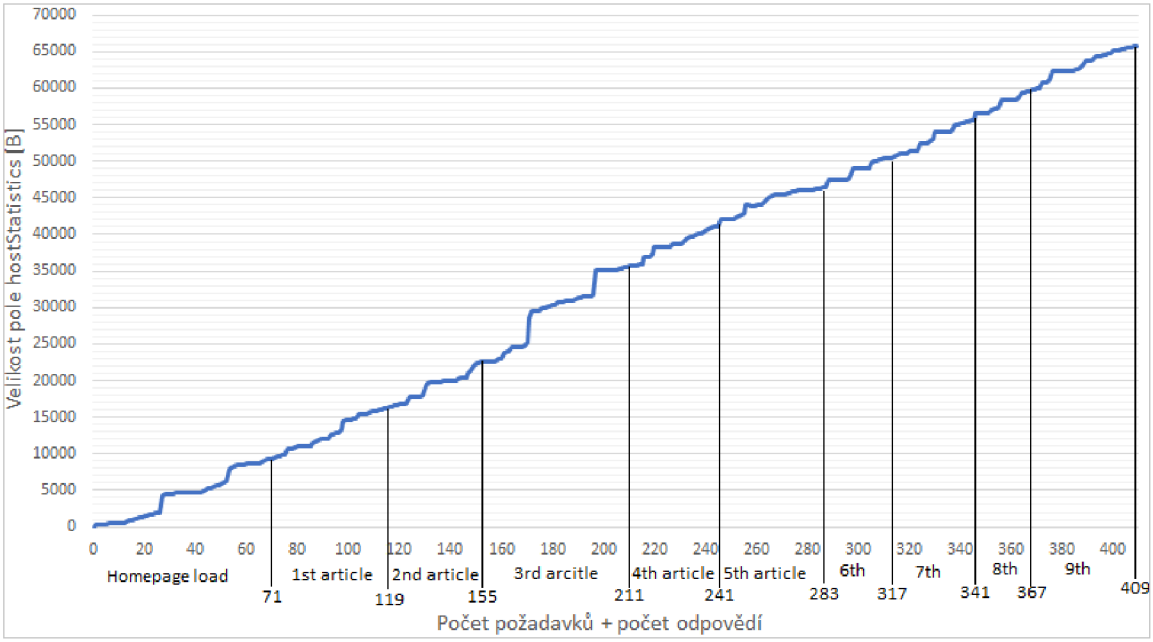
[2] https://www.alexa.com/topsites/category



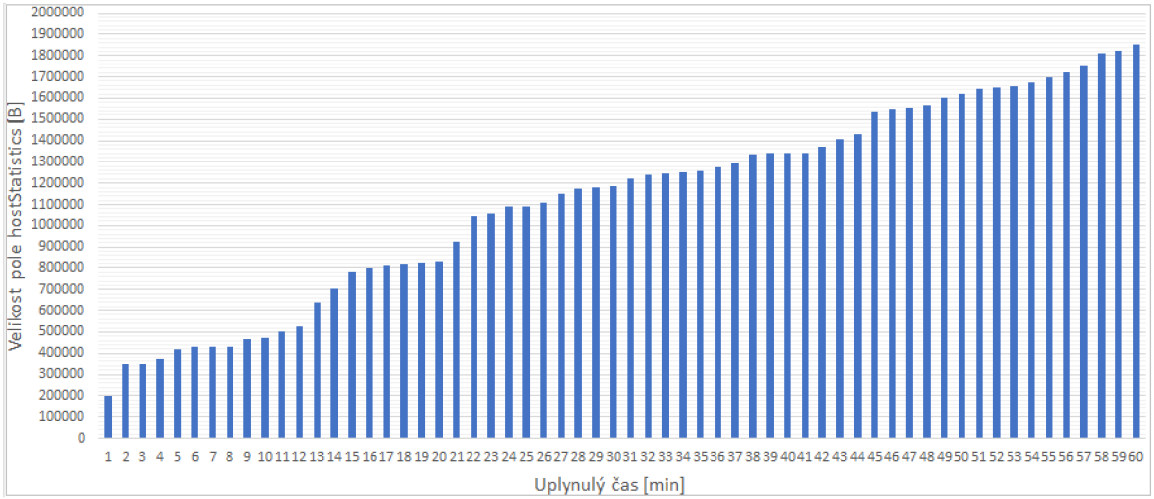
Obrázek 8.8: Graf růstu alokace paměti pole `hostStatistics` v závislosti na počtu požadavků a odpovědí při návštěvách deseti různých webových stránek.

na grafu pozorovatelná místa, kde nedochází k žádnému nárůstu paměti, což je něco, co se na grafu na obrázku 8.8 neobjevuje. Celková velikost alokované paměti potom dosáhla necelých 66 kB, což je značně nižší hodnota než v předchozím měření.

Samotné tyto výsledky nepřináší žádnou představu o tom, jak se bude vyvíjet velikost alokované paměti při běžném používání rozšíření. Neboť předešlé testy cílily na umělé scénáře, a to za účelem demonstrace chování implementovaného rozšíření. Nelze tedy opomenout ještě poslední měření týkající se využití paměti při běžném používání rozšíření. V tomto měření bylo, co možná nejlépe, simulováno chování uživatele při prohlížení různých webových stránek a byla sledována rostoucí velikost alokované paměti. Měření probíhalo po dobu jedné hodiny a zachycení velikosti alokované paměti bylo provedeno každou minutu. Výsledek měření lze vidět na grafu na obrázku 8.10. Tentokrát představuje osa X uplynulý čas v minutách, lze tedy pozorovat jakým způsobem docházelo k navyšování využité paměti minutu od minuty. Výsledky měření dokazují to, co zde bylo již několikrát zmíněno a to fakt, že velikost alokované paměti roste, na konci měření odpovídala tato velikost hodnotě 1,85 MB. Pro úplnost je rovněž důležité zmínit, že v průběhu měření bylo zachyceno 10398 požadavků a stejný počet odpovědí. Výsledná velikost alokované paměti je na hranici přijatelnosti, uvážíme-li, že prohlížeč může zůstat spuštěný například deset hodin, dostala by se velikost alokované paměti až k hranici 20 MB. Samozřejmě za předpokladu, že by paměť rostla stejnou rychlostí jako v tomto měření.



Obrázek 8.9: Graf růstu alokace paměti pole `hostStatistics` v závislosti na počtu požadavků a odpovědí při návštěvách deseti stránek pod stejnou doménou.

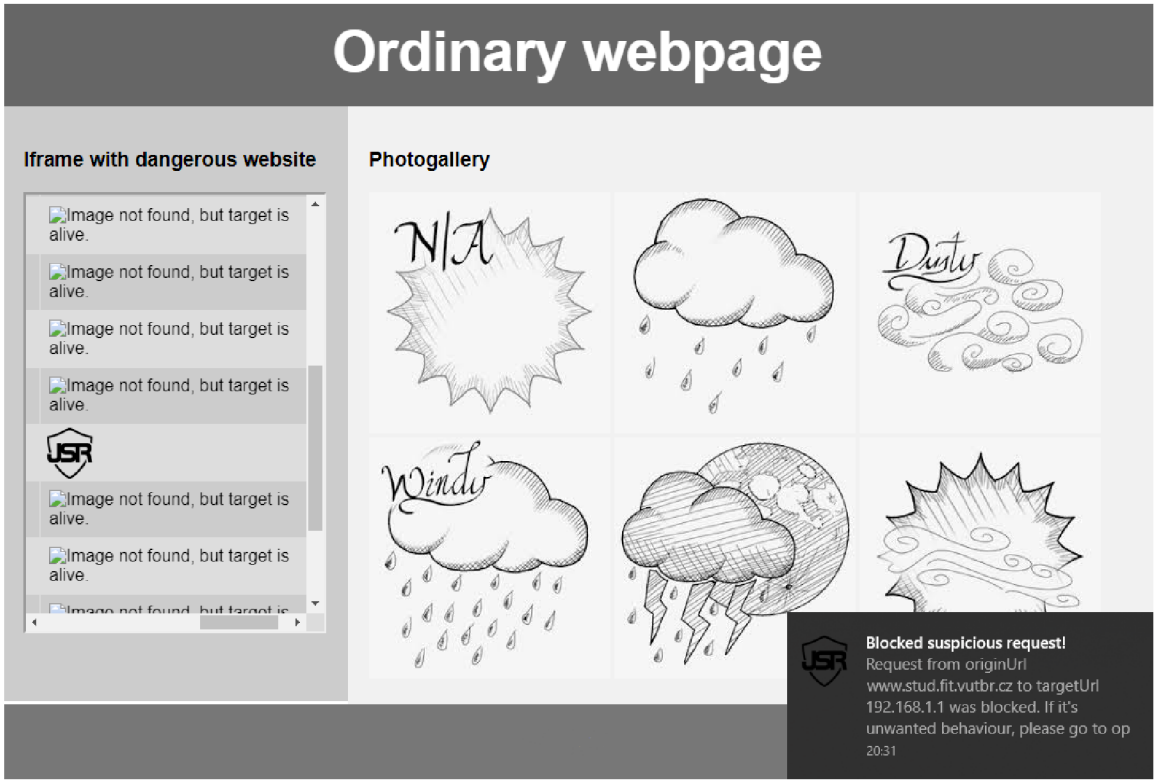


Obrázek 8.10: Graf růstu alokace paměti pole `hostStatistics` v závislosti na čase.

8.3 Testování webové stránky s injektovanou nebezpečnou stránkou

Cílem tohoto testu bylo ověření korektní funkcionality rozšíření v situaci, kdy je do důvěryhodné stránky injektována stránka potencionálně nebezpečná pomocí HTML tagu `<iframe>`. Tato situace může v praxi běžně nastat například v případě, že je do stránky obdobným způsobem vložena reklama. Korektní fungování rozšíření je takové, že zablokuje původce oné nebezpečné stránky v `iframe` a přitom nenaruší funkcionalitu důvěryhodné

stránky, do které je tato nebezpečná vložena. Autoři webových stránek by měli sice ručit za to, co se nachází na jejich stránkách a měli by být tedy zodpovědní i za jakákoliv nebezpečí, které vziknou pro uživatele, návštěvou jejich stránek. Nicméně se na toto z mnoha důvodů nedá spoléhat, a proto je důležité, aby rozšíření chránilo uživatele i proti této formě nebezpečí. Toto testování nemá smysl provádět v případě verze pro prohlížeč Firefox a to z toho důvodu, že tato verze posuzuje každý požadavek odděleně. Nehraje tedy roli, zda požadavek vznikl ve stránce vložené do `iframe`, nebo v původní navštívené stránce. Testována byla tedy pouze verze pro prohlížeč Chrome.



Obrázek 8.11: Snímek obrazovky zobrazující webovou stránku s injektovanou nebezpečnou stránkou v `iframe`.

Za účelem tohoto testování byla vytvořena jednoduchá webová stránka, která má představovat menší internetovou fotogalerii (viz obrázek 8.11). Do levé části této stránky byl vložen `iframe` obsahující nebezpečnou stránku, která provádí skenování lokální sítě. Je to ta stejná stránka, která byla využita pro testování v podkapitole 8.1. Pokud uživatel navštíví tuto fotogalerii bez aktivního rozšíření, webový prohlížeč rozešle několik HTTP požadavků. Část z nich bude mít shodný původ s webovou stránkou fotogalerie a bude mířit do veřejné sítě internet, to jsou ty požadavky, které načítají samotné obrázky ve fotogalerii. A část z nich bude mít původ stránky nacházející se v `iframe` a budou sloužit pro skenování lokální sítě, tj. budou mířit na adresy v lokální síti uživatele.

Na obrázku 8.11 je k vidění snímek obrazovky zobrazující návštěvu testované stránky s aktivním rozšířením v prohlížeči Chrome. Lze si povšimnout, že většina z obrázků viditelných v `iframe` se nenačetla, to znamená, že rozšíření správně rozpoznalo hrozící nebezpečí a včas stránku zablokovalo. Fungování hostitelské stránky však nebylo ovlivněno a obrázky tvořící skromnou fotogalerii byly bez problému načteny. V pravém spodním rohu obrázku

lze rovněž vidět notifikaci, pomocí které rozšíření upozorňuje uživatele na zablokování nebezpečného požadavku. Těchto notifikací se objevilo hned několik, zde je však zachycena pouze jedna z nich.

8.4 Porovnání řešení s existujícími rozšířeními

Poslední testování bylo provedeno s cílem zařadit implementované řešení do kontextu již existujících rozšíření. Test měl porovnat funkcionalitu implementovaného řešení s rozšířeními popsanými v kapitole 5, zejména s uMatrix a NoScript. Přesto, že se každé toto rozšíření zaměřuje na něco trošku jiného, jsou dosažené výsledky do jisté míry porovnatelné. Testování bylo provedeno opět prostřednictvím webové stránky ve veřejné síti internet, která se pokoušela přistupovat a načítat rozličné zdroje, z různých míst ve veřejné i lokální síti. Tato stránka byla postupně navštívena vždy s jedním aktivním rozšířením a výsledky byly zaznamenány do tabulky na obrázku 8.12. Testována byla pouze verze pro prohlížeč Firefox, ve kterém toto testování probíhalo. V případě NoScriptu byla použita starší verze označovaná jako NoScript Classic, neboť poskytuje veškerou funkcionalitu včetně ABE (viz podkapitola 5.2).

URL	Typ	Veřejná/lokální	JSR	uM	NoS	uM+	NoS+
stud.fit.vutbr.cz/ ~xpohne01/test_script.js	skript	veřejná	✓	⊘	⊘	✓	✓
ajax.googleapis.com/.../jquery.min.js	skript	veřejná	✓	✓	⊘	✓	✓
stackpath.bootstrapcdn.com/.../bootstrap.min.css	styly	veřejná	✓	✓	✓	✓	✓
fonts.googleapis.com/css?family=Merienda+One'	font	veřejná	✓	✓	✓	✓	✓
stud.fit.vutbr.cz/ ~xpohne01/test_page.css	styly	veřejná	✓	✓	✓	✓	✓
image.shutterstock.com/.../management-icon.jpg	obrázek	veřejná	✓	✓	✓	✓	✓
stud.fit.vutbr.cz/ ~xpohne01/index.html	dokument	veřejná	✓	✓	✓	✓	✓
qoret.com/.../Lose_yourself.mp3	audio	veřejná	✓	⊘	⊘	⊘	⊘
router.asus.com/client_function.js	skript	lokální	⊘	⊘	⊘	⊘	⊘
192.168.1.102/test_script.js	skript	lokální	⊘	⊘	⊘	⊘	⊘
192.168.1.1/images/New_ui/asustitle.png	obrázek	lokální	⊘	✓	✓	⊘	⊘
router.asus.com/images/New_ui/asustitle.png	obrázek	lokální	⊘	✓	✓	✓	✓
192.168.1.102/logo.png	obrázek	lokální	⊘	✓	✓	⊘	⊘
192.168.1.102/index.html	dokument	lokální	⊘	⊘	✓	⊘	⊘
192.168.1.102/audio.mp3	audio	lokální	⊘	⊘	⊘	⊘	⊘

Legenda: ⊘ Zablokováno, ✓ Povoleno

Obrázek 8.12: Tabulka zobrazující výsledky testování při návštěvě testované stránky na serveru `stud.fit.vutbr.cz` s aktivními rozšířeními.

Tabulka obsahuje celkem osm sloupců, první dva sloupce popisují odkud byl daný zdroj načítán a o jaký typ zdroje se jedná, třetí sloupec nese informaci o tom, zda se jedná o lokální, nebo veřejnou adresu. Tato informace byla do tabulky zařazena z toho důvodu, že ne pro každou URL je jasné o jaký typ adresy se jedná. Zbývající sloupce zobrazují již výsledky testování pro jednotlivá rozšíření. Sloupec nadepsaný `JSR` obsahuje výsledky testování implementovaného řešení, sloupce nadepsané `uM` a `NoS` obsahují výsledky testování s aktivním rozšířením uMatrix, respektive NoScript, ve výchozím nastavení. Zbývající dva sloupce, tedy sloupce nadepsané `uM+` a `NoS+`, obsahují výsledky testování s odpovídajícím aktivním rozšířením s pokročilým nastavením.

Ve sloupci `JSR` lze dobře vidět, že implementované řešení spolehlivě zablokuje všechny zdroje, které se nachází na lokálních URL, což je účel, se kterým bylo toto řešení navrženo a implementováno. Porovnáme-li to s výsledky dosaženými rozšířeními uMatrix a NoScript ve výchozím nastavením, lze vidět, že cílem těchto rozšíření je něco trochu jiného, nežli blokovat přístup webových stránek z veřejné sítě do lokální. NoScript i uMatrix se soustředí spíše na blokování načítání skriptů a médií (viz podkapitoly 5.2 a 5.3), v jejich výchozím chování lze vidět jen dva rozdíly, první ve druhém řádku, při načítání skriptu jQuery, který uMatrix povolí, protože se doména `googleapis.com` nachází na interním seznamu důvěryhodných domén, NoScript načtení však zablokuje i přesto, že i zde je tato doména označena jako důvěryhodná, nicméně NoScript nedůvěřuje rodičovské stránce s doménou `stud.fit.vutbr.cz`, což je důvodem k zablokování načítání skriptu jQuery. Druhým rozdílem je potom načtení HTML stránky `index.html` do *iframe* z lokální IP adresy, toto chování NoScript povolí, neboť ve výchozím nastavení není blokováno načítání žádných stránek do *iframe*, v uMatrix toto však dovoleno pouze pro domény v rámci doménové hierarchie rodičovské stránky, což adresa `192.168.1.102` není.

Protože NoScript ani uMatrix ve výchozím nastavení nedisponují prostředky pro blokování přístupu webových stránek z veřejné sítě do sítě lokální, bylo nezbytné u obou těchto rozšíření aktivovat pokročilé funkce a lépe je nastavit. V případě NoScript byl aktivován mechanismus ABE s pravidlem vymezující hranice přístupu webové stránky pouze do veřejné sítě internet. V případě uMatrix byl manuálně zakázán přístup na lokální IP adresy prostřednictvím matice (podobná matice je k vidění na obrázku zde 5.2). Rovněž byla, v obou případech, navštívená stránka označena jako důvěryhodná. Dosažené výsledky (ve sloupcích `uM+` a `NoS+`) jsou nyní mnohem lépe porovnatelné s implementovaným řešením. Lze vidět, že obě rozšíření nyní blokují přístup na číselné lokální IP adresy, nedokáží však rozpoznat lokální adresu zadanou prostřednictvím doménového jména (případ `router.asus.com`). Navíc si uMatrix a NoScript zachovaly schopnost blokovat vešekrá média, ale i skripty, které nejsou načítané z adres uvedených na interním seznamu důvěryhodných adres.

Z těchto výsledků lze usoudit, že implementované řešení má mezi podobnými rozšířeními své místo, protože ani při pokročilém nastavení těchto rozšíření, se nepodařilo dosáhnout stejných výsledků, a proto NoScript ani uMatrix nechrání proti útoku popsaném v kapitole 4. NoScript i uMatrix dokáží zajisté uživatele ochránit proti běžným hrozbám, nicméně zcela otevřeně narušují funkcionalitu prohlížených stránek a uživatel musí strávit určité množství času, aby si tato rozšíření nastavil víceméně pro každou navštívenou stránku zvlášť. Oproti tomu, implementované řešení sice nechrání uživatele proti všem hrozbám, například neblokuje načítání médií či skriptů z veřejných adres. Nicméně chrání uživatele proti specifickému útoku, popsaném v kapitole 4, nezištně a bez nutnosti pokročilého nastavování, důraz je kladen i na to, aby implementované řešení co nejméně ovlivňovalo funkcionalitu navštívených stránek a při tom těmto stránkám zamezilo v přístupu ke zdrojům v lokální síti uživatele.

8.5 Zhodnocení a další možnosti vývoje

Testování prokázalo, že implementované řešení splňuje požadavky, se kterými bylo vyvíjeno. V průběhu testování se ovšem projevily určité slabiny návrhu i implementace, jejichž opravení je předmětem dalšího vývoje tohoto řešení. Implementované řešení ve verzi pro Firefox je ve velmi dobré pozici, paměťová náročnost je zanedbatelná a spolehlivě splňuje účel, za kterým bylo navrženo a implementováno. Jediným nedostatkem verze pro prohlížeč Firefox

je zpomalení načítání jednotlivých webových stránek, zejména pokud se jedná o prvotní načtení této stránky.

Oproti tomu, řešení ve verzi pro prohlížeč Chrome trpí větším množstvím nedostatků. Vzhledem k tomu, že tato verze spoléhá na analýzu požadavků a hledání potenciálně nebezpečných původců z HTTP provozu, nemůže zaručit úplnou spolehlivost blokování přístupu z veřejné sítě do lokální. V případě, že útočník zná strukturu vnitřní sítě a je schopný posílat takové požadavky, které nevyústí v chybovou odpověď, je implementované řešení bezmocné úplně. Stojí totiž na předpokladu, že se útočník nebude schopný vyhnout chybovým odpovědím. Na druhou stranu, tato verze téměř nezpomaluje načítání webových stránek, uživatel tedy ani nezaznamená, že je rozšíření aktivní. Problémem však je vysoká paměťová náročnost, která neustále roste. Toto je způsobené tím, že je nutné si pamatovat celé URL, které byly cílem některého z požadavků. Rovněž nelze zaručit, že tato verze chybně nezablokuje některou z důvěryhodných stránek. Nicméně i přes všechny tyto nedostatky, je tato verze plně funkční a je dobrou alternativou pro prohlížeče, které nedisponují DNS API, jako je například právě Chrome. Ve chvíli, kdy bude toto API dostupné i pro ostatní prohlížeče, je doporučeno jej využít stejně jako v případě verze pro prohlížeč Firefox.

Napravení výše zmíněných nedostatků je hlavním cílem dalšího vývoje implementovaného řešení. V případě verze pro prohlížeč Firefox je možné zrychlit vykonávání DNS dotazů, a to ukládáním předešlých výsledků do asociativního pole. Klíčem do tohoto pole by potom bylo doménové jméno, pro které je potřeba vykonat DNS dotaz a položkou informace o tom, zda toto doménové jméno bylo přeloženo na veřejnou IP adresu, nebo lokální. To by potom znamenalo, že se DNS dotaz musí vykonat pro každé doménové jméno pouze jednou. Bylo by však nutné zajistit, aby toto pole neomezeně nerostlo. Řešením by mohl být fixní počet položek, které by se cyklicky přepisovaly.

V případě verze pro prohlížeč Chrome, je nutné lépe otestovat a stanovit limitní hodnoty a to tak, aby nedocházelo k chybnému blokování důvěryhodných stránek, ale aby byla zachována detekce nebezpečného chování. Co se týče paměťové náročnosti této verze, tu je možné snížit několika způsoby. První možností je pravidelné promazávání asociativního pole, které slouží pro ukládání celých URL, zde je možnost u každé URL ukládat časové razítko a po uplynutí určité doby tento záznam smazat. Kontrola časových razítek by však vyžadovala dodatečnou režii. Rovněž je možné pravidelně uvolňovat celé toto pole, což by mělo rostoucí paměťovou náročnost stabilizovat. Druhou možností je ukládání doménových jmen na místo celých URL. Nejenom, že unikátních doménových jmen bude na každé navštívené stránce méně nežli unikátních URL, ale je zde i šance, že různé webové stránky budou požadovat zdroje z URL se stejným doménovým jménem. Toto by efektivně snížilo rychlost růstu využité paměti a potenciálně by mohlo dojít i ke stabilizaci. Obě dvě tyto možnosti však poněkud snižují schopnost rozšíření detekovat útočníka a umožňují mu těchto úprav zneužít.

Co se týče rozšíření funkcionality, nabízí se možnost implementace podobných funkcí, které nabízí uMatrix nebo NoScript, tedy vyhodnocování a blokování načítání nedůvěryhodných skriptů a médií, a to za účelem toho, aby se JavaScript Restrictor jednoho dne stal jediným bezpečnostním rozšířením, který uživatelé potřebují. Tyto funkce spočívají v zachycení a analýze HTTP požadavků, což je základem implementovaného řešení, samotné rozšíření by tedy nemělo být větším zásahem do aktuální implementace. Bylo by však nezbytné navrhnout vhodný způsob analýzy požadavků. Z požadavků lze samozřejmě vyčíst o jaký typ zdroje se jedná, což by mohl být ideální začátek pro blokování skriptů nebo médií.

Kapitola 9

Závěr

V této práci jsem se zabýval možnostmi zneužití internetového prohlížeče jako *proxy* pro skenování lokální sítě uživatele, jak toto chování detekovat a také jak mu zamezit. Získané znalosti a vědomosti z oblastí politiky stejného původu, prohlížečových rozšíření a analýzy útoku na lokální síť dopomohly k návrhu řešení detekce a ochrany proti tomuto typu útoku. Navržené řešení sleduje HTTP požadavky odesílané prohlížečem a klasifikuje je na bezpečné a nebezpečné na základě toho, zda je jejich cílem zařízení s veřejnou IP adresou, nebo lokální. Nebezpečné požadavky jsou implicitně zablokovány a uživatel je upozorněn na potenciálně nebezpečnou činnost aktuálně navštívené webové stránky.

Navržené řešení bylo implementováno ve dvou verzích. První verze je určena pro prohlížeč Firefox a využívá DNS API pro překlad doménových jmen na IP adresy. Druhá verze je určena pro prohlížeč Chrome a ostatní prohlížeče, které nedisponují tímto DNS API. Tato verze spočívá v analýze HTTP požadavků a odpovědí a snaží se v tomto provozu identifikovat potenciálně nebezpečné původce. Přestože obě tyto verze byly implementovány za účelem rozšíření stávající funkcionality existujícího prohlížečového rozšíření JavaScript Restrictor, byla zachována i samostatná *standalone* verze implementovaného řešení.

Implementované řešení bylo rovněž podrobeno několika testům a měřením za účelem ověření funkcionality a výkonnosti. Testování prokázalo, že implementované řešení splňuje požadavky, které vedly k jeho navržení a implementaci. Testování však poukázalo i na nedostatky implementovaného řešení. Implementované řešení ve verzi pro Firefox má negativní dopad na rychlost načítání webových stránek, na druhou stranu verze pro prohlížeč Chrome tímto nedostatkem netrpí, avšak má zvýšené požadavky na využití paměti a nenabízí spolehlivou ochranu. Implementované řešení bylo taktéž porovnáno s existujícími rozšířeními, které se zaměřují na bezpečnost uživatele při prohlížení webových stránek a z výsledků tohoto porovnání lze říci, že implementované řešení má mezi těmito rozšířeními své místo.

V závěru práce, bylo celé řešení zhodnoceno, byly vyjmenovány silné i slabé stránky a bylo navrženo několik způsobů jak napravit stávající nedostatky. Rovněž bylo navrženo možné rozšíření řešení o novou funkcionalitu v rámci budoucího vývoje.

Literatura

[1] Al Fannah, N. M. One leak will sink a ship: WebRTC IP address leaks. In: IEEE. *2017 International Carnahan Conference on Security Technology (ICCST)*. 2017, s. 1–5.

[2] Angelo Spampinato. *uMatrix and Why You Should be Using it* [online]. 2019 [cit. 2020-05-11]. Dostupné z: https://medium.com/@angelospmusic/umatrix-and-why-you-should-be-using-it-c747015717e4.

[3] Barth, A. *The Web Origin Concept*. RFC 6454. RFC Editor, prosinec 2011. Dostupné z: https://tools.ietf.org/html/rfc6454.

[4] Bergbom, J. *Attacking the internal network from the public Internet using a browser as a proxy* [online]. 2019 [cit. 2019-12-30]. Dostupné z: https://www.forcepoint.com/sites/default/files/resources/files/report-attacking-internal-network-en_0.pdf.

[5] Giorgio Maone. *ABE - Application Boundaries Enforcer* [online]. 2009 [cit. 2020-05-11]. Dostupné z: https://noscript.net/abe/.

[6] Giorgio Maone. *NoScript* [online]. 2020 [cit. 2020-05-11]. Dostupné z: https://noscript.net/features.

[7] Luotonen, A. a Altis, K. World-wide web proxies. *Computer Networks and ISDN systems*. Elsevier. 1994, roč. 27, č. 2, s. 147–154.

[8] Maone, G. *ABE Rules Syntax And Capabilities* [online]. 2017 [cit. 2020-05-11]. Dostupné z: https://noscript.net/abe/abe_rules.pdf.

[9] MDN Contributors. *Anatomy of an extension* [online]. 2019 [cit. 2020-01-05]. Dostupné z: https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/Anatomy_of_a_WebExtension.

[10] MDN Contributors. *Building a cross-browser extension* [online]. 2019 [cit. 2019-12-29]. Dostupné z: https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/Build_a_cross_browser_extension.

[11] MDN Contributors. *Dns* [online]. 2019 [cit. 2020-01-04]. Dostupné z: https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/API/dns.

[12] MDN Contributors. *Same-origin policy* [online]. 2019 [cit. 2019-12-25]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy.

[13] MDN Contributors. *WebRequest* [online]. 2019 [cit. 2020-01-05]. Dostupné z: https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/API/webRequest.

[14] MDN Contributors. *Cross-Origin Resource Sharing* [online]. 2020 [cit. 2020-02-22]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS.

[15] Mehta, P. *Creating Google Chrome Extensions*. Springer, 2016.

[16] Nidecki, T. A. *What Is Same-Origin Policy* [online]. 2019 [cit. 2019-12-26]. Dostupné z: https://www.acunetix.com/blog/web-security-zone/what-is-same-origin-policy/.

[17] R. Fielding, J. R. *Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content*. RFC 7231. RFC Editor, červen 2014. Dostupné z: https://tools.ietf.org/html/rfc7231.

[18] Sebastiaan Lokhorst. *uMatrix popup panel* [online]. 2017 [cit. 2020-05-11]. Dostupné z: https://github.com/gorhill/uMatrix/wiki/The-popup-panel.

[19] Timko, M. *Vylepšení rozšíření pro omezení volání JavaScriptu*. Brno, CZ, 2019. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Dostupné z: https://www.vutbr.cz/studenti/zav-prace/detail/122066.

[20] van Kesteren, A.. *Cross-Origin Resource Sharing* [online]. W3C, 2014 [cit. 2020-01-18]. Dostupné z: https://www.w3.org/TR/2014/REC-cors-20140116/.

[21] Wikipedia. *Cross-site scripting* [online]. 2020 [cit. 2020-05-11]. Dostupné z: https://en.wikipedia.org/wiki/Cross-site_scripting.

[22] Zalewski, M. *Browser Security Handbook* [online]. 2009 [cit. 2019-12-25]. Dostupné z: https://code.google.com/archive/p/browsersec/wikis/Part2.wiki.