VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ
FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

ÚSTAV TELEKOMUNIKACÍ
DEPARTMENT OF TELECOMMUNICATIONS

SPECIFICKÉ MODULY PRO PODPORU MANUÁLNÍHO BEZPEČNOSTNÍHO TESTOVÁNÍ
SPECIFIC MODULES FOR MANUAL SECURITY TESTING SUPPORT

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — Jakub Osmani
AUTHOR

VEDOUCÍ PRÁCE — Ing. Daniel Paučo
SUPERVISOR

BRNO 2021

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

Bakalářská práce

bakalářský studijní program **Informační bezpečnost**

Ústav telekomunikací

Student: Jakub Osmani ***ID:*** 211805

Ročník: 3 ***Akademický rok:*** 2020/21

NÁZEV TÉMATU:

Specifické moduly pro podporu manuálního bezpečnostního testování

POKYNY PRO VYPRACOVÁNÍ:

Hlavním cílem práce je návrh a implementace speciálních modulů podporujících manuální testování bezpečnosti webových aplikací. V teoretické části práce analyzujte současný stav problematiky a nastudujte manuální testování bezpečnosti webových aplikací pomocí metodologie OWASP (The Open Web Application Security Project). V teoretické části se zaměřte na specifické zranitelnosti, které nelze nebo je nebezpečné testovat automatem. Navrhněte a implementujte nástroj v programovacím jazyku Python, který umožní usnadnění realizace bezpečnostního testování nejméně pro tři oblasti z metodologie OWASP. Funkčnost programu ověřte a prezentujte na příkladu webového serveru (př. OWASP Broken Web Applications).

DOPORUČENÁ LITERATURA:

[1] MEUCCI, Matteo; MULLER, Andrew. OWASP Testing Guide V. 4.0. Open Web Application Security Project, 2014, 30.

[2] SEDEK, Khairul Anwar, et al. Developing a Secure Web Application Using OWASP Guidelines. Computer and Information Science, 2009, 2.4: 137-143.

Termín zadání: 1.2.2021 ***Termín odevzdání:*** 31.5.2021

Vedoucí práce: Ing. Daniel Paučo

doc. Ing. Jan Hajný, Ph.D.
předseda rady studijního programu

UPOZORNĚNÍ:
Autor bakalářské práce nesmí při vytváření bakalářské práce porušit autorská práva třetích osob, zejména nesmí zasahovat nedovoleným způsobem do cizích autorských práv osobnostních a musí si být plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č.40/2009 Sb.

ABSTRAKT

Tato bakalářská práce se zabývá penetračním testováním a s touto činností spojenými standardy. Hlavním cílem teoretické části je objasnit svět penetračních testů a nejznámější dokumentace organizace OWASP. Popsány jsou zranitelnosti ze seznamu top deseti zranitelností a metodiky pro bezpečnější aplikace, zvané Application Security Verification Standard (ASVS). V praktické části je práce zaměřena na tvorbu tří nástrojů sloužících k automatizaci jistých aspektů penetračních testů.

KLÍČOVÁ SLOVA

Penetrační testování, OWASP, nástroje penetračního testování, XSS

ABSTRACT

This bachelor thesis deals with the concept of penetration testing and the standards that coincide with it. The main aim of the theoretical part of this thesis is to describe the world of penetration testing, and the widely known OWASP documentation. Vulnerabilities from the top 10 vulnerabilities list as well as recommendations about secure web application development, from the Application Security Verification Standard (ASVS), are provided. The practical part of this thesis is focused on the development of three tools, that are to be used to help automate certain aspects of penetration testing.

KEYWORDS

Penetration testing, OWASP, tools for penetration testing, XSS

OSMANI, Jakub. *Specifické moduly pro podporu manuálního bezpečnostního testování*. Brno, 2020/21, 62 s. Bakalářská práce. Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav telekomunikací. Vedoucí práce: Ing. Daniel Paučo

Vysázeno pomocí balíčku `thesis` verze 4.03; `http://latex.feec.vutbr.cz`

Prohlášení autora o původnosti díla

Jméno a příjmení autora:	Jakub Osmani
VUT ID autora:	211805
Typ práce:	Bakalářská práce
Akademický rok:	2020/21
Téma závěrečné práce:	Specifické moduly pro podporu manuálního bezpečnostního testování

Prohlašuji, že svou závěrečnou práci jsem vypracoval samostatně pod vedením vedoucí/ho závěrečné práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.

Jako autor uvedené závěrečné práce dále prohlašuji, že v souvislosti s vytvořením této závěrečné práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a/nebo majetkových a jsem si plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon), ve znění pozdějších předpisů, včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č. 40/2009 Sb.

Brno ....

podpis autora*

*Autor podepisuje pouze v tištěné verzi.

PODĚKOVÁNÍ

Rád bych poděkoval vedoucímu bakalářské práce panu Ing. Danielu Paučovi za odborné vedení, konzultace, trpělivost, podnětné návrhy k práci a především příjemný a podpůrný přístup.

Obsah

Seznam obrázků

Seznam výpisů

Úvod

Tato práce je zaměřena na bezpečnostní praktiky, doporučení pro bezpečný vývoj webových aplikací a vývoj tří aplikací, které mohou být užity při ověřování bezpečnosti webové stránky.

Náš každodenní život je úzce spojen s internetem a na něm fungujícími aplikacemi. Prakticky veškerá administrativa se přesunuje na internet, nebo se o to usiluje, a to včetně internetového bankovnictví, sociálního života a podobně.

S touto skutečností se pojí fakt, že data obyčejného člověka musí neustále čelit hrozbám, kterým mají zamezit právě vývojáři aplikací. Jednou z možností zajištění bezpečnosti je využití penetračního testu, za účelem zjištění, jak aplikace obstojí při útoku.

Zaměření této práce je především na penetrační testování, se kterým napomáhá testerovi mnoho standardů. Nejdůležitější metodologií, o kterou se tato práce opírá, je od neziskové organizace OWASP (Open Web Application Security Project), která si dala za úkol pomoci se zabezpečením našeho internetu. Důležitými dokumenty jsou „OWASP top 10 zranitelností“ a „OWASP Application Security Verification Standard“.

Kromě teoretického zpracování penetračních testů je i cílem práce vytvořit tři aplikace, které mají k tomuto účelu dopomoci. Praktickým výstupem této práce jsou tři nástroje: *8search*, *crossget* a *newspeak*.

Nástroje *8search* a *newspeak* jsou investigativního charakteru. Slouží jako nástroje pro první fázi testu. *8search* hledá adresáře webové stránky, aby si tester mohl udělat obrázek o její struktuře. *newspeak* poté ze stránky může vytáhnout klíčová slova a vytvořit jedinečný seznam pro danou stránku. Zároveň obsahuje dva moduly pro hledání klíčových slov na sociálních sítích.

Aplikace *crossget* slouží k nalezení zranitelnosti XSS (Cross Site Scripting) na dané webové stránce. Nejedná se o kompletní a robustní prohledání, program pouze zkouší možnosti tzv. *reflektovaného XSS* a *slepého XSS*.

1 Penetrační testování

1.1 Co je to penetrační testování?

Penetrační testování je simulací průběhu útoku na zvolenou entitu. Touto entitou může být počítačová síť, mobilní zařízení, IoT (Internet of Things) zařízení, server, kritická infrastruktura či město. Jedná se o jakýsi test bezpečnosti sytému proti reálnému útoku bez špatných následků, jako je například únik citlivých dat nebo poškození systémů. Umožňuje komplexní prozkoušení nově nasazených aplikací, bezpečnostních systémů, firewallů a IPS/IDS. V této práci se budeme soustředit na testování webových aplikací

Testy lze obecně rozlišit na externí a interní: [1]

- *Externí* - Externím testováním se simuluje útok z vnější sítě. Jedná se o standardní útok, který má původ mimo testovanou organizaci. Útočí se na servery, aplikace, webové aplikace apod. Tento typ útoku je mnohem zdlouhavější, komplexnější a budeme se ním v této práci zabývat.
- *Interní* - Interním testováním se simuluje škoda, která může být způsobena důvěryhodnou osobou, či útočníkem v síti. Tento typ testování se soustředí na odhalování chyb, zranitelností, špatné nastavení autentizace. Obecně tester získává roli zaměstnance a snaží se svou roli eskalovat.

Je důležité rozlišit termíny penetrační test a posouzení zranitelností, které se mohou zdát velmi podobné.

Posouzení zranitelnosti je více automatizovanou činností, zaměřena především na hledání známých zranitelností, ale ne na jejich zneužití a demonstraci dopadu zneužití. Po skončení posouzení je vytvořen seznam zranitelností, který je seřazen podle jejich závažnosti a dopadu na fungování organizace. V menším rozsahu se může jednat i o fázi v penetračním testu. [2]

Penetrační test (také označován jako pentest) bývá proveden etickým hackerem manuálně a komplexně, a také nebývá prováděn tak pravidelně jako posouzení zranitelností. Při tomto testování etický hacker nehledá pouze přítomnost zranitelností či chyb, ale demonstruje jejich účinky a potenciální dopady na bezpečnost organizace.

Metodologie OWASP rozděluje testování na dvě fáze, které budou blíže popsány v kapitole 1.3. [3]

Dále lze testy rozlišit podle znalosti testera o systému na *Black Box*, *Grey Box* a *White Box*: [4]

- *Black Box* - Při tomto typu testu je etický hacker v typické pozici hackera, který nemá interní a detailní znalosti o vlastnostech a fungování systému. Testerovi jsou dostupné pouze veřejně přístupné zdroje, jako html kódy a veřejné adresáře. Tento typ testování vyžaduje znalost skenovacích technik, především mapování sítě/systému. Nevýhodou je, že pokud nedojde k proniknutí do sítě, nedojde k žádnému otestování bezpečnosti vnitřní infrastruktury.
- *White Box* - Jedná se o přesný opak Black Box testingu. Při tomto testu je testerům vše zpřístupněno, od dokumentace po zdrojové kódy. Právě skrze tyto masivní kolekce dat se jedná o nejvíce časově náročný způsob testování. Testeři provádí statickou analýzu kódu, využívají nástrojů pro dynamickou analýzu kódu a seznamují se s nástroji použitými pro vývoj daného systému. Existuje tu možnost odhalení nejen vnitřních zranitelností, ale i vnějších. Nevýhodou může být, že testeři nepřemýšlejí jako skutečný útočník.
- *Gray Box* - Tento typ testu je jakýsi mezistupeň mezi Black Box a White Box testingem. Testerům není vše zpřístupněno, ale dostávají některé zdroje, které nejsou veřejně přístupné, například informace dostupné běžným interním uživatelům. Tito testeři typicky mají základní informace o vnitřní struktuře sítě a přístup do této sítě. Výhodou je možnost testerů se soustředit na části sytému, které jsou náchylnější na útok a nemuset prohledávat celý systém bez jakýchkoliv znalostí o prostředí.

1.2 Motivace za penetračním testováním

Naše společnost a služby, které využíváme, jsou čím dál více propojovány s internetem, což vede ke zvýšení možností útoků na citlivá data či jiná aktiva. Jak vývojáři, tak poskytovatelé služeb se snaží o vytvoření co nejbezpečnějšího softwaru pro uživatele, aby mohli využívat jejich služeb bezstarostně.

Při vývoji může dojít ke vzniku chyb, které mohou být odhaleny až při útoku, v horším případě dokonce dávno po něm. Z tohoto důvodu zavádíme pojem penetračního testování, kdy tester dává vývojářům možnost podrobit jejich bezpečnostní mechanismy testům, zároveň dává možnost zjištění předem neznámých zranitelností a následně je opravit dříve, než dojde k nasazení.

Testuje se během celého vývojového cyklu, tedy od koncepce myšlenky až po finální nasazení. Ideálně by vývojáři měli být v kontaktu s bezpečnostními experty, nejen z důvodu vykonávání testů, ale i sepsání bezpečnostní politiky, *code review* apod.

Tester je vysoce kvalifikovaný člověk, schopný vyhledat a otestovat velkou škálu zranitelností. Jelikož test může být pouze tak kvalitní jako tester, je tedy nutné, aby testeři byli zkušení a mohli co nejlépe simulovat skutečný útok na organizaci.

Pomocí penetračních testů je možné předejít značné škodě a být připraven na situaci, kdy k opravdovému útoku dojde.

1.3 Průběh penetračního testování webových aplikací

V této kapitole se soustřeďme na penetrační testování webových aplikací podle metodologie OWASP. Tato metodologie rozděluje test na dvě základní fáze a to pasivní (kapitola 1.3.1) a aktivní (kapitola 1.3.2), která je následně dále rozdělena.

Existují různé standardy rozdělující průběh penetračního testu. Alternativně lze užít PTES, tj. Penetration Testing Execution Standard, který však není zaměřený přímo na webové aplikace, a navíc se od roku 2012 nedočkal aktualizace. Proto budeme vycházet z dříve zmíněné OWASP metodologie.

1.3.1 Pasivní fáze

V této fázi se útočník snaží pochopit logiku fungování webové aplikace, zkoumá její funkcionality a provádí základní výzkum prostředí. Na konci této fáze by tester měl mít základní znalosti o prostředí webu, jako jsou jeho přístupové body, funkcionality a obsah. Tomuto procesu se říká *fingerprinting*. [3]

Možnosti pro získávání informací je mnoho. Jedna ze základních je využití klasických vyhledávačů (Google, Bing, apod.) k nalezení volně dostupných informací o obsahu webových stránek, včetně .html souborů nebo jiných veřejně přístupných komponentů.

Další možností je nastavení HTTP proxy, která umožní testerovi sledovat všechny žádosti zaslané na webový server a jeho následné odpovědi. Tester může z těchto žádostí vyčíst, jaký operační systém běží na webovém serveru, a o jaký webový server se jedná (např. Apache nebo Microsoft). Tento způsob identifikace serveru je možný pouze, pokud nebyl na straně serveru zakázán či modifikován.

1.3.2 Aktivní fáze

Aktivní fáze se skládá z deseti podfází:

1. Testování konfigurace a nasazení:

   Moderní webové servery již neobsahují pouze pár aplikací, jedná se spíše o spletitou síť vzájemně spolupracujících funkcí. Dle metodologie OWASP je právě proto kritické, aby aplikace byly nejen správně a bezpečně nakonfigurovány, ale i nasazeny.

   Jediná chyba může vést k nepředvídatelným následkům, zranitelnost může být v back-end databázi, front-end aplikaci, či samotném serveru. Z tohoto důvodu je důležité otestovat správnost konfigurace a nasazení aplikace včetně ostatních služeb.

   Obecný zjednodušený postup je následující: Tester musí prozkoumat data získaná z pasivní fáze, přesněji pochopit složení celé webové aplikace a jaký to má dopad na její bezpečnost. Následně budou jednotlivé komponenty podrobeny testům, aby byly odhaleny případné zranitelnosti. Dalším testům budou podrobeny nástroje administrátora a autentizačních systémů. Jako poslední je nezbytné, aby byly spravovány všechny porty, které jsou klíčové pro běh aplikace. [3]

2. Testování přiměřenosti rozdělených uživatelských rolí:

   Tato fáze je zaměřena na přidělování rolí v systému. Klasicky bývají dvě základní role, uživatel a administrátor. Je ovšem kritické, aby uživatel měl přístup pouze k souborům, které jsou nezbytné pro vykonávání jeho činnosti.

   Zásadní je dodržování přiměřenosti, tedy aby nebylo užito příliš mnoho rolí s úzkým rozsahem, nebo málo rolí s moc širokým rozsahem.

   Úlohou testera v této fázi je zkontrolovat, jestli jsou role logicky rozděleny a pokud ne, tak deklaruje nové efektivně rozdělené role. [3]

3. Testování autentizace:

   V této fázi se testuje jakákoliv implementace autentizačních metod. Hledá se, zdali jsou autentizační data přenášena přes zabezpečený kanál HTTPS, nebo pouze HTTP a útočník může číst data.

   Další možností je bruteforce výchozích hesel. Pokud si uživatel nezapíše nové heslo a výchozí hesla se generují předvídatelným způsobem, pak může

útočník tuto skutečnost zneužít a hesla prolomit. Na tuto část se váže implementace lockout mechanismu, který po předem definovaném počtu neúspěšných pokusů o přihlášení (zpravidla tři až pět) omezí další možnost přihlášení na stanovenou dobu, případně musí administrátor povolit další pokus.

Testuje se i způsob obejití autentizačních funkcí. Může se jednat například o zadání přímé cesty ke stránce, která vyžaduje autentizaci, ale mechanismus je nevhodně implementován a tím pádem je neautentizovaný uživatel připuštěn. [3]

4. Testování autorizace:

V serverech bývají běžně obsaženy adresáře a v nich uložené soubory. Tyto soubory a adresáře jsou mapovány na ACL (Access Control List), kde jsou definovány skupiny uživatelů i samotní uživatelé, kteří mohou k těmto souborům přistupovat, případně modifikovat.

K usnadnění práce se soubory bývá využíváno server-side skriptů. Ty pomáhají se správou souborů, fotek, načítání textu a jiných služeb. V těchto skriptech mohou vzniknout chyby, které umožní útočníkovi přistupovat k souborům a adresářům mimo jeho povolený dosah, přistupovat k souborům bez nutnosti autentizace a přistupovat k souborům po odhlášení. Mimo přístup zde existuje i možnost nahrávání skriptů třetí strany na webový server.

Tester v této fázi mimo jiné zjišťuje, kde je útočníkovi umožněno vkládat vlastní vstup a otestovat techniky manipulace se soubory, skripty a vlastními právy. [3]

5. Testování správy relace:

Jedna ze základních komponent dnešních webových aplikací je správa relace. Přesněji je využíváno různých technik, aby nedocházelo k opakovanému autentizování uživatele. K tomuto účelu je využíváno cookies, či jiných tokenů.

Tyto tokeny (nejčastěji cookies) stanovují dobu, po kterou potrvá relace, ke kterým službám bude uživatel mít přístup, případně i roli uživatele a další. [5]

Uživatel pak tedy serveru předává svůj token namísto opětovné autentizace například heslem. Skrze výše popsané vlastnosti jsou cookies/tokeny častým terčem útoku, z tohoto důvodu je jejich implementace v této fázi ověřena testerem.

Běžným postupem při testování, a zároveň i při útoku, je sběr cookies,

zpětné rozkládání cookies (pochopení algoritmu za jejich vytvořením) a manipulace cookie, tedy pokus o vytvoření přijatelného cookie na základě pozorování.

Zároveň je úkolem testera posoudit, zdali bylo cookie správně nakonfigurováno, např.: nastavením atributu HTTPOnly, kterým je snížena efektivita využití cross site scripting k přístupu a modifikaci cookie. [3]

6. Testování validace vstupů:

Validací vstupů rozumíme proces kontroly vstupních dat, které jsou webové aplikaci předány nejen přes její rozhraní, ale dokonce i přes backend od dodavatelů, partnerů apod.

Validaci lze provádět dvěma základními způsoby. *Syntakticky* - tato validace vynucuje určitý formát zadávaných dat (např.: datum, měna, ...) a *sémanticky* - ta zajišťuje správnost vstupu v logickém slova smyslu, např. cena je v určitém stanoveném rozmezí. [3]

V této fázi tester hledá chyby v implementaci validace a k jakým zranitelnostem a exploitům tato skutečnost vede. Nejčastěji špatná implementace vede k XSS (Cross Site Scripting), či SQL injection, které jsou pravidelně umístěny v seznamu OWASP Top Ten Web Application Security Risks. [6]

7. Testování reakce na chyby:

V této fázi tester zkouší reakci webového serveru na nečekané HTTP žádosti, nebo neplatné URL. Z odpovědí serveru (např. se status kódem 404) lze často vyčíst informace o operačním systému na serveru, o serveru samotném, službách a využívaných portech.

Další možnou reakcí na chybu je výpis stack trace, který může vést k prozrazení informací o vnitřní podobě aplikace, např. jak interně pracuje s objekty. Tento výpis může nastat při manipulaci se vstupními formuláři pomocí přetvořených HTTP žádostí či jinými vstupními daty. [3]

Tato fáze se podobá a rozšiřuje informace získané z fingerprintingu (viz kapitola 1.3.1).

8. Hledání slabých kryptografických algoritmů:

   Tester v této fázi zkouší implementaci kryptografických algoritmů a metod. Zároveň testuje, zdali byly dodrženy normy a doporučení ohledně využití kryptografie.

   Zásadní chybou bývá přenos citlivých dat přes nezabezpečené kanály HTTP nebo využití funkcí, které jsou známé jako náchylné na útok např. hashovací funkce MD5. Kromě funkcí musí být zvoleny i správně parametry, jako např. správná délka klíčů. [7]

   Další možnou zranitelností je využití *Padding Oracle* ve webové aplikaci. Jedná se o funkci, která dešifruje zašifrovaná data od klienta. Přítomnost této funkce může potenciálně umožnit útočníkovi přístup k citlivým datům nebo k eskalování svých práv v rámci systému. Častá chyba *Padding Oracle* bývá oznamování validity výplně (padding) zadaných dat. [3]

9. Business logic testing:

   Testování *business logic* je činnost, kterou nelze vykonávat automaticky. Vyžaduje od testera značnou kreativitu a nekonvenční myšlení. V této fázi tester zkouší, je-li implementovaná logika systému napadnutelná, nebo dokonce může být naprosto ignorována uživatelem.

   Jedná se o zkoušení naprosto hraničních a nečekaných situací. Zjednodušeně, pokud byl v systému definován postup jako soubor kroků 1, 2 a poté 3, jak bude systém reagovat, když uživatel přeskočí z kroku 1 přímo na krok 3?

   Komunikace mezi testerem a uživateli je žádoucí za účelem detailnějšího sblížení se s aplikací, kromě toho může tester komunikovat i s vývojáři za účelem pochopení vnitřního fungování a logiky. [3]

   Příkladem by mohl být e-shop generující věrnostní body, kterými lze nakupovat. V běžném průchodu systémem by si uživatel vybral zboží, vložil do košíku, zaplatil a obdržel body. Tester by mohl zkoumat situaci, kdy uživatel ihned po obdržení bodů objednávku zruší a de facto dostane body zdarma.

10. Testování client-side zranitelností:

    Mnoho webových aplikací využívá jazyků, jako je JavaScript, které se vykonávají na straně uživatele. Nejčastěji se jedná o email klienty, chatovací služby a jiné. Tato skutečnost prezentuje útočníkovi možnost se zaměřit na počítač klienta jako svůj vektor útoku, namísto přímého útoku na často lépe

zabezpečený server.

Tester tedy musí otestovat, zdali existují zranitelnosti na straně klienta. Mezi nejčastější patří XSS, HTML injection, CSS injection, Clickjacking, Code injection a jiné.

Pomocí zmíněných zranitelností může útočník získávat informace o klientovi, přístup do sítě, cookie nebo dokonce přístupové údaje administrátora. [3]

Tato fáze často úzce souvisí s fází validace vstupů, jelikož pomocí správné validace existuje možnost zamezení např. XSS, HTML a CSS injection.

1.4 Manuální X Automatizované testování

V předešlých kapitolách bylo popsáno penetrační testování jako takové. Po definicích a popisech nastává otázka, kdy využívat automatického testování a kdy manuálního?

Automatické testování je prováděno za pomocí automatizačních nástrojů, tester tedy pouze spouští nástroje a interpretuje data. Výhodou těchto testů je jejich rychlost, jednoduchá proveditelnost, možnost častějšího testování a prevence vzniku nových chyb a zranitelností. Po dokončení předá tester informace společnosti ve formě reportu, ve kterém jsou zahrnuty i návrhy na opravu chyb v systému.

U manuálního testování existuje možnost využití některých automatických nástrojů, ale celý princip tohoto testování spočívá na kreativitě testera a nekonvenčním myšlení. Na rozdíl od automatického testování, které nabízí pouze obecné testy a řešení, manuální testování je prováděno vždy specificky pro daný systém nebo aplikaci. Tester tedy může zachytit zranitelnosti specifické pouze testované aplikaci/systému. Navíc na vyžádání klienta může tester své testy dynamicky měnit. Nevýhodou je potřeba vysoce zkušených testerů, aby mohlo být s jistou určitostí řečeno, je-li systém dostatečně zabezpečen. Tester navíc nemusí mít konzistentní výsledky, tedy dva různé testy mohou dát dva různé výsledky.

Využití automatického testování při testování zranitelností jako je SQL injection, XSS či XXE (XML External Entities) je žádoucí, jelikož pro tuto činnost stačí obecné funkcionality. Problém nastává u zranitelností specifických aplikací, nebo dokonce při výskytu chyb v nástroji samotném. [8]

Uvedeme si příklad na hypotetickém *web spideru*. *Spider* tj. *web crawler* je nástroj, jehož činnost spočívá v indexování webových stránek a v nich vnořených odkazů za účelem sestavení topologie komplexní webové aplikace. Tato skutečnost poukazuje na fakt, že se může jednat o nástroj vhodný ve fázi fingerprintingu (viz 1.3.1), má však svá rizika spočívající v jeho implementaci. [9]

Činnost crawlera spotřebovává značné množství zdrojů, v naprosto krajních případech by se mohlo jednat i o formu DOS (Denial of Service) útoku. Z tohoto důvodu se zavádí určitá pravidla, po určitém množství žádostí by crawler měl svou činnost pozastavit. Dále vývojáři užívají speciálních souborů, zvaných *robots.txt*, kde jsou vymezeny podadresáře, které nesmí crawler navštívit. [10]

Pokud by crawler těmto požadavkům nevyhověl, může být jeho IP blokována, jeho user-agent blokován, nebo může být „chycen“ do *spider trap*. Jedná se o záměrné zmatení crawleru za účelem odvedení pozornosti od hlavní adresářové struktury, buď pomocí zacyklení v adresářové struktuře, nebo přetížením lexikálního analyzéru, které by vedlo k zamrazení aplikace.

2 Metodologie OWASP

2.1 OWASP top 10

V první kapitole byl popsán postup penetračního testu webové aplikace, podle metodologie OWASP. V této kapitole bude popsán žebříček deseti nejvýznamnějších zranitelností webových aplikací a v kapitole 3.2 bude popsáno doporučení pro vývoj webové aplikace.

OWASP top 10 byl naposledy vydán v roce 2017, a bylo vytvořeno na základě dat od více jak čtyřiceti firem, a je tedy zatím největším top 10, co se spolupráce firem s organizací týče. Žebříček je sestaven na základě rizikového faktoru, který udává pravděpodobnost a dopad daného rizika.

Při porovnání s vydáním z roku 2013 je zřejmé, že byly přidány tři nové zranitelnosti, a to A4:2017 – XML externí entity (XXE), A8:2017 – Nebezpečná deserializace a A10:2017 – Nedostatek loggingu a monitoringu. Dále byly sloučeny zranitelnosti A4 – Nebezpečné přímé reference na objekt a A7 – Chybějící kontrola přístupu na úrovni funkcí v obecnější zranitelnost A5:2017 – Chybná kontrola přístupu. Dvě zranitelnosti z vydání 2013: A8 – Cross Site Request Forgery (CSRF) společně s A10 – Nevalidované přesměrování a odesílání nebyly zahrnuty ve verzi 2017, stále se však jedná o nebezpečné zranitelnosti, pouze jejich četnost je na ústupu. Desítka je tedy následující [6]

1. Injekce:

   Jedná se o chybu, která vzniká při špatné validaci vstupů umožňující útočníkovi zasílat nebo vkládat škodlivá data. Nejčastější formou injekce je SQL injekce, kde se většinou vkládají logické operátory do špatně sanitovaných vstupních polí.

   Aplikace je náchylná na tento útok, pokud nevaliduje vstupní data. Společně s ORM (Object-Relational Mapping) jsou využity nebezpečné vstupy k získání citlivých dat nebo zřetězením nebezpečného vstupu tak, aby byla zachována původní struktura a data útočníka.

   Mějme tabulku účty, kde jsou obsažena data jméno, příjmení, bankovníÚčet a telefonníČíslo. Tato data jsou seřazena pomocí čísla id, které zadává uživatel a doplňuje tento příkaz:

   *SELECT * FROM účty WHERE id =*

   Útočník pak může doplnit logický operátor „or“, čímž mu budou zdostupněny veškeré informace v tabulce. Příkaz bude tedy po doplnění vypadat ná-

sledovně:

*SELECT * FROM účty WHERE id = 1.5 or 1=1;*

Prevence je možná pomocí implementace vhodných API (Application Programming Interface), které minimalizují potřebu využití interpretů, využití SQL parametru LIMIT zabraňující masivnímu výpisu dat, a správná sanitace dat. [6]

2. Nekorektní funkce autentizace:

   Tato chyba je mnohem obecnější, může se jednat o chybu správy relace nebo autentizačního mechanismu jako takového. K dnešním autentizačním mechanismům patří dříve zmíněné cookies, které pokud útočník ukradne a změní, může je pak zneužít k autentizaci. Jelikož útočníkům jsou volně dostupné seznamy známých uživatelských jmen a hesel, tak jejich jediným úkolem je zjistit, zdali autentizační mechanismy skutečně fungují, jak mají.

   K nejčastějším útokům patří tzv. *credential stuffing*, kdy útočník využívá seznamů jmen a hesel, dále klasický brute force, zneužití slabých hesel (toor, admin), hesla uložená v plain text podobě nebo pouze slabě hashovány.

   Aplikace by tedy měly, pokud možno, využívat dvou-fázovou autentizaci, nevydávat produkt s výchozími hesly, testovat hesla oproti databázím slabých hesel, dodržovat standardy a další. [6]

3. Odhalení citlivých dat:

   Díky této chybě může útočník odposlouchávat přenos citlivých dat, často se totiž jedná o přenos dat přes nezabezpečené kanály. Nevyužitím kryptograficky ochráněných sourozenců známých protokolů jako HTTP, SMTP a FTP je přenos uskutečněn v plaintext. Tato skutečnost umožňuje útočníkovi provedení MITM (Man in the Middle).

   Další chybou bývá nedodržování doporučení ohledně ukládání citlivých dat. Nejčastější chybou je nevyužití přiměřených kryptografických algoritmů, např. hashování hesel pomocí algoritmu MD5, využívání výchozích kryptografických klíčů, či data ukládána v plain text podobě. [6]

   Dopad těchto chyb může být v podobě sankcí nedodržení GDPR (General Data Protection Regulation), ztráta důvěry u zákazníků a možnost útočníka zneužít získaná data k cíleným útokům jako je *spear phishing*.

   V rámci prevence by neměla být citlivá data ukládána dlouhodobě, jen pokud je to nezbytně nutné. V takovém případě by měla být dodržena doporučení podle PCI DSS (Payment Card Industry Data Security Standard). Zašifrování všech citlivých dat za pomocí moderních algoritmů podle standardů je nutností, stejně tak jako správný key management, což je detailněji

obsaženo uvnitř doporučení OWASP ohledně proaktivní ochrany dat. [11]

4. XML externí entity (XXE):

   Jedná se o zranitelnost, při které útočník nahraje XML soubor do zranitelné aplikace. V tomto souboru může být obsažen škodlivý prvek využívající slabin XML procesoru umožňující odkaz na externí referenci, slabin kódu, integrace systému nebo jiných.

   Aplikace je náchylná, pokud povoluje nahrávání nevalidovaných XML souborů, nebo přímo vkládání XML. Další chybou je vkládání nevalidovaných dat do souboru. Náchylnost na XXE zároveň může vést k útokům DOS (Denial of Service), jako je *The billion laughs* útok. [6]

   Tento útok patří do rodiny XML bomb zneužívající vlastnosti parserů takovým způsobem, že jsou interpretovány jako validní, ale způsobují výpadky nebo pozastavení systému. *Billion laughs* využívá DTD (Document type Definition), která umožňuje uživateli definovat entity v dokumentu, které se chovají jako makro v jazyce C. Tedy, pokud někdo deklaruje entitu zákazník mající ve své deklaraci uživatelem deklarovaný element id a jméno, tak při volání zákazníka se pole vyplní id a jméno.

   Tohoto principu zneužívá *Billion laughs* útok. Deklaruje 9 do sebe zanořených entit, kde každá je volána vícekrát. Takto se jeden blok entity bude rapidně rozšiřovat až bude zabírat skoro 3Gb paměti.

   Ukázka entity *lol9*, obsahující 9 entit *lol8*, ty obsahují 9 entit *lol7* až do entity *lol2* obsahující 9 entit *lol*. [12]

   <!ENTITYlol9„&lol8;&lol8;&lol8:&lol8;&lol8;&lol8;&lol8:&lol8;&lol8“>

   Aplikaci lze co nejefektivněji chránit pomocí nástrojů pro statickou analýzu kódu, nebo naopak nástroji pro dynamickou analýzu. Dále je nezbytné validovat vstupní data a nepovolit uživateli definovat vlastní Document type Definition, implementace *whitelistingu*, aktualizace XML procesorů a další.

5. Chybná kontrola přístupu:

   Podobně jako tomu je u operačních systémů, využívají i webové aplikace metod kontroly přístupu. Takto je zamezen běžným uživatelům přístup k částem aplikace, které jsou například vyhrazeny pro administrátora. Pokud se na této úrovni vyskytne chyba, může ji útočník zneužít ke zjištění citlivých údajů o uživatelích aplikace, ničit data a jiné.

   Možností útoku a kompromitace má útočník mnoho. Lze modifikovat url za účelem získání administrátorského přístupu jakožto běžný, nebo dokonce neautentizovaný uživatel. Dále může existovat možnost změny primárního klíče

a tím možnost editace účtu uživatele a další.

Nástroje pro statickou a dynamickou analýzu nemusí tyto zranitelnosti zachytit, proto se doporučuje hlavně manuální testování.

Obrana proti těmto útokům spočívá hlavně v prevenci, tedy v dobře nastavených přístupových pravidlech, zakazování přístupu k neveřejným částem aplikace, správná implementace přístupových mechanismů a jejich opakované použití. Kromě toho se doporučuje zakázat directory listing, logovat nepovolené pokusy o přístup a v neposlední řadě notifikovat administrátora. [6]

6. Špatná konfigurace bezpečnostních politik

   Základní technikou útočníků je brute force výchozích hesel, uživatelských jmen, nebo dokonce aplikací, které jsou na serveru předinstalované. Tyto útoky jim mohou umožnit přístup do systému, nebo získání informací o jeho fungování.

   Tyto chyby vznikají, pokud administrátoři spoléhají pouze na výchozí konfigurace a neprovedli dostatečnou úpravu aplikace. Úpravami mohou být změny hesel, segmentace aplikace na části s rozdílnými přístupovými právy nebo odstraňování nepoužitých částí aplikace. Zároveň jsou administrátoři povinni aktualizovat všechny užité knihovny a služby, aby takto předešli možné exploitaci známých zranitelností, stejně tak nesmějí využívat zranitelných služeb či knihoven (viz bod 10).

   Příkladem chyby může být detailní výpis chyby, nebo dostupnost zdrojových kódů aplikace, které útočník po stažení dekompiluje a zpětně sestaví podobu aplikace, dávající možnost k white-box analýze. [6]

7. Cross Site Scripting (XSS):

   Jedná se o druhou nejčastěji se vyskytující chybu na OWASP top 10, bližší čísla v kapitole 3.3. Tato chyba vzniká opět při nedostatečné kontrole vstupních dat, což umožňuje útočníkovi vkládat JavaScript nebo HTML do webové aplikace. XSS lze rozdělit na tři typy:

   - Reflected XSS vzniká, když aplikace přijímá od uživatele data, která se následně zobrazí na webové stránce (tedy se vloží do HTML aplikace). Tato skutečnost může vést ke vkládání jednoduchých JavaScript příkazů, HTML nebo CSS. Jelikož tato data budou zpracována serverem, tak bude škodlivý kód vykonán. Díky těmto technikám může útočník zasílat URL odkazující na stránku (i v rámci aplikace), která po načtení pošle útočníkovi cookie uživatele.

   - Stored XSS (tj. ukládané XSS) typicky nastává v aplikacích typu diskuzních fór, chatovacích aplikací apod. Základním požadavkem je uložení

dat, která jsou vložena uživatelem. Stejně jako v předešlém příkladu může útočník krást cookies. Tentokrát ale získá cookies všech, kteří navštíví fórum obsahující jeho nebezpečný komentář. [6]

- DOM based XSS nastává při dynamickém vkládání dat, která jsou pod kontrolou útočníka. Není tedy měněn obsah stránky obsažen v HTTP odpovědi jako takové, pouze kód na straně klienta bude vykonán neočekávaným způsobem z důvodu modifikace DOM prostředí aplikace. [13]

Prevence spočívá především v oddělení vkládaných dat od aktivní části aplikace a k tomuto účelu mohou sloužit knihovny typu *Ruby on rails* nebo *ReactJS*. Je však třeba mít na vědomí, že žádná z těchto prevenčních knihoven není bezchybná a při špatné implementaci existuje možnost vytvoření nových zranitelností. Dále je třeba se chránit proti nedůvěryhodným HTTP hlavičkám, tímto bude z velké části zamezeno *reflected* a *stored* typům útoku. [6]

Častým typem payload bývá využití *img* objektu:

“><img src=# onerror=alert(‘XSS’)>

Počáteční uvozovky ukončí načítání očekávaného vstupu od uživatele, zatímco znak > ukončí vstupový tag. Poté je vytvořena fotka, jejíž zdroj bude vždy vracet error. Tato skutečnost je využita atributem onerror, který pro pouhé otestování zranitelnosti zavolá alert(‘XSS’). Uživateli bude zobrazeno dialogové okno s tímto obsahem.

8. Nebezpečná deserializace:

   Serializací rozumíme proces, kdy je datový objekt převeden do formátu, který lze uložit na disk nebo zaslat přes síť. Tento formát může být binární nebo textový (XML, JSON (JavaScript Object Notation) , YAML (YAML Ain’t Markup Language) apod.). Deserializace je tedy procesem opačným, převádějící serializovaná data do datových objektů. Tato operace ve své podstatě není nebezpečná, ovšem deserializací neověřených souborů může zapříčinit vykonání cizího kódu. [14]

   Aplikace je zpravidla náchylná na tento typ útoku, pokud zpracovává nedůvěryhodná data předána útočníkem. Toto zpracování může vést k modifikaci logiky aplikace nebo k dříve zmíněnému vykonání kódu. Dalším typem je útok na přístupová práva, kde jsou užity existující datové struktury, ale jejich obsah je změněn.

Základní metodou ochrany je metoda prevence, tedy nepovolovat zpracování serializovaných dat od nedůvěryhodných zdrojů, tj. povolit data pouze od aplikace nebo zevnitř organizace. Krom tohoto mohou vývojáři aplikovat kontroly integrity dat, izolaci a spouštění kódu v prostředí s omezenými právy, monitoring deserializace a varovat při opakovaných pokusech, logovat deserializační chyby apod. [6]

9. Užívání komponent se známou zranitelností:

   Tato chyba nastává především z důvodu vývoje velmi komplexních aplikací, které se skládají z mnoha externích komponent (skriptů, knihoven apod.). Vývojáři takto ztrácí přehled nad zranitelnostmi v daných knihovnách a zdali jsou využívány aktuální verze. Využití skenerů jako *retire.js* pomáhá s detekcí, ovšem nedokazuje exploitabilitu systému.

   Mezi typické chyby vedoucí k exploitaci patří: neznalost verze knihovny-/balíčku nebo verze knihoven/balíčků obsažených, využívání zranitelné verze operačního systému, API, serverů nebo databázových manažerů, neprovádění pravidelného vulnerability assesmentu, nalezení, ale neopravení chyb a netestování kompatibility aktualizovaných systémů. [6]

10. Nedostatek loggingu a monitoringu:

    Tato zranitelnost je základem snad každého kybernetického útoku. Útočníci spoléhají na nedostatek monitoringu a delší dobu odezvy, aby byli schopni dosáhnout svých cílů. Vhodným způsobem otestování logování, je zkontrolování logů po penetračním testu.

    Kdy tedy hovoříme o nedostatečném loggingu informací? Pokud nejsou monitorovány a zapisovány neúspěšné pokusy o přihlášení, přihlášení nebo důležité transakce. Pokud se varování nevyskytují v logu nebo se nevyskytují v přijatelné míře. Pokud logy nejsou uloženy lokálně, nejsou logovány aplikace a ani z důvodu sledování chování, aplikace nedokáže zachytit nebo upozornit na aktivní pokus o útok.

    Zabránit této zranitelnosti by mělo být prioritním krokem při bezpečnostním auditu, nebo bezpečnostní analýze. Kromě napravení dříve zmíněných chyb, je důležité sepsat a důsledně trénovat/dodržovat tzv. Incident response policy. [6]

2.2 Doporučení pro vývoj bezpečné aplikace

V této kapitole budou sepsána doporučení podle OWASP Application Security Verification Standard (Standard pro verifikaci bezpečnosti aplikací). Standard se skládá ze 14 hlavních kapitol, ovšem pro účely této práce bude čerpáno z prvních pěti kapitol:

V1 Požadavky na architekturu, design a modelování hrozeb

V2 Požadavky na verifikaci autentizace

V3 Požadavky na správu relací

V4 Požadavky na ověření přístupových práv

V5 Požadavky na validaci, sanitaci a zakódování vstupů

Zbývajících devět kapitol je již nad rámec této práce. Postupně se jedná o kapitoly:

V6 Požadavky na uložené kryptografické prostředky

V7 Požadavky na zpracování errorů a logování

V8 Požadavky na ověření ochrany dat

V9 Požadavky na ověření komunikací

V10 Požadavky na ověření nebezpečného kódu

V11 Požadavky na ověření business logic

V12 Požadavky na ověření zdrojů a souborů

V13 Požadavky na ověření API a webových služeb

V14 Požadavky na ověření konfigurací

Podle kapitoly **V1** je jednou ze zásadních složek bezpečnosti zajistit fluiditu jak ve vnímání, tak v implementaci bezpečnostních mechanismů. Bezpečnost je komplexní neustále se vyvíjející pojem, a proto rigidita nemůže vést k úspěchu.

Důraz je kladen na praktiku známou jako *security by design*, tedy myslet na zabezpečení aplikace již ve fázi plánování a aktivně s ní počítat, nikoliv ji implementovat až zpětně. S tímto souvisí i bezpečný životní cyklus aplikace, tedy aby bezpečnost softwaru nebyla opomenuta v kterékoliv z fází vývoje. Dále je důležité neustále myslet na čtyři základní kameny bezpečnosti: *důvěrnost*, *integrita*, *dostupnost*, *nepopíratelnost* a *dodatečně ochrana soukromí*. [15]

Kromě základních kamenů bezpečnosti jsou v kapitole **V1** definované i doporučení týkající se návrhu aplikace: zajistit, že všechny autentizační mechanismy jsou stejně silné; předem určit, jak se kterými vstupy pracovat; implementovat kryptografii ihned a ne retroaktivně. Na toto navazuje i správný management klíčů; validovat zdrojový kód, používat verzovací systémy, kde lze dohledat provedení změn a jejich autora. [15]

Kapitola **V2** se zabývá verifikací autentizačních mechanismů. Oproti dřívějším vydáním je tato kapitola do jisté míry psána v souladu se standardem NIST 800-63, čímž se mění postoj k autentizaci jako takové.

Zásadní je zajistit bezpečnou politiku hesel. Zpravidla se doporučuje užívat spíše tzv. *passphrase* namísto *password*, tedy tajemství ve formě věty. Tímto se zmenšuje jak pravděpodobnost zapomenutí hesla, tak jeho obsažení ve slovnících hesel. Kromě toho by hesla měla mít alespoň 12 znaků, ale nikdy ne více než 128 znaků. Zároveň je nutné po každé registraci či změně jej ověřit oproti databázi prolomených hesel. Nestandardně se i doporučuje nenutit uživatele ke změně hesla v pravidelných intervalech.

ASVS dále doporučuje vývojářům, aby neomezovali nejen své uživatele, ale zároveň i sebe, autentizačními metodami. Čím více jich bude implementováno, tím jednodušeji se bude reagovat na změny v doporučení. Navíc, tímto může být dána uživateli možnost užití média, které již má a aktivně používá, jako například usb token.

Poslední doporučení se týkají ukládání a využití autentizačních mechanismů jiných aplikací. Ukládat se má zásadně v hashované formě s využitím kryptografické soli o délce alespoň 32 bitů. Z externích autentizátorů se upřednostňuje hardwarová implementace, ale aplikace typu Google Authenticator nebo RB (Raiffeisen Bank) klíč jsou také vhodné. [15]

Následující kapitola (**V3**) pojednává o správě relací, kde je ihned vznesen základní požadavek; relace nemůže být útočníkem uhodnutelná a jedna relace může náležet pouze jednomu uživateli. Zároveň pokud nějáká relace již není nutná, nebo uběhla stanovená doba, bude násilně ukončena.

Vývojáři by měli vždy využívat bezpečného přenosového kanálu, ať už TLS (Transport Layer Security) nebo jiné. Pro každou úspěšnou autentizaci musí být vytvořena nová relace a bezpečně ukládat a generovat relační tokeny.

Pokud je tokenem cookie, pak musí být vždy užito speciálních atributů HttpOnly, SameSite a Secure, čímž se mitiguje možnost využití CSRF (Cross Site Request Forgery) útoku.

A nakonec je třeba ověřovat, zdali byla navázána plnohodnotná relace, jestli se jedná o změnu hesla, či čekání na potvrzení multifaktorové autentizace. Zároveň by se nikdy neměly sdílet objekty uživatelů skrz tyto tři stavy, protože by tato skutečnost mohla umožnit útočníkům po otevření neplnohodnotné relace změnit údaje obsažené v objektu, čímž převezmou účet od uživatele. [15]

V4 je zaměřeno na přístupová práva, jak by měla fungovat a best-practice. Opět se zde projevují dva základní požadavky: *Deny by default* - nový uživatel by měl mít

minimální nebo žádná práva; *Princip nejnižších práv* - uživatel může přistupovat pouze k těm zdrojům, ke kterým je specificky autorizován.

Dále je kladen důraz na mechanismy jako takové. Je třeba ověřit, zda uživatel nemůže změnit data užita ke kontrole přístupu a navíc, jestli přístupové mechanismy pracují na důvěryhodné vrstvě aplikace (především se předchází manipulaci dat na straně klienta). Kromě tohoto je třeba implementovat vhodných mechanismů k zabránění CSRF, zajistit multifaktorovu autentizaci pro administraci a zakázat procházení adresářů, a to včetně zákazu přístupu k souborům obsahujícím metadata, jako jsou soubory .git, .svn apod.

Jako častý útok je uveden IDOR (Insecure Direct Object Reference), kdy útočník může přistupovat k datům, které nespadají pod jeho přístupová práva. Kromě toho je může i modifikovat, což může vést k dalším exploitacím. [15]

Pro účely této práce je poslední kapitolou **V5** pojednávající o validaci, sanitaci a zakódování vstupů. Tato kapitola je velmi důležitá, a to z toho důvodu, že zranitelnosti XSS a SQL injekce jsou jedny z nejčastějších útoků (více v kapitole 2.3). Zajištění validace vstupů tedy vede ke zvýšení bezpečnosti aplikace.

Základním doporučením je užívání tzv. *whitelistů*, tedy povolit pouze specifické vstupy, stejně tak kontrolovat délku vstupu může značně snížit útoky pomocí injekce. [15]

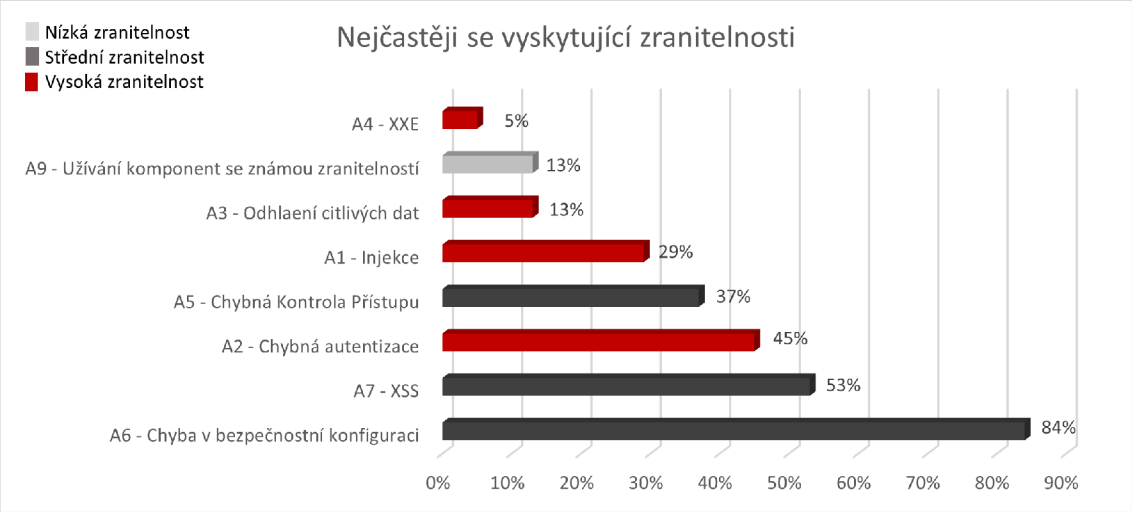
Rozšířením výše zmíněných praktik je ochrana parametrů HTTP žádostí před nebezpečnou modifikací, ochrana před změnou vlastností parametrů v aplikaci samotné, zajištění přesměrování na bezpečné stránky po rozkliknutí URL a účelná typizace vstupních polí, a to včetně jejich délky, typu vstupu a struktury. Dalším důležitým bodem je *sandboxing* nedůvěryhodného vstupu, tedy jeho bezpečné vykonání a ověření, pokud nelze dostatečně verifikovat.

Nakonec je třeba vhodné kódování výstupu tak, aby sedělo do kontextu aplikace. Např. zvážit, jestli je zapotřebí UTF-8 znaků, nebo postačí ASCII apod., zároveň je třeba zajistit, aby bylo zachováno stejné kódování na výstupu jako na vstupu. Pod problematiku validace spadá i deserializace. Tento problém a způsoby ochrany proti němu jsou popsány v kapitole 2.1 bod 8. [15]

2.3 Statistický pohled

V této kapitole bude hlavním tématem statistický pohled na zranitelnosti a útoky na webové aplikace. Rok co rok se objevují nové zranitelnosti a čísla provedených útoků kolísají. Jak tomu je ale přímo s webovými aplikacemi? Jsou s každým rokem bezpečnější, nebo snad stagnuje aplikace bezpečnostních praktik?

První zaměření bude v poněkud menším měřítku, ale spojené s dříve probraným OWASP top 10 (viz 2.1). Data byla získána z 35 reálně nasazených a funkčních aplikací, společností PT Security (data za rok 2019):



Obr. 2.1: Převzato z Web Application Vulnerabilities and Threats: Statistics for 2019 [16]

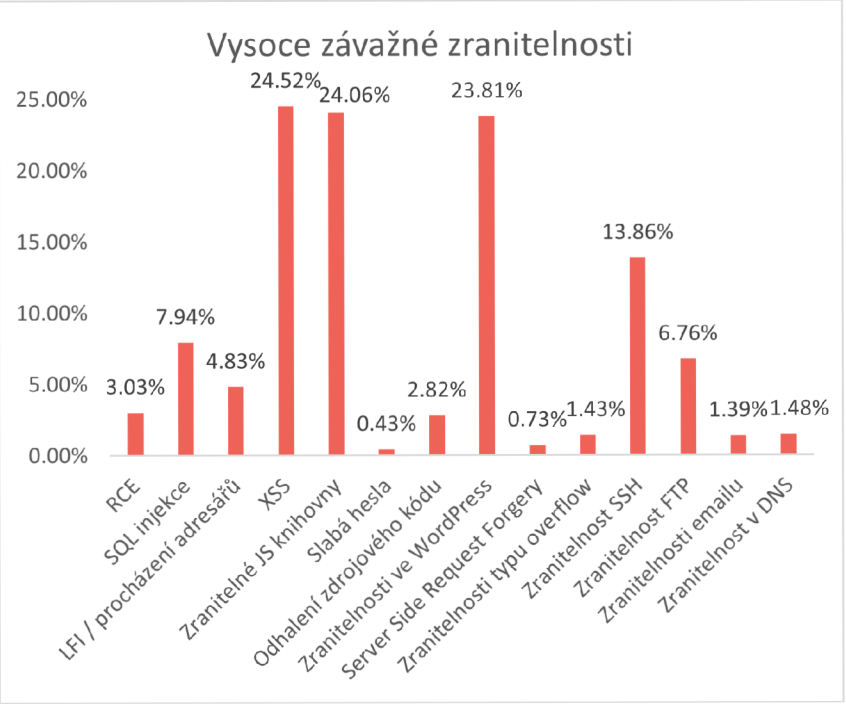
Lze vidět, že nejčastější zranitelností je chyba v bezpečnostní konfiguraci. Z tohoto lze vyvodit, že doporučení nejsou dodržována v dostatečné míře a na toto se vážou další komplikace, jako je druhá nejčastěji se vyskytující zranitelnost, a tou je XSS.

V reportu je dále psáno „*Z našich zkušeností víme, že většina zranitelností webů jsou způsobena chybami ve zdrojovém kódu aplikace.*“. [16] To může poukazovat na nedostatečné množství white-box testování nebo code-review. Dále je uvedeno, že při white-box testování (na 11 aplikacích) byly zjištěny následující zranitelnosti: XML External Entities (XXE) ve 100% případů, Injekce v 76% případů a XSS v 67% případů.

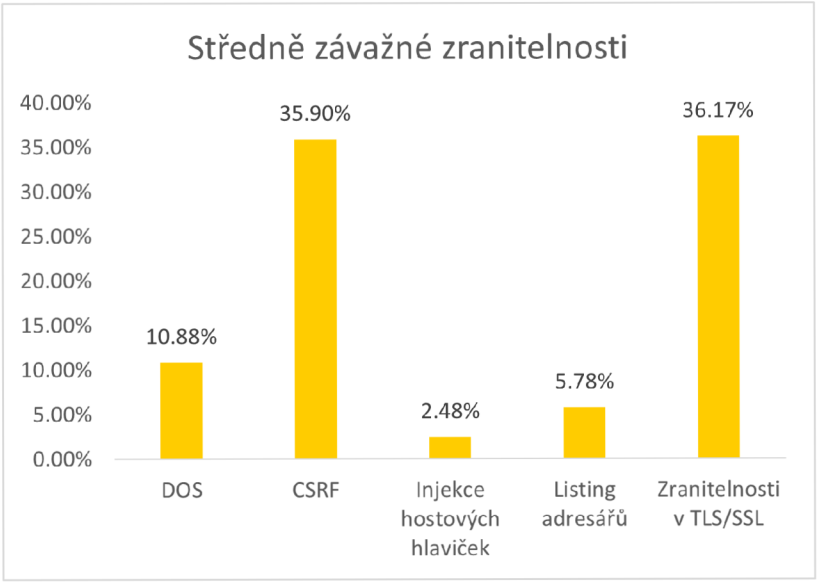
Ovšem lze říci, že počet nalezených zranitelností procentuálně klesl oproti roku 2018. Problémem je však malý počet testovaných aplikací.

Následující report od společnosti Acunetix za rok 2020 byl sepsán na základě dat získaných ze vzorku o velikosti 5000 webových aplikací (úmyslně zranitelné nebyly započítány) v časovém rozmezí od března 2019 do února 2020.

Nalezené zranitelnosti byly rozděleny do dvou skupin, a to na vysoce závažné a středně závažné:



Obr. 2.2: Převzato z Web Application Vulnerability Report 2020 [17]



Obr. 2.3: Převzato z Web Application Vulnerability Report 2020 [17]

Z obr. 2.2 je zřejmé, že nejčastěji se vyskytující zranitelností je XSS. Tomuto faktu může dopomáhat i skutečnost, že existují velmi časté chyby v knihovnách JavaScriptu, které pak mohou i souviset s validací vstupu, nebo vést k XSS.

Vysoce závažné zranitelnosti, jako XSS, SSRF (Server Side Request Forgery) apod. mají potenciál k vykonání vysoce účinných útoků, které mohou kompromitovat server, databázi nebo získat administrátorská privilegia, a to bez spolupráce uživatele, nebo bez pomoci externích útočníkem neovlivnitelných faktorů.

Středně závažné zranitelnosti (2.3) mohou dosáhnout potenciálně stejných cílů jako vysoce zranitelné, ovšem potřebují uživatelský přístup do systému, nevědomou spolupráci uživatele nebo jiné faktory, které jsou mimo kontrolu útočníka.[17]

Pokud se blíže zaměříme na data ohledně zranitelností XSS a SQL injection získaná z databáze zranitelností NIST, lze vyčíst znepokojivé hodnoty.

Zranitelnost XSS je na vzestupu, oproti předešlým dvěma rokům, kde NIST ohlašuje 1090 ze 16511 (2018) a 1006 ze 17305 (2019) se za rok 2020, ke dni 1.12, objevilo 1161 XSS zranitelností z celkového počtu 16807. [18]

Lze pozorovat, že celkový počet zranitelností byl snížen, ovšem vkládání škodlivého JavaScriptu se stále zvyšuje. Tato skutečnost může souviset s dříve zmíněnými JS knihovnami. Mezi developery se neustále rozšiřují nové knihovny, které sice mohou být užitečné, ovšem často obsahují chyby.

Oproti XSS je SQL injekce značně na ústupu. Ke dni 1.12.2020 se v databázi NIST vyskytuje 402 zranitelností tohoto typu z celkového počtu 16807 (2,39 %). Tato data vypadají pozitivně v porovnání s daty z roku 2018, kde z celkového počtu 16511 zranitelností jich bylo nalezeno 479 (2,90 %). Za poslední tři roky bylo SQL injekce nalezeno nejvíce v roce 2019, z celkového počtu 17305 zranitelností bylo zaznamenáno 552 zranitelností (3,19 %). [19]

Toto může značit zlepšení co se validace vstupu při výběru z databází týče. Tato data jsou dále potvrzena i předchozími daty.

3 Nástroj 8search pro listing adresářů

První ze tří programů sloužící k usnadnění manuálních penetračních testů webových aplikací je nástroj 8search, který bude blíže popsán v této kapitole.

3.1 Motivace

Motivací za vytvořením tohoto python nástroje byla důležitost fáze získávání informací. Bez dobře zmapované aplikace nelze provést kvalitní penetrační test. Jak již bylo popsáno v kapitole 1.3, *fingerprinting* aplikace je zcela prvotní fází. Jelikož tester začíná (v případě *black-box* testu) naprosto od nuly, je nezbytné, aby zjistil, jak je aplikace rozvržená a kam až může hned ze začátku dojít.

Zjištění, že jako neautentizovaný uživatel aplikace získá přístup k */admin*, značně ušetří čas a může vést k objevení mnoha dalších zranitelností již ve fázi průzkumu.

K tomuto může tester využít nástroje 8search, jehož účel je zrychlit mechanickou a monotónní práci penetračního testera při hledání dostupných částí aplikace. Primární funkcionalitou aplikace je *bruteforce* adresářů podle url, které zadá uživatel. *Bruteforce* adresářů je založený na čtení známých jmen z tzv. *wordlist*, tak jako bývá využit např. *rockyou* při lámání hesel.

Ovšem platí, že kvalita analýzy a počet zjištění závisí na kvalitě seznamu. Například v distribuci Kali Linux, která je určena pro etický hacking, jsou již obsaženy seznamy webových adresářů, se kterými byla aplikace testována.

3.2 Popis funkcionality a vývoje

Paralelně běžící programy představují pro programátora skvělý způsob, jak rozdělit a vylepšit využití procesoru v počítači. Pro účely 8search je například naprosto neefektivní, aby bruteforce probíhal pouze v hlavním procesu programu. Z tohoto důvodu byla importována knihovna *multiprocessing*, přesněji funkce *Pool*.

Pool vytvoří zadaný počet procesů, kterým se říká *workers*, tito pracovníci poté vykonávají určitou část kódu paralelně. Uživatel si může zvolit počet procesů, které bude 8search využívat, prostřednictvím přepínače *-p* nebo *–processes*. Pokud tak neučiní, bude nastavena implicitní hodnota pěti procesů. Každý *worker* dostává funkci k vykonání pomocí funkce *map_async*, opět z multiprocessing knihovny. Ta nám společně s funkcí *partial* umožňuje předávat procesům funkce s více argumenty (viz Výpis 3.1).

Výpis 3.1: Ukázka práce s pool a mapování

```
1  pool = Pool(5) #vytvoří se "Pool" pěti procesů
2  try:
3      if args.proxy: #pokud uživatel chce proxy
4        pool.map_async(partial(function, proxy=proxy),\
5                       new_list).get(999999)
6        #mapování funkce a předání argumentů
7        pool.close()
8        pool.join() #ukončení pool po vykonání
9      else:
10       pool.map_async(partial(function, proxy=None),\
11                      new_list).get(999999)
12       #stejné, ale bez proxy
13       pool.close()
14       pool.join()
15 except Exception: # chyby jsou ignorovány
16     pass
```

Kromě jednoduché kombinace url s adresáři má uživatel i možnost specifikovat koncový identifikátor, v tomto případě se tedy adresář chová jako soubor. Kód zároveň obsahuje krátký seznam obvyklých koncovek souborů pro webové aplikace. Uživatel může zadat i možnost zkoušení všech koncovek se všemi slovy v seznamu. Tento seznam si může nechat vypsat pomocí přepínače *-ls* nebo *–listext*. Využití této funkce komunikuje uživatel prostřednictvím přepínače *-ext* nebo *-–extensions* a následným zadáním požadované koncovky, pokud si přeje otestovat všechny, zadá *all*.

Značný problém s paralelně běžícím programem je však s jeho předčasným ukončením, standardně pomocí kláves *Ctrl+C*. V implementaci knihovny pro python verze 3 a výše se vyskytl problém se zastavením programu podle libosti uživatele. Vývojáři pro účely předčasného ukončení určili funkci *pool.terminate()*, bohužel se ukázalo, že tato možnost není úplně funkční. Proto bylo nutné přejít k možnosti násilného ukončení procesu využitím funkce *os.kill* z knihovny os a zasláním signálu (pomocí knihovny *sig*) podle nativního systému: *sig.SIGTERM* pro Linuxové systémy, *sig.CTRL_BRK_EVENT* pro Windows (viz Výpis 3.2).

Výpis 3.2: Funkce ukončující kód dle systému

```
1 def kill_sys(): # funkce pro určení systému a zabití procesu
2     if sys.platform.startswith('win32'):
3         os.kill(os.getpid(), signal.CTRL_BREAK_EVENT)
4     elif sys.platform.startswith('linux'):
5         os.kill(os.getpid(), signal.SIGTERM)
```

Program dále obsahuje možnost ignorování přesměrování nebo vypisování i přesměrovaných částí stránky. Uživatel si zvolí, pokud chce plný výpis HTTP response kódů (krom 404) pomocí přepínače *-re* nebo *–redirects*.

Se zasláním velkého množství žádostí v krátkém časovém úseku se pojí potenciální riziko ukončení spojení firewallem, nebo získání pozornosti administrátora či IDS. Z tohoto důvodu byly do 8search přidány i maskovací funkce využívající proxy a síť tor.

Jestliže si uživatel přeje zasílat žádosti přes proxy server, sděluje tuto skutečnost aplikaci pomocí přepínače *-px* nebo *–proxy*. Tato možnost je již zahrnuta v knihovně requests. [20]

Komunikací přes tor se však může uživatel skutečně dobře skrýt. K využití této funkcionality je zapotřebí mít nainstalovaný tor prohlížeč a vypnutou daemonizaci v konfiguraci. Pro zasílání žádostí přes síť tor byla využita knihovna *torrequest*. [21]

Aplikace i zde stále pracuje paralelně, ale připojení k tor síti může zpomalit její práci oproti klasickému módu. S tímto více procesovým modelem se pojí problém, kterým je využití dříve zmíněné funkce *map_async* a *partial* společně s funkcí *torrequest*. Tento wrapper na knihovnu *requests* pracuje se třídou zvanou *TorRequest*. Aby bylo možné komunikovat přes tor síť, musí být vytvořen objekt této třídy.

Funkce *map_async* nedokáže s tímto objektem pracovat, a proto musel být zvolen způsob vytvoření objektu při každém volání požadované funkce. Zároveň je užito globální proměnné typu *Value* z knihovny multiprocessing. Tímto si budeme moci zajistit, že nedojde k souběžnému přístupu více procesů ke sdílené proměnné. Proměnná se inkrementuje s každým voláním funkce. Pokud je dělitelná deseti, změní se ip adresa a program pokračuje s žádostmi dál (viz Výpis 3.3).

Výpis 3.3: Funkce ukončující kód dle systému

```
1  def T_combine(word ,url, redirect, passwd):
2      # kombinace url se slovy ze seznamu
3
4      global rund
5      # globální bezpečná proměnná
6
7      tor_obj = tor(password=passwd)
8      # vytvoření nového tor objektu
9
10     with rund.get_lock():
11     # uzamčení přístupu k rund a provedení operací
12         rund.value += 1
13         if(rund.value % 10 == 0):
14             tor_obj.reset_identity()
15
16    try:
17         new = str(url + "/") + str(word)
18         if redirect:
19             return T_request_redirs(tor_obj, new)
20             # funkce s přesměrováním
21         else:
22             return T_request_no_redirs(tor_obj, new)
23             # funkce ignorující přesměrování
24         except Exception:
25             pass
26         except KeyboardInterrupt:
27             # při zmáčknutí Ctrl+C zabij program
28             kill_sys()
```

Zájem o spojení s *tor* komunikuje uživatel programu pomocí přepínačů *-t* nebo *–tor* společně se zadáním hesla. Pokud jeho *tor* prohlížeč nemá specifikované heslo, zadá *not*.

3.3 Ukázka funkcionality

V této kapitole budou předvedeny ukázkové výstupy programu, demonstrace byla provedena na doméně www.hackthissite.org a www.youtube.com.

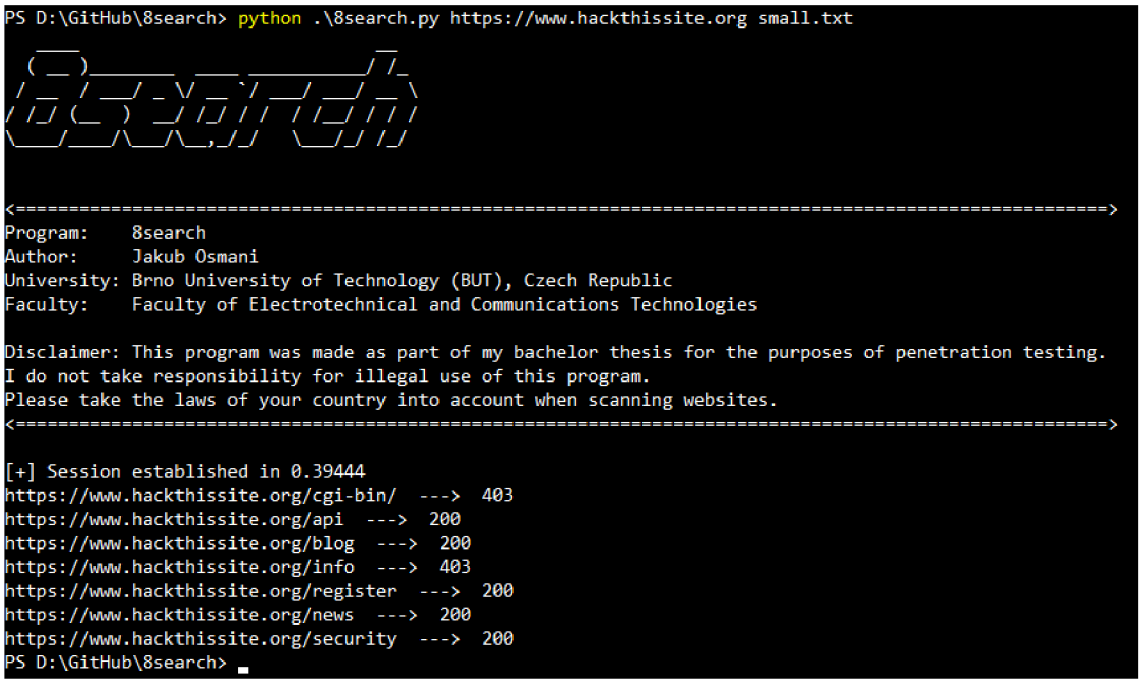
Prvním výstupem je ukázka základní práce software. Spuštění s dodáním pouze url a wordlistem nám dá jednoduchý výpis viz Obr. 3.1. Již zde lze ovšem vidět jisté nedostatky automatického testování. Adresář */api* sice zašle odpověď 200 (OK), ale po otevření v prohlížeči je tato část webu prázdná.

Výpis druhého výstupu obsahuje výsledek spuštění 8search s přepínači pro zadání koncového identifikátoru (.php) a s možností výpisu všech odpovědí, kromě 404 (přepínač *-re*). Bohužel lze pozorovat, že chyba s */api* bude propagovat skrze všechny iterace. Zároveň však Obr. 3.2 demonstruje, co tester může z těchto odpovědí vyčíst. Na základě odpovědi 200 na dotaz */index.php* lze usoudit, že server využívá jazyka *php*. Tato informace může dále posloužit při hledání chyb a exploitů.

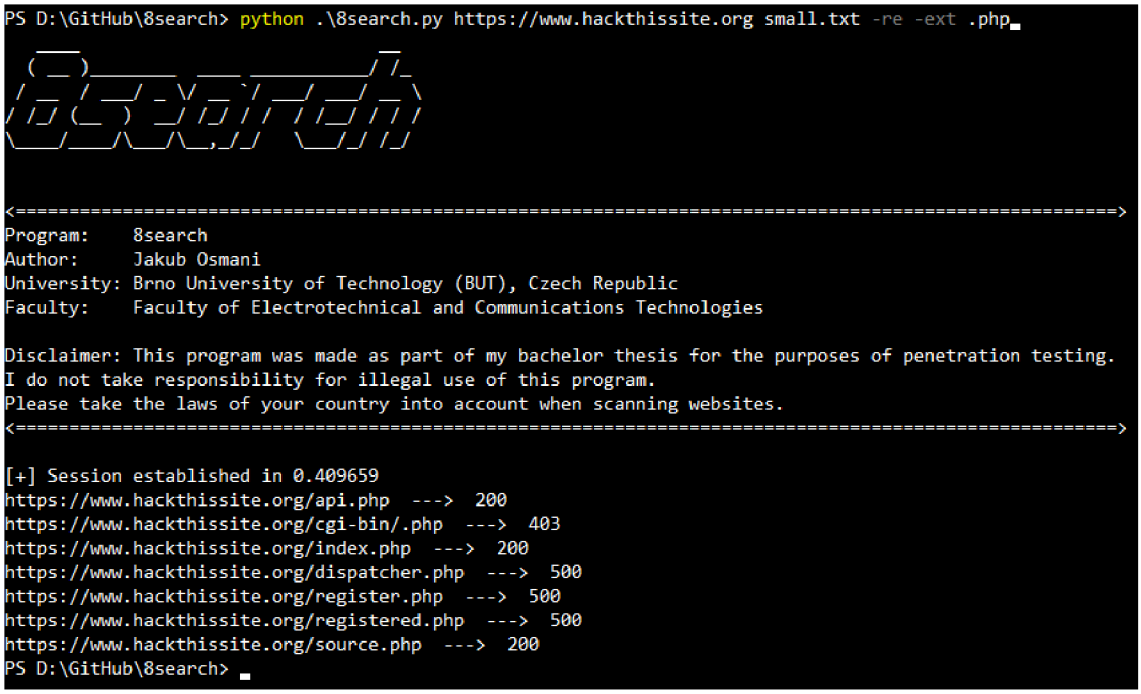
Další výstup (Obr. 3.3) ukazuje výstup při využití všech koncových identifikátorů, výpisu všech odpovědí a nastavení počtu využitých procesů na 10. Délka výstupu neumožňuje ukázku celého textu v této práci.

Poslední ukázkou je připojení do sítě tor a zasílání požadavků na youtube.com. Skutečnost, že se nejdříve musí navázat spojení s tor a poté youtube.com zpomaluje hledání; toto lze pozorovat v Obr. 3.4. Pro youtube platí, že každá záložka odpovídá uživateli jejich služby.

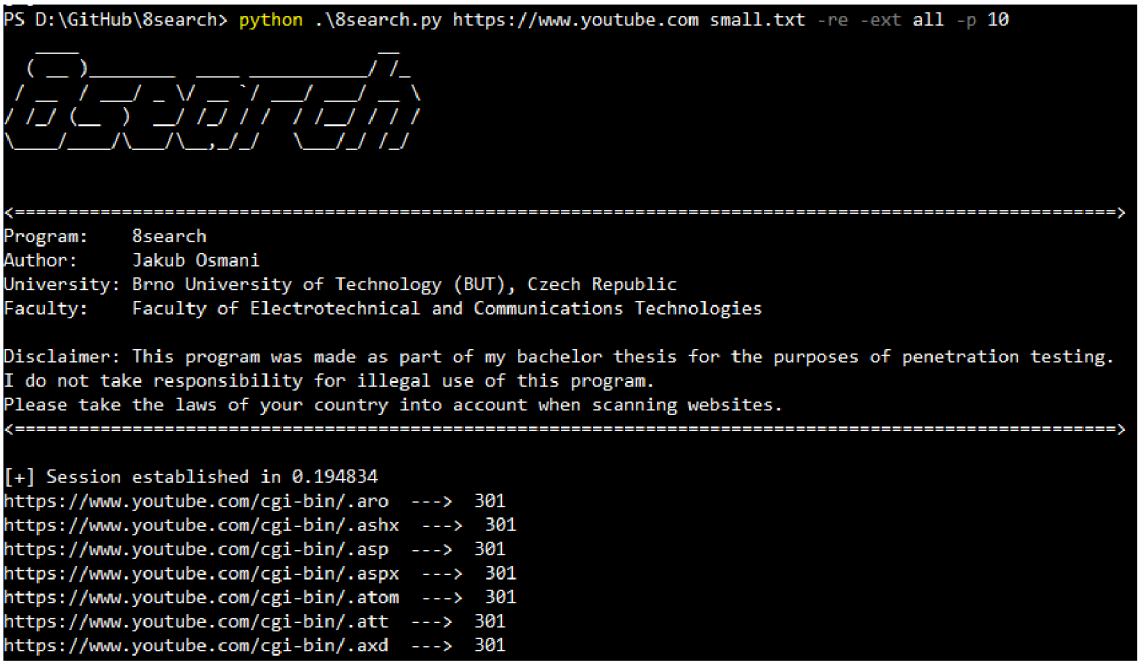
Pozn.: banner 8search se generuje pomocí knihovny pyfiglet [22]



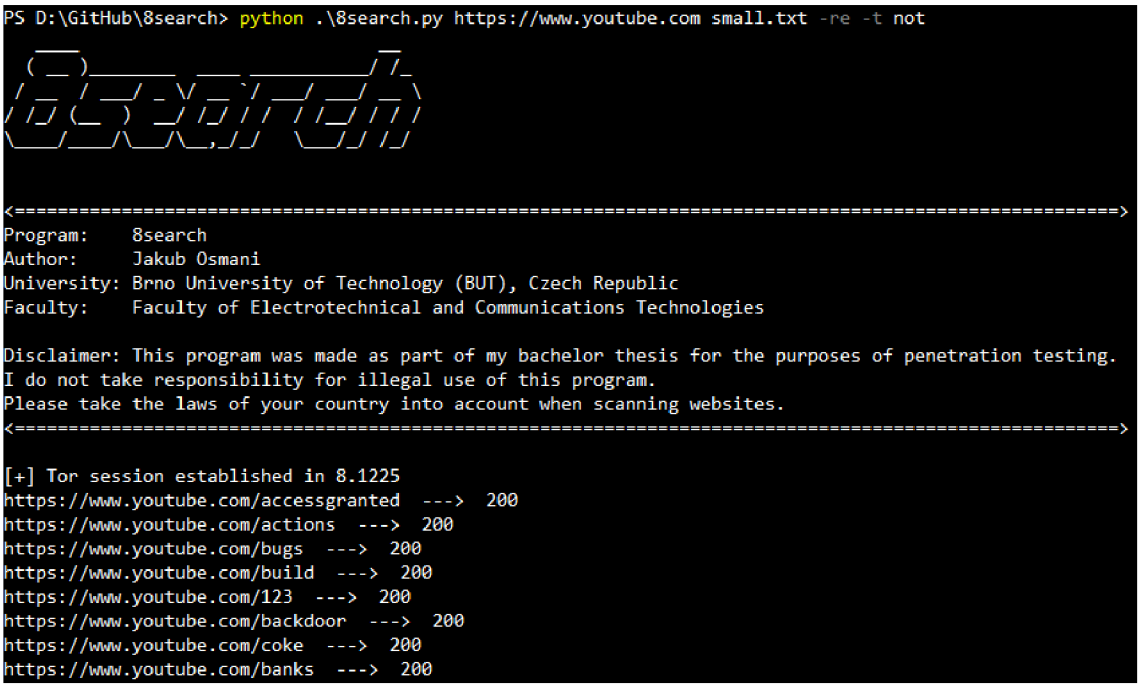
Obr. 3.1: Základní spuštění



Obr. 3.2: Hledání .php souborů a využití přepínače *-re*



Obr. 3.3: Hledání všech souborů a využití přepínačů *-re* a *-p*



Obr. 3.4: Využití služeb tor a přepínače *-re*

4 Nástroj crossget pro nalezení zranitelnosti XSS

Druhý program slouží k automatizaci nalezení některých ze zranitelností XSS (přesněji pouze *reflected* a *blind*). Umožňuje ulehčit manuální zadávání *payloadů*.

4.1 Motivace

Cílem bylo vytvořit nástroj podobný SQLmap[1], který je určený k automatizaci hledání *SQL Injection* (zmíněno v kapitole 2.1). Jelikož SQLmap je velmi robustní nástroj, za kterým stojí více vývojářů, nelze předpokládat, že by *crossget* dosahoval stejné robustnosti. Z toho důvodu je zaměřen na základní zranitelnosti *Cross Site Scripting*.

Největším problémem testera při hledání XSS je nejprve najít, které *payloady* fungují a poté zjistit proč, aby mohl v reportu i zmínit možnosti opravy. Tento nástroj pomůže právě s první částí.

A dále klasický kód obsahující *alert* (vyvolá kontextové okénko na vrchu prohlížeče se zvoleným textem) nemusí vždy fungovat. Přesněji by se dalo říci, že stránka možná kód interpretuje a vykoná, ale tato skutečnost nemusí být vizuálně prezentována testerovi. Pak se jedná o tzv. *blind* (slepé) XSS. Tento fenomén lze testovat pomocí například zachycení žádostí (GET/POST) zaslané vloženým JavaScriptem. Crossget opět obsahuje funkcionalitu pro tento typ zranitelnosti.

[1] `http://sqlmap.org/`

4.2 Popis funkcionality a vývoje

Základním stavebním kamenem této aplikace je balíček *selenium*, který umožňuje spouštět prohlížeče v tzv. *headless* režimu (bez grafického rozhraní). Krom toho poskytuje i možnost automatizace tohoto prohlížeče, tedy hýbat s myší, klikat, simulovat stisk kláves a hlavně načítat a vykonávat *client-side* kód (JavaScript). [23]

Aplikace podporuje tři prohlížeče (přesněji jejich drivery) MS Edge, Firefox (gecko driver), Google Chrome. Pro každý prohlížeč existuje zvláštní modul ve složce *modules/drivers/*. Všechny jsou nakonfigurovány do dříve zmíněného negrafického režimu. Pro konfiguraci prohlížeče Edge je však potřeba speciální balíček *msedge-selenium-tools*. [24]

Cestu k *driveru* prohlížeče může uživatel buď zadat ručně pomocí přepínače *–driver*, nebo jej obsáhnout ve speciálním konfiguračním souboru *CONFIG.xss*. Ukázka konfiguračního souboru lze vypsat pomocí *–config-eg*.

Výpis 4.1: Příklad CONFIG.xss

```
1 CHROME=/drivers/chromedriver
2 WORDLIST=/opt/payloadsAllTheThingsWordlist
3 CALLBACK=10.8.147.71:8888
```

Z výpisu 4.1 je zřejmé, že uživatel musí zadat ještě možnosti *WORDLIST*, případně i *CALLBACK* (pokud vyžaduje testování *blind xss*). Obě tyto možnosti nahrazují přepínač *–wodrlist* resp. *–callback*.

Wordlist vyžaduje cestu k souboru obsahující payloady (*payloadsAllTheThingsWordlist* je výchozí seznam, který byl složen z payloadů a jejich variací z repozitáře PayloadsAllTheThings[2]), zatímco *callback* vyžaduje IP adresu a port, ve formátu *ip:port*, na kterém bude naslouchat pro příchozí žádosti.

Přejděme k samotnému postupu hledání zranitelností. Nejprve se vyšle žádost ke specifikovanému *URL*. Zkontroluje se zda se spojení vydařilo a získané HTML se předá do konstruktoru *BeatufiulSoup* z knihovny *beautifulsoup4* pro tzv. *scraping* (získávání informací z kódu webových aplikací). [25]

Následně se pomocí funkce *find_forms_all* získají všechny html formuláře ze stránky (viz výpis 4.2). Nalezené formuláře se poté rozdělí na GET a POST formuláře. Z těch se následně získají html *input*, přesněji jejich jména.

[2] `https://github.com/swisskyrepo/PayloadsAllTheThings/tree/master/XSS%20Injection`

Výpis 4.2: Získání všech formulářů z kódu

```
try:
  page = requests.get(str(args.url)).text
except requests.exceptions.Timeout:
    print('[!] ERROR: Connection timed out!')
    sys.exit(1)
except requests.exceptions.ConnectionError:
    print('[!] ERROR: Could not connect to specified host!')
    sys.exit(1)

soup = BeautifulSoup(page, 'html.parser')
forms = find_forms_all(soup)
```

Před počátkem testování se načte obsah souboru do *listu* "wordlist"a pokud se jedná o *payloadsAllTheThingsWordlist*, tak smaže první položku. Následně se identifikuje, který *driver* se uživatel rozhodl použít, a všechny získané informace se předají dvěma (čtyřem, pokud je zapnutý *callback*) funkcím:

- *reflected_xss_get*
- *reflected_xss_post*
- *callback_xss_get*
- *callback_xss_post*

Je nutno podotknout, že pro možnost *–callback* se dynamicky generuje seznam. Využívá se předem vytvořeného seznamu *call_back_wordlist*, který se nachází uvnitř adresáře *modules/callback/*. Proměnná *IP* se nahradí zadanou adresou a portem, proměnná *POSITION* se nahradí pořadím payloadu v seznamu a proměnná *BASE64* se nahradí base64 zakódovaným payloadem[3]. Proměnné jsou obsaženy ve výpisu 4.3.

Výpis 4.3: Ukázka částí seznamu pro dynamické generování

```
<img src=http://IP/?payload=POSITION>
<object data="data:text/html;base64,BASE64"></object>
```

Přejděme tedy do funkcí pro *reflected xss*. Každý payload se dá dohromady se jménem inputu jako součást hlavičky k zaslání na původní URL, nebo na jiné, které je obsažené v proměnné *action*. Tímto se získá URL, které se předá funkci kontrolující tuto adresu v *headless* browseru (buď Chrome, Edge nebo Firefox). Automatizovaný prohlížeč zkontroluje, jestli se zobrazilo kontextové okno, uzavře jej a vrátí *0* (viz Výpis 4.4). Uživatel je pak v konzoli informován, který payload fungoval a v jakém inputu. (viz obrázek 4.1)

[3] <script>new Image().src="http://IP/?payload = POSITION ";</script>

Hlavní rozdíl mezi *get* a *post* funkcí je v názvu proměnné pro hlavičku a použité metody: *.get()* resp. *.post()*. Uživatel je navíc informován o tom, jestli daný formulář byl typu GET nebo POST.

Výpis 4.4: Funkce pro uzavření kontextového okna

```
def alert_accept(browser):
  try:
    alert = browser.switch_to.alert
    alert.accept()
    return 0
  except:
    return 1
```

Ve funkci pro kontrolu vykonání kódu se nejprve nastaví browser do negrafického módu. Poté se přesměruje na zadané URL a podle toho, jestli je nastavená možnost alert (značící že se jedná o *reflected XSS* testování), se zavolá funkce *alert_accept()* (viz 4.4), nebo počká na zaslání žádosti JavaScriptem (pro *blind XSS*).

Jelikož v seznamu pro *blind XSS* jsou i payloady, které se vykonají pouze po stisknutí klávesy, tj. *keylogger*, existuje ve funkci možnost virtuálního stisknutí klávesy. Hlavní tělo funkce je obsaženo ve výpisu 4.5.

Výpis 4.5: Funkce pro kontrolu XSS

```
browser.get(url)

sleep(0.1) # the alert may not pop quick enough

if alert:
  status = alert_accept(browser)
else:
  status = 0
  sleep(0.7) # wait for callback

if key_down:
  action = ActionChains(browser)
  action.send_keys('X').perform()
  print('[?] Pressing key \'X\' with headless browser')
  sleep(0.1) # wait for reaction to keypress

browser.close()

return status
```

Funkce pro *blind XSS* je prakticky stejná, jako pro callback. Pouze před jejím voláním se volají dvě funkce (viz Výpis 4.6). Nejprve se vytvoří dříve zmíněný dynamický seznam a poté se zavolá funkce *run_listener*. Tato funkce nejprve identifikuje se kterým operačním systémem pracuje (Windows/Linux), poté zkontroluje výchozí python verzi v systému. Pokud je jako výchozí *python2*, upozorní uživatele, že mu skončil životní cyklus a pokusí se otevřít nový terminál (*xterm* pro Linux, *cmd* pro Windows). V tomto novém terminálu se otevře specifikovaný port naslouchající na dané IP adrese, pomocí modulu *http.server* (nativní pro *python3*).

Výpis 4.6: Přípravné funkce pro blind XSS

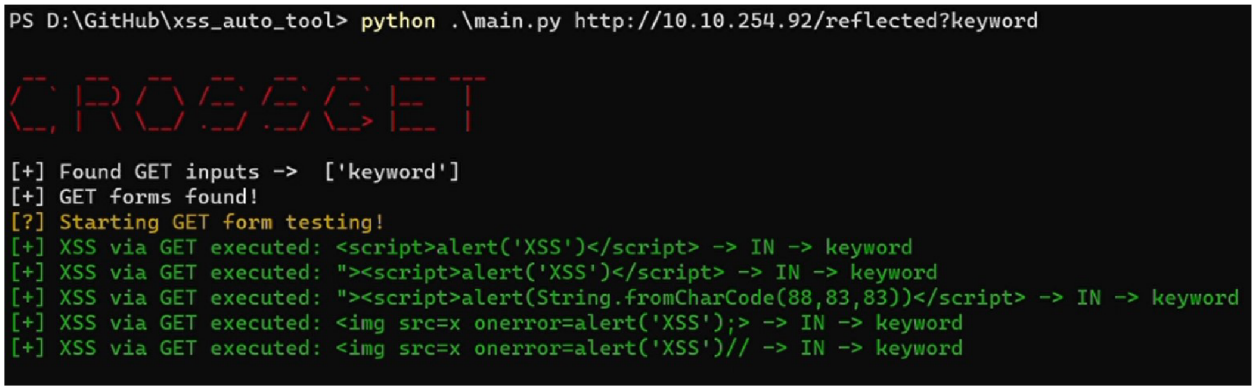
```
1 if args.callback:
2   callback_wordlist = make_wordlist(args.callback)
3   IP = args.callback.split(':')[0]
4   port = args.callback.split(':')[1]
5
6   if port == '': # port not given
7     port = '8888'
8
9   run_listener(IP, port, not args.no_color)
```

Uživatel nebude upozorněn na příchozí žádosti v původním terminálu. Skutečnost, že žádost prošla, bude viditelná pouze v novém terminálu. Výstupy v terminálech lze vidět na obrázcích 4.2 a 4.3.

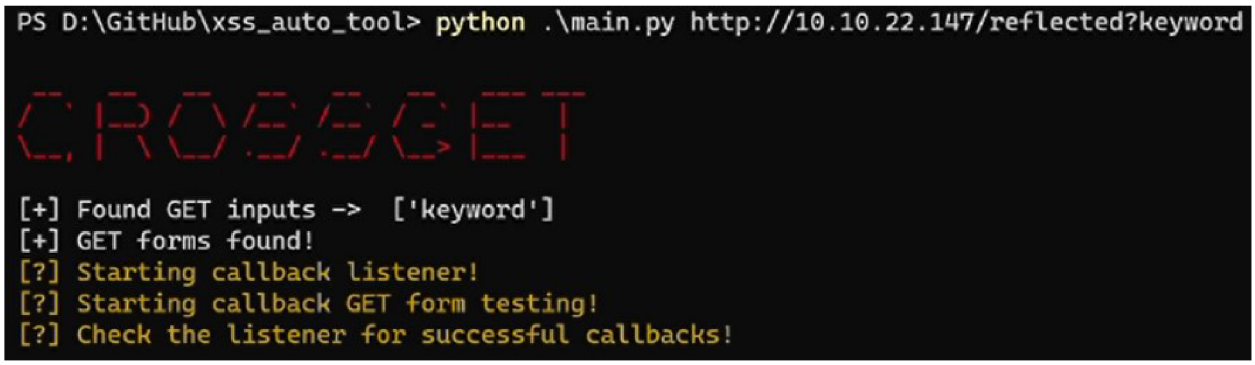
Možností pro rozšíření je stále mnoho. Například se pokusit o XSS pro jiné HTML části, jako *textbox*, anonymizační schopnosti, jako zasílání žádostí přes proxy, nebo popřípadě Tor. I pomocí *Selenia* by bylo možné pracovat s dynamicky generovanými stránkami. Bohužel rozšíření o *stored XSS* nelze jednoduše realizovat. Jakmile by se našel jeden pozitivní payload, pak se uloží a při každém opětovném načtení se opět vykoná. Z tohoto důvodu by bylo prakticky nemožné rozpoznat falešná pozitiva od skutečných pozitiv.

4.3 Ukázka funkcionality

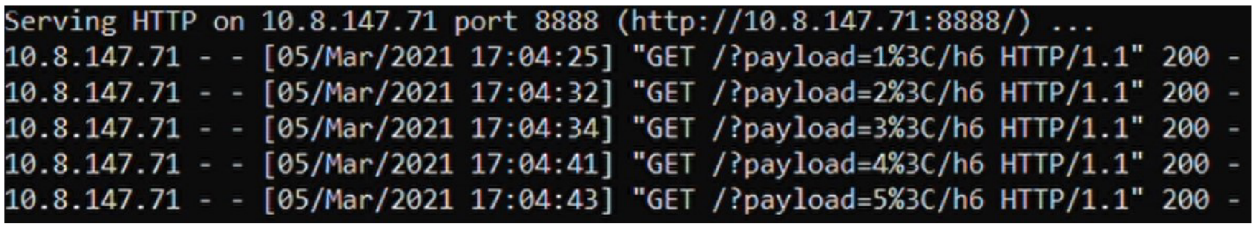
V této kapitole bude uvedeno pár ukázek výstupu při použití aplikace, *crossget*. Slouží jako doplnění k předešlé kapitole 4.2.



Obr. 4.1: Základní použití crossget



Obr. 4.2: crossget při testování *blind xss* (zkrácený výstup)



Obr. 4.3: Výstup v naslouchacím terminálu

Obr. 4.4: Výstup na operačním systému Kali Linux

5 Nástroj newspeak pro tvorbu vlastních wordlistů

Třetí nástroj (*newspeak*) podobně jako nástroj v předešlé kapitole (4) staví na tzv. *scrapingu*, tedy získávání informací z dostupného HTML zdroje webových stránek.

5.1 Motivace

Hlavní myšlenkou za tímto nástrojem, je předpoklad, že uživatel webové aplikace si vymýšlí snadno zapamatovatelné heslo. Proto si jako heslo (nebo jeho část) vybere slovo právě z webu kde se registruje. Popřípadě přidá variace, pár písmen přemění na velká a popřípadě přidá na konec jednoduchou kombinaci čísel a speciálních znaků.

Při penetračním testu je běžné provádět i tzv. *bruteforce attack* (útok hrubou silou). Pokud si uživatelé nevolí dostatečně silná hesla, lze předpokládat, že si tester může sestavit seznam unikátní pro uživatele/stránku. Tento seznam může být využit i při případném získání hashů hesel, čímž tester otestuje, zda vývojáři používají kryptografickou sůl a standardy doporučené hashovací funkce.

V původním plánu pro *newspeak* byly funkcionality pro nejznámější sociální sítě, jako Facebook, Instagram, Twitter, apod. Bohužel *Terms of Service* (Podmínky Služby) pro Facebook, Instagram, Google+ a další sítě nepovoluje scraping dat jejich uživatelů, i když se jedná o veřejně dostupná data. Z tohoto důvodu jsou implementovány funkcionality pro scraping Twitteru uživatele a omezeně i pro TikTok.

5.2 Popis funkcionality a vývoje

Hlavním balíčkem aplikace je *beautifulsoup4*, určený pro scraping. Právě díky němu dokáže aplikace najít veškerý text z daného webu, provést nad ním potřebné operace a vepsat do seznamu. [25]

Při zkoušení aplikace se však vyskytl problém. K získání zdroje stránky byl využit balíček *requests*, který dokáže pracovat pouze s tzv. *statickými stránkami*. To znamená, že *webový server* nám po obdržení žádosti o stránku zašle kompletní HTML zdroj stránky (popřípadě CSS soubory). Dnešním trendem jsou ovšem weby dynamické, což umožňuje vývojářům vytvářet weby, které se mohou za chodu měnit podle uživatele (například jako doporučení od služby Netflix). Tyto stránky se kompletně načtou až při otevření v browseru (často pomocí skriptů v JavaScript na straně klienta a kódu na straně serveru). [26]

Zároveň například služba Twitter je generována dynamicky, a to umožňuje načítání dalších *tweetu* (příspěvků) na stránce uživatele při sjíždění myší. Právě proto byl opět použit balíček *selenium* s jeho možnostmi využití negrafických *browserů*. Jeho funkcionality umožňují na pozadí *browser* otevřít a získat dynamicky generovaný zdroj. Tentokrát je implementován pouze pro *chromedriver*. Cesta k němu je obsažena v konfiguračním souboru *driver.conf* (viz Výpis 5.1). Konfigurační soubor je potřeba vytvořit pokud se bude pracovat s dynamickými stránkami nebo *Twitterem* či *TikTokem*. Příklad souboru lze vypsat pomocí *–config-eg*. [23]

Výpis 5.1: Příklad driver.conf

```
1 PATH=/opt/chromedriver
```

Stejně tak jako je potřeba mít konfigurační soubor, je potřeba vždy využít přepínače *–file*, kterým uživatel sdělí cestu k souboru, který chce vytvořit jako nový seznam. Pokud však nechce nový seznam vytvořit, lze využít přepínače *–append*, kdy se soubor otevře v "připisovacím" módu (text se bude připisovat na konec souboru). Pokud přepínač *–file* použit není nebo cesta neexistuje (při použití *–append*), bude uživatel varován a aplikace ukončena.

Jestliže není specifikováno, zda se jedná o sociální platformu *Twitter* nebo *TikTok*, aplikace pracuje se stránkou jako s obecnou, tedy použije funkci *run_general()*. Tato funkce může běžet v *dynamickém* režimu, pokud je uvedena možnost *–dynamic*. Pro identifikaci důležitých částí HTML je použita funkce ve výpisu 5.2.

Výpis 5.2: Funkce scrape_text pro získání textu

```
def scrape_text(source_code, word_length, path, booleans):
  text_tags = [
    'p',
    'div',
    'h1',
    'h2',
    'h3',
    'h4',
    'h5',
    'h6',
    'ul',
    'ol',
    'dl',
    'span',
  ]

  soup = bs(source_code, 'html.parser')

  found_tags = []

  if booleans[1]:
    print(f'[?] Scraping')
  else:
    print(f'{Colors.CYAN}[?] Scraping {Colors.END}')

  for t in text_tags:
    for s in soup.find_all(t):
      found_tags.append(s.string)

  if booleans[1]:
    print(f'[+] Finished scraping')
  else:
    print(f'{Colors.GREEN}[+] Finished scraping {Colors.END}')

  split_and_write(found_tags, word_length, path, booleans)
```

Pythonovský seznam (*list*) *text_tags* obsahuje části HTML, které nejčastěji obsahují text. Tento seznam je použit v kombinaci s *beautifulsoup* objektem *soup*, který pomocí své funkce *find_all* najde každý výskyt HTML "tagu" z dříve zmíněného seznamu. [27]

Veškerý nalezený text je předán funkci *split_and_write()*, který jej rozdělí na slova, vyfiltruje anglické spojky, předložky a ignoruje slova, která nesplňují kritérium pro minimální délku slova (implicitně čtyři, uživatel může měnit pomocí *–len*). Slova, která prošla filtrem, jsou nakonec zapsána do seznamu.

Ovšem slovo jako takové sice může být heslem, ale pravděpodobnost je velmi nízká. Proto byly zavedeny tři doplňující funkce, sloužící k rozšíření seznamu pomocí vytvoření variací na všechna slova. Tyto možnosti jsou:

- *–leet*
- *–randomcase*
- *–numpad*

První přepínač spouští funkci *leetify()*, tato funkce převede písmena v každém slově z doposud vytvořeného seznamu do písmen korespondující s abecedou tzv. *leetspeak*. Tato abeceda je variantou, která se běžně používá na Internetu. Typicky se například písmeno *l* nahrazuje svislicí (”|”), *i* se nahrazuje číslem 1. Kompletní abeceda (minimálně jedna z jejich variant) je uložena ve formě seznamu ve funkci *to_leet()* (viz Výpis 5.3). Tato funkce převede zadané písmeno do *leet* abecedy a vrátí jej. Funkce je volána nad počtem písmen podle úrovně, která je zadána uživatelem. Úrovně jsou 4, první až třetí úroveň znamenají sekvenčně rostoucí počet písmen (tedy 1 písmeno až 3 písmena), čtvrtá úroveň značí převedení všech písmen. Po ”přeložení” se všechna slova zapíší na konec seznamu.

Druhý přepínač taktéž spouští funkci nad celým doposud získaným seznamem (včetně výstupu z *leetify*). Stejně jako předešlá funkce bere od uživatele požadovanou úroveň. První až třetí úroveň vybere stále rostoucí počet náhodných písmen (2,3,4), která převede na velká. Se čtvrtou úrovní se vybere náhodný počet písmen ke změnění. Po celkovém převedení se výstup opět zapíše na konec seznamu.

Třetí funkce nikterak nezmění slova obsažená v souboru, nicméně připojí kombinaci čísel a speciálních znaků ke konci každého slova. Čísla se kombinují ve speciálních variantách podle typického rozložení čísel na pravé straně klávesnice. Čísla se kombinují vertikálně, horizontálně a diagonálně. Tyto kombinace jsou generovány tak, jak je pro uživatele nejpohodlnější pro zapsání a zapamatování (např. 123, 741, 159). Všechna čísla jsou zároveň generována i pozpátku a speciální znaky taktéž (např. *-,/*). Kombinace jsou přidány a postupně zapsány. Tato funkce značně rozšíří seznam čímž udělá z malého seznamu robustní soubor, ale nevýhodou je její časová náročnost. Zrychlit by šla za využití paralelizace. Pro všechny tři funkce navíc platí, že celý soubor je načten do paměti (aby nemohlo dojít k vyčerpání maximální velikosti python *list*).

Výpis 5.3: Funkce to_leet

```
def to_leet(character, upper):
  leet = {
    "a" : "4",
    "b" : "8",
    "c" : "(",
    "d" : "17",
    "e" : "3",
    "f" : "ph",
    "g" : "#",
    "i" : "1",
    "j" : "]",
    "k" : "|<",
    "l" : "|",
    "m" : "nn",
    "n" : "/V",
    "o" : "0",
    "p" : "9",
    "q" : "&",
    "r" : "12",
    "s" : "5",
    "t" : "7",
    "u" : "v",
    "v" : "\\/",
    "w" : "uu",
    "x" : "}{",
    "y" : "j",
    "z" : "2"
  }
  upper_leet = { k.upper():v.upper() for k,v in leet.items() }

  if upper:
    return upper_leet[character]
  else:
    return leet[character]
```

Největším problémem při vývoji bylo zjištění, že aplikace je nefunkční pro dynamicky generované webové stránky. Především pro Twitter. S tímto pomohlo dříve zmíněné *Selenium*. Díky němu se otevře požadovaná stránka v prohlížeci na pozadí, dynamický kód je načten a poté získán. Následně ale přišlo zjištění, že Twitter načítá příspěvky postupně s tím, jak uživatel sjíždí s myší. Z tohoto důvodu byla přidána funkcionalita *–scroll* (uživatel zadá, kolikrát má prohlížeč sjet až na konec stránky). Tohoto je docíleno pomocí JavaScriptu, který se vykoná v *chromedriveru* (viz Výpis 5.4).

Výpis 5.4: Funkce pro sjíždení na webu

```
1 def scroll(browser):
2     payload = 'window.scrollTo(0,document.body.scrollHeight)'
3     browser.execute_script(payload)
```

Fakt, že zadané *URL* je na platformě Twitter, uživatel sdělí pomocí *–twitter*. Program poté *URL* zkontroluje a porovná se souborem *robots.txt*. Pokud je odkaz obsažen v zakázaných, *newspeak* před tímto uživatele varuje a skončí. Ovšem byla přidána možnost *–ignore-robots* (zatím pouze pro twitter, může být vytvořena pro každou platformu), která odkaz kontrolovat nebude.

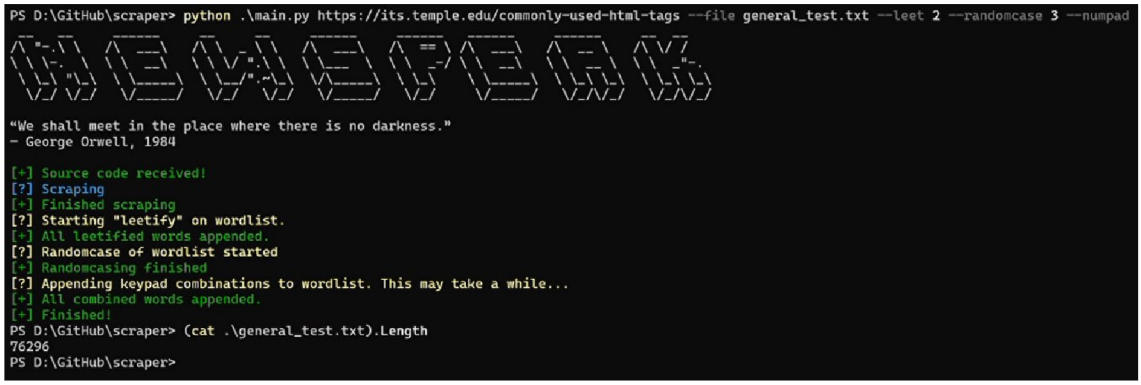
Stejně tak pro TikTok se používá přepínač *–tiktok*. Vyskytl se však problém se strukturou zdroje, kdy každý příspěvek musí být manuálně rozkliknut, aby byl získán jeho popis. Tato skutečnost by nešla spolehlivě naprogramovat, proto bylo potřeba se uchýlit k JSON, který je obsažený ve statické stránce TikTok. V něm jsou obsaženy informace o prvních šesti příspěvcích a o popisu profilu. Přepínač *–dynamic* nebude mít žádnou působnost a uživatel bude na tuto skutečnost upozorněn.

5.3 Ukázka funkcionality

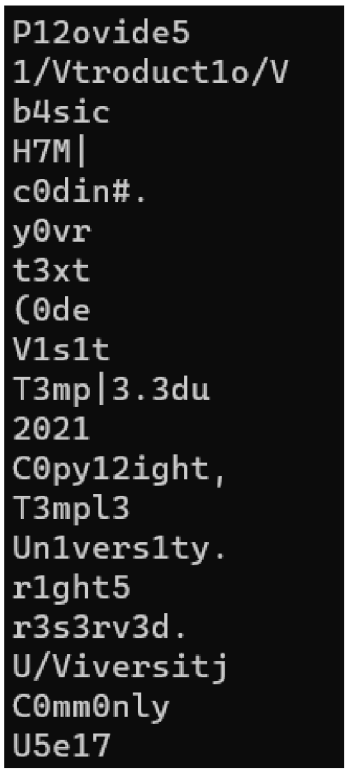
V této podkapitole budou opět obsaženy ukázky výstupu sloužící jako doplněk k předešlým kapitolám. Ukázky byly pořízeny z webové stránky `https://its.temple.edu/commonly-used-html-tags` vyhovující souboru *robots.txt*.

Obr. 5.1: Základní funkce

Obr. 5.2: Výstup za použití *–leet*

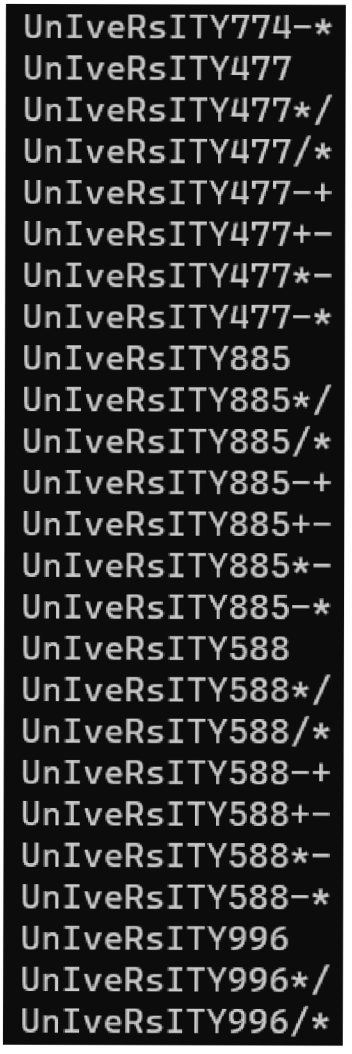


Obr. 5.3: Použití všech funkcí



Obr. 5.4: Kousek seznamu po *leet*

V Obr. 5.1 lze vidět, jak *newspeak* sděluje informace uživateli a délku výstupného seznamu (34 položek). Jedná se o poměrně krátký seznam. Následuje Obr. 5.2 opět ukazující informace a fakt, že délka souboru se zdvojnásobila po použití *leet*. Nejdůležitější je však Obr. 5.3, kde byly použity všechny možnosti, lze vidět značný skok ve velikosti seznamu. Skoky nejsou pouze vždy dalších 34 položek, nýbrž se seznam rozšířil na 76 296. Obr. 5.4 a Obr. 5.5 ukazují obsah seznamu po provedení *leet* resp. *randomcase* a *numpad*.



Obr. 5.5: Kousek seznamu po *randomcase* a *numpad*

Závěr

Tato práce byla zaměřena na metodologie penetračních testů a penetrační testování jako takové. Nejprve byl čtenář seznámen s potřebnými teoretickými znalostmi souvisejí s testováním, následně se přešlo k popisu metodologií a dokumentů. Druhá polovina práce již byla zaměřena na praktický výstup práce v podobě tří nástrojů. Tato část rozebírá postup vývoje, problematické části aplikací, jejich funkcionalitu a také možnosti pro zlepšení.

První kapitola byla zaměřena na objasnění termínu *penetrační testování*, co obnáší, jak se dá dále dělit na *black-box*, *white-box* a *gray-box* a nakonec i rozdíl mezi automatizovaným a manuálním testování. V druhé kapitole byly popsány dva dokumenty OWASP. V první podkapitole byl popsán soubor deseti nejzávažnějších zranitelností na webu, které byly jednotlivě popsány a u některých byly uvedeny příklady. V následující podkapitole bylo popsáno pět stěžejních bodů z dokumentu "OWASP Application Security Verification Standard". V poslední části druhé kapitoly byl ukázán statistický pohled na věc, kterým byla podložena tvrzení ze zbytku druhé kapitoly.

Kapitoly tři až pět se věnují praktickému výstupu práce. Postupně se jedná o programy *8search*, *crossget* a *newspeak*. V jednotlivých kapitolách byl blíže popsán postup při vývoji, vzniklé problémy a možnosti zlepšení aplikací do budoucna. Všechny tři aplikace jsou založené na práci v příkazové řádce pomáhající testerovi s identifikací zranitelností na webové stránce (případ *crossget*), mapování struktury stránek, resp. jaké na serveru existují adresáře a popřípadě soubory (případ *8search*) a jako poslední umožňující vytvoření vlastního seznamu podle textu obsaženém ve webové stránce (sloužící jako seznam možných hesel, případ *crossget*).

Literatura

[1] NIST SP 800-115. *Technical Guide to Information Security Testing and Assessment. 1* [online]. Gaithersburg, MD 20899-8930: NIST CSRC, 2008. Dostupné z: <https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-115.pdf>.

[2] MUSCAT, Ian. *The difference between Vulnerability Assessment and Penetration Testing* [online]. August 17, 2017 [cit. 2020-10-20]. Dostupné z: <https://www.acunetix.com/blog/articles/difference-vulnerability-assessment-penetration-testing/>

[3] MEUCCI, Matteo; MITCHELL, Rick; SAAD, Elie. *OWASP Testing Guide V. 4.1* [online]. Open Web Application Security Project, 2020. Dostupné z: <https://github.com/OWASP/wstg/releases/download/v4.1/wstg-v4.1.pdf>

[4] HOWARD, P. *What are Black Box, Grey Box, and White Box Penetration Testing?* [online]. August 11, 2020. Dostupné z: <https://resources.infosecinstitute.com/what-are-black-box-grey-box-and-white-box-penetration-testing>

[5] BARTH, A. *RFC-6265. HTTP State Management Mechanism. Network Working Group* [online], 2011. 2070-1721. Dostupné z: <https://tools.ietf.org/html/rfc6265>

[6] WICHERS, Dave; WILLIAMS, Jeff. *Owasp top-10 2017* [online]. OWASP Foundation, 2017. Dostupné z: https://owasp.org/www-pdf-archive/OWASP_Top_10-2017_%28en%29.pdf.pdf

[7] TURNER, S. a L. CHEN. RFC-6151. *Updated Security Considerations for the MD5 Message-Digest and the HMAC-MD5 Algorithms* [online]. Network Working Group, 2011. 2070-1721. Dostupné z: <https://tools.ietf.org/html/rfc6151.>

[8] VUMETRIC, *Why Automated App Pentests Are Not Enough* [online]. 2020-04-10 [cit. 2020-10-22]. Dostupné z: <https://www.vumetric.com/blog/why-automated-application-penetration-testing-is-not-enough/>

[9] TECHOPEDIA, *What is a Spider Trap? - Definition from Techopedia*. In: Techopedia [online]. November 2017 [cit. 2020-12-07]. Dostupné z: <https://www.techopedia.com/definition/5197/spider-trap>

[10] KOSTER, Martijn. *A Standard for Robot Exclusion* [online]. 30 June 1994n. l. Dostupné z: <https://www.robotstxt.org/orig.html>

[11] KATY, Anton; MANICO, Jim; BIRD, Jim. *Owasp top-10 proactive controls, v3 2018* [online]. OWASP Foundation, 2018. Dostupné z: <https://owasp.org/www-project-proactive-controls/>

[12] SULLIVAN, Bryan. *XML Denial of Service Attacks and Defenses. MSDN Magazine* [online]. 2009, November 2009, (24) [cit. 2020-12-07]. Dostupné z: https://docs.microsoft.com/en-us/archive/msdn-magazine/2009/november/xml-denial-of-service-attacks-and-defenses>

[13] OWASP, *DOM Based XSS Software Attack | OWASP Foundation*. In: OWASP Foundation: Open Source Foundation for Application Security [online]. [cit. 2020-12-07]. Dostupné z: <https://owasp.org/www-community/attacks/DOM_Based_XSS>

[14] ACUNETIX, *What is Insecure Deserialization?: Acunetix*. In: Acunetix: Web Application Security Scanner [online]. December 7, 2017 [cit. 2020-12-07]. Dostupné z: <https://www.acunetix.com/blog/articles/what-is-insecure-deserialization/>

[15] ABHAY, Bhargav, et al. *OWASP Application Security Verification Standard 4.0.2*. October 2020. Dostupné z: <https://github.com/OWASP/ASVS/raw/v4.0.2/4.0/OWASP%20Application%20Security%20Verification%20Standard%204.0.2-en.pdf.>

[16] PTSECURITY. *Web application vulnerabilities and threats: statistics for 2019* [online]. In: . February 13, 2020 [cit. 2020-12-07]. Dostupné z: <https://www.ptsecurity.com/ww-en/analytics/web-vulnerabilities-2020/>

[17] ACUNETIX. *Acunetix Web Application Vulnerability Report 2020* [online]. In: . 2020-05-06 [cit. 2020-12-07]. Dostupné z: <https://www.acunetix.com/acunetix-web-application-vulnerability-report/>

[18] NIST. *NVD - Statistics* [online]. [cit. 2020-12-07].
Dostupné z: <https://nvd.nist.gov/vuln/search/statistics?form_type=Advanced&results_type=statistics&query=Cross+Site+Scripting&search_type=all&pub_start_date=01%2F01%2F2018&pub_end_date=12%2F01%2F2020>

[19] NIST. *NVD - Statistics* [online]. [cit. 2020-12-07].
Dostupné z: <https://nvd.nist.gov/vuln/search/statistics?form_type=Advanced&results_type=statistics&query=SQL+injection&search_type=all&pub_start_date=01%2F01%2F2018&pub_end_date=12%2F01%2F2020>

[20] REITZ, Kenneth et al. *requests 2.24.0* [software]. Jun 17, 2020 [cit. 2021-04-15].
Dostupné z: <https://github.com/psf/requests>

[21] AKER, Erdi. *torrequest 0.1.0* [software]. Aug 17, 2016 [cit. 2021-04-15].
Dostupné z: <https://github.com/erdiaker/torrequest>

[22] WALLER, Peter et al. *pyfiglet 0.8.post1* [software]. Feb 15, 2019 [cit. 2021-04-15].
Dostupné z: <https://github.com/pwaller/pyfiglet>

[23] BURNS, David et al. *selenium 3.141.0* [software]. No 1, 2018 [cit. 2021-04-15].
Dostupné z: <https://github.com/SeleniumHQ/selenium>

[24] MICROSOFT. *msedge-selenium-tools 3.141.3* [software]. Nov 25, 2020 [cit. 2021-04-15].
Dostupné z: <https://github.com/microsoft/edge-selenium-tools>

[25] RICHARDSON, Leonard et al. *beautifulsoup4 4.9.3* [software]. Oct 3, 2020 [cit. 2021-04-15].
Dostupné z: <https://pypi.org/project/beautifulsoup4/>

[26] SAMBOL, Anita. MOTOCMS. *Static vs Dynamic Website: Ultimate Comparison and Cool Examples* [online]. 24 October, 2019n. l. [cit. 2021-04-16].
Dostupné z:
<https://www.motocms.com/blog/en/static-vs-dynamic-website/>

[27] COPES, Flavio. FLAVIOCOPES.COM. *HTML tags for text* [online]. Aug 07, 2019 [cit. 2021-04-16].
Dostupné z: <https://flaviocopes.com/html-text-tags/>

Seznam symbolů, veličin a zkratek

ACL	Access Control List
API	Application Programming Interface
ASCII	American Standard for Information Interchange
ASVS	Application Security Verification Standard
CSRF	Cross Site Request Forgery
CSS	Cascading Style Sheet
DOM	Document Object Model
DOS	Denial of Service
DTD	Document type Definition
FTP	File Transfer Protocol
GDPR	General Data Protection Regulation
HTML	Hyper Text Markup Language
HTTP	Hyper Text Transfer Protocol
IDOR	Insecure Direct Object Reference
IDS	Intrusion Detection System
IP	Internet Protocol (address)
IoT	Internet of Things
IPS	Intrusion Prevention System
JS	JavaScript
JSON	JavaScript Object Notation
MD5	Message Digest Algorithm 5
MITM	Man in the Middle
NIST	National Institute of Standards and Technology
ORM	Object-Relational Mapping

OWASP	Open Web Application Security Project
PCI DSS	Payment Card Industry Data Security Standard
PTES	Penetration Testing Execution Standard
RB	Raiffeisen Bank
SMTP	Simple Transfer Protocol
SQL	Structured Query Language
SSRF	Server Side Request Forgery
TLS	Transport Layer Security
URL	Uniform Resource Locator
UTF-8	Unicode Transformation Format - 8 bit
XML	Extensible Markup Language
XSS	Cross Site Scripting
XXE	XML External Entities
YAML	YAML Ain't Markup Language

Seznam příloh

A Struktura zdrojových souborů

Následující výpisy ilustrují strukturu zdrojových souborů praktického výstupu bakalářské práce. Jedná se o tři aplikace: *8search*, *crossget* a *newspeak*.

Aplikace byly testovány na operačních systémech MS Windows 10 Home (Version 20H2 - Build 19042.928) a Kali Linux 2020.4. Testováno s python verze 3.7.4 (na obou systémech).

8search

```
├── eight_tor ..... Adresář s modulem pro spojení přes Tor
│   └── tor.py
├── 8search.py ..... Hlavní soubor pro spouštění
├── LICENSE
├── README.md
├── small.txt ..... Testovací slovník
└── requirements.txt ..... Soubor obsahující názvy balíčků 3. strany
```

crossget

```
├── modules
│   ├── callback ..... Modul pro testování blind XSS
│   │   ├── call_back_wordlist.txt
│   │   ├── simple_server.py
│   │   └── wordlist_read.py
│   ├── drivers ..... Modul pro práci s drivery prohlížečů
│   │   ├── selenium_chromium.py
│   │   ├── selenium_edge.py
│   │   └── selenium_firefox.py
│   ├── extras ..... Doplňující modul
│   │   ├── banner.py
│   │   └── support.py
│   ├── parsing ..... Modul pro parsing konfiguračních souborů a argumentů
│   │   ├── args_parsing.py
│   │   ├── driver_identify.py
│   │   └── parse_config.py
│   ├── callback_xss_tester.py
│   ├── form_scraping.py
│   └── xss_tester.py
├── LICENSE
├── eg_config.txt ..... Příklad konfiguračního souboru
├── crossget.py ..... soubor pro spouštění
├── payloadsAllTheThingsWordlist ..... Slovník obsahující payloady
├── README.md
└── requirements.txt
```

newspeak

- modules
  - general_scrape ..... Modul pro *scraping* stránek
    - general.py
  - parsing
    - parser.py
  - socials ..... Modul pro *scraping* sociálních sítí
    - social_scraper.py
    - tiktok.py
    - twitter.py
  - specials ..... Modul obsahující doplňující funkce
    - banners.py
    - colors.py
    - conf.py
    - ignore.py
    - keypad.py
    - leet.py
    - randomcase.py
  - writer ..... Modul pro práci se soubory
    - file.py
- README.md
- newspeak.py ..... Soubor pro spouštění
- requirements.txt