Zadání diplomové práce

Název:	Analýza krypterů a jejich detekování
Student:	Bc. Jakub Kaloč
Vedoucí:	Ing. Josef Kokeš
Studijní program:	Informatika
Obor / specializace:	Počítačová bezpečnost
Katedra:	Katedra informační bezpečnosti
Platnost zadání:	do konce zimního semestru 2022/2023

Pokyny pro vypracování

1) Seznamte se s problematikou nástrojů pro ochranu kódu před reverzní analýzou - krypterů, packerů a protektorů.
2) Zaměřte se na kryptery používané k ochraně škodlivého softwaru a analyzujte, jaké techniky používají.
3) Relevantní techniky (zvolené na základě dohody s vedoucím práce) zanalyzujte a zakategorizujte v rámci Mitre Att&ck frameworku.
4) Na základě svých zjištění vytvořte detekční pravidla pro nástroj YARA, detekující tyto techniky.
5) Ověřte funkčnost detekce na vzorcích malwaru.
6) Diskutujte své výsledky.

Elektronicky schválil/a prof. Ing. Róbert Lórencz, CSc. dne 13. září 2021 v Praze.

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Diplomová práca

Analýza krypterů a jejich detekování

Bc. Jakub Kaloč

Katedra informační bezpečnosti
Vedúci práce: Ing. Josef Kokeš

21. decembra 2021

Poďakovanie

Chcel by som sa poďakovať Ing. Josefovi Kokešovi, vedúcemu mojej práce, za cenné rady, veľkú ochotu, trpezlivosť a čas, ktorý mi venoval pri písaní tejto práce. Ďalej by som sa chcel poďakovať mojim rodičom, sestre a priateľke za spätnú väzbu k obsahovej aj jazykovej stránke textu, ktorú mi poskytovali počas vytvárania tejto práce.

Prehlásenie

Prehlasujem, že som predloženú prácu vypracoval samostatne a že som uviedol všetky informačné zdroje v súlade s Metodickým pokynom o etickej príprave vysokoškolských záverečných prác.

Beriem na vedomie, že sa na moju prácu vzťahujú práva a povinnosti vyplývajúce zo zákona č. 121/2000 Sb., autorského zákona, v znení neskorších predpisov, a skutočnosť, že České vysoké učení technické v Praze má právo na uzavrenie licenčnej zmluvy o použití tejto práce ako školského diela podľa § 60 odst. 1 autorského zákona.

V Prahe 21. decembra 2021 . . . .

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2021 Jakub Kaloč. Všetky práva vyhradené.
Táto práca vznikla ako školské dielo na FIT ČVUT v Prahe. Práca je chránená medzinárodnými predpismi a zmluvami o autorskom práve a právach súvisiacich s autorským právom. Na jej využitie, s výnimkou bezplatných zákonných licencií, je nutný súhlas autora.

Odkaz na túto prácu

Kaloč, Jakub. *Analýza krypterů a jejich detekování.* Diplomová práca. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2021.

Abstrakt

Cieľom tejto práce je analýza a popísanie techník používaných kryptermi, ktorých primárna funkcionalita je ochrana malwaru pred detekciou a analýzou. Práca ukazuje čitateľovi, ako aplikovať znalosti reverzného inžinierstva na analýzu malwaru a využiť poznatky nadobudnuté pri analýzach na vytvorenie a zdokonalenie obrán pred malwarom.

Práca vo svojej teoretickej časti predstavuje vybrané nástroje na analýzu malwaru a vytváranie detekcií. Zároveň je predstavený framework, ktorý slúži na štrukturalizáciu jednotlivých techník, s ktorými sa je možné pri malwari stretnúť. Na pozadí tohto frameworku je tematicky štrukturovaná aj táto práca. V rámci jednotlivých kapitol sa text zameriava na techniky vybraných kategórií spomínaného frameworku, ktoré kryptery používajú.

Praktická časť práce nadväzuje na teóriu predstavenú v jednotlivých kapitolách. V prvom rade prebehne analýza vzoriek reálnych obfuskátorov a krypterov, ktoré je možné vídavať použité na ochranu malwaru. Analýza je vždy zameraná na časti, kde vzorka používa tematicky relevantné techniky. Na záver praktickej časti sú znalosti nadobudnuté pri analýze využité na vytvorenie detekčných pravidiel. Tieto pravidlá popisujú vzorky chránené analyzovanými kryptermi alebo aj samotné analyzované techniky.

Kľúčové slová krypter, packer, protektor, reverzné inžinierstvo, obfuskácie, YARA, Cuckoo Sandbox

Abstract

The aim of this thesis is an analysis and description of different techniques used by crypters whose main functionality is the protection of malware from being detected and analyzed. The thesis introduces the reader to the application of the knowledge of reverse engineering to analyzing malware and to the use of information from this analysis in creation and improvement of the protection against malware.

The theoretical part of this thesis presents selected tools for malware analysis and creation of detection rules. It also presents the framework used to logically structure individual techniques which can be found in malware. This framework is also used as the background structure of this thesis. Within individual chapters the thesis focuses on techniques of selected categories of the framework which are used by the crypters.

The practical part of the thesis builds on the theory presented in previous chapters. First, the analysis of samples of real obfuscators and crypters used for malware protection is performed. The analysis is focused on parts where the sample uses thematically relevant techniques. At the end of the practical part, the information obtained from the analysis is used to create detection rules which describe individual crypters or the analysed techniques.

Keywords crypter, packer, protector, reverse engineering, YARA, obfuscations, Cuckoo Sandbox

Obsah

Úvod **1**

1 Packery, kryptery a protektory **5**

1.1 Packery 6
1.2 Kryptery 7
1.3 Protektory 8
1.4 Problém falošnej pozitivity 9
1.5 Význam a prínosy identifikácie a klasifikácie 11

2 Nástroje a systémy na detekciu malwaru **13**

2.1 MITRE ATT&CK framework 13
2.1.1 MITRE ATT&CK matica 14
2.1.2 Časti Enterprise matice relevantné pre ďalšiu prácu . . . 15
2.2 Cuckoo 16
2.2.1 Signatúry 16
2.2.2 Vytváranie signatúr 17
2.3 YARA 18
2.3.1 Štruktúra YARA pravidiel 19
2.3.2 Moduly a rozšírenia 22
2.3.3 Rozšírenie Cuckoo modulu 22
2.3.4 YARA v Cuckoo 24
2.3.5 Monitorovanie nových pravidiel 24

3 Obfuskačné techniky **25**

3.1 Steganografia 25
3.1.1 Cassandra Crypter 25
3.2 Obfuskácia názvov v .NET-e 29
3.2.1 Prekonanie obfuskácie názvov v .NET-e 30
3.3 ConfuserEx 2 30
3.3.1 Prehľad funkcionalít 32

3.3.2 Detekcia 38

4 Slabiny dešifrovacích algoritmov **41**
4.1 Prístup k novej vrstve v .NET-e 42
4.1.1 Prvá metóda – statická extrakcia 42
4.1.2 Druhá metóda – dynamická extrakcia pomocou uloženia obsahu premennej 43
4.1.3 Tretia metóda – dynamický prístup pomocou načítania modulu 43
4.2 Kazy Crypter 44
4.2.1 Analýza dešifrovacieho procesu 44
4.2.2 Vytvorenie detekcie 46
4.3 Morpheus Crypter 47
4.3.1 Prvá vrstva – prvotná analýza 47
4.3.2 Prvá vrstva – analýza do hĺbky 47
4.3.3 Prvotná analýza druhej vrstvy 50
4.3.4 Detekcia 50

5 Techniky spustenia payloadu **53**
5.1 Process Hollowing 54
5.1.1 Popis realizácie techniky 54
5.1.2 Detekcia techniky 56
5.2 Self Hollowing 57
5.2.1 Pozične nezávislý kód 57
5.2.2 Analýza techniky 57
5.2.3 Detekcia techniky 61

6 Techniky obrany pred analýzou **63**
6.1 Kazy Crypter 63
6.1.1 Obrany protektoru 64
6.1.2 Dokončenie analýzy 64
6.2 Morpheus Crypter 65
6.2.1 Obrany protektoru 65
6.2.2 Dokončenie analýzy 66
6.3 Aegis Crypter 66
6.4 Detekcia analyzovaných techník 67
6.4.1 Kontrola prítomnosti dynamických knižníc 67
6.4.2 Kontrola prítomnosti debuggeru 69
6.4.3 Kontrola spustených procesov 69
6.4.4 Kontrola otvorených okien 69
6.4.5 Kontrola virtualizácie cez WMI dotazy 69
6.4.6 Detekcia preskakovania spánku 70
6.4.7 Zhodnotenie detekcií 70

Závi er **73**

Zoznam obrázkov

1.1 Princíp fungovania packeru, krypteru a protektoru 6
1.2 Pomenovanie sekcií po použití UPX packeru 7
1.3 Ukážka funkcionalít protektoru 10
1.4 Šírenie *BetaBot* malwaru pomocou OnionCrypter-u 12

2.1 Definícia funkcie `on_process` z abstraktnej triedy `Signature` . . . 17
2.2 Definícia funkcie `on_call` z abstraktnej triedy `Signature` 18
2.3 ukážka YARA pravidla 19

3.1 náhľad na *stub* Cassandra Crypter-u č. 1 27
3.2 náhľad na *stub* Cassandra Crypter-u č. 2 28
3.3 náhľad na obfuskované názvy Morpheus Crypteru 31
3.4 extrakcia textových reťazcov zo vzorky ConfuserEx 2 34
3.5 Obfuskácia toku riadenia nástrojom ConfuserEx 2 37

4.1 Morpheus Crypter – pred deobfuskáciou 48
4.2 Morpheus Crypter – po deobfuskácii 48

5.1 Onion Crypter – hlavičky sekcií pred prevedením self hollowing-u . 59
5.2 Onion Crypter – hlavičky sekcií po prevedení self hollowing-u . . . 59
5.3 Onion Crypter – PE Optional Header pred self hollowing-om . . . 60
5.4 Onion Crypter – PE Optional Header po self hollowing-u 60

6.1 Aegis Crypter – obrana pred virtualizovaným prostredím 68
6.2 Signatúry spustené vzorkou Morpheus Crypteru 72

Zoznam tabuliek

2.1 Výrez z MITRE ATT&CK matice 14

Úvod

Pod pojem *threat intelligence* sa zahŕňajú všetky informácie o kybernetických hrozbách a aktéroch. Zhromažďovanie takýchto informácií je kritické pre monitorovanie malwaru, obranu pred prebiehajúcimi útokmi a vytváranie efektívnych preventívnych opatrení na ochranu pred budúcimi útokmi. Práve vďaka kvalitnej a rozsiahlej *threat intelligence* je možné sledovať aktivitu existujúcich aktérov a vznik nových skupín alebo jednotlivcov ohrozujúcich kybernetický priestor. S dostatočným množstvom informácií je následne možné na základe histórie a predošlých verzií malwaru rýchlo rozpoznať nové a aktualizované verzie, ochrániť sa pred nimi a pripísať ich autorstvo konkrétnym aktérom (atribúcia malwaru). Všetky takéto informácie sú kľúčové nielen pri ochrane organizácií a jednotlivcov, ale aj pre vypátranie kyberzločincov, ktorí za útokmi stoja, ich následné zadržanie a zastavenie ich útokov.

Zhromažďovanie a dokumentovanie *threat intelligence* je výraznou pomocou pre každého analytika malwaru. Čím obsiahlejšie a kvalitnejšie informácie sú analytikovi dostupné, tým rýchlejšie je schopný zhodnotiť, čo je to, čo aktuálne skúma a akú hrozbu predmet skúmania predstavuje.

Práca bola vytvorená s motiváciou rozšírenia dostupnej *threat intelligence* a poskytnutia uceleného náhľadu na prvú líniu obrany takmer každého moderného malwaru. Informácie obsiahnuté v tejto práci je možné využiť na urýchlenie analýzy nových a neznámych vzoriek spustiteľných programov a efektívnejšiu identifikáciu malwaru.

Sekundárnou motiváciou pre vytvorenie tejto práce bolo predstavenie možnej aplikácie reverzného inžinierstva na analýzu malwaru a jednotlivých rodín malwaru. V práci sa predstavia poznatky a prostriedky potrebné na využitie vedomostí nadobudnutých pri analýze vzoriek malwaru za účelom vytvorenia obrán pred malwarom a rozšírenia existujúcej *threat intelligence*.

Hlavným cieľom práce je preskúmanie, analýza a popísanie techník, ktoré používajú packery, kryptery a protektory na to, aby dokázali ochrániť malware, ktorý ukrývajú pred detekciou antivírusu a skomplikovali analytikovi prácu. To je dosiahnuté splnením viacerých vedľajších cieľov, ktoré sú:

- Predstavenie packerov, krypterov a protektorov a vymedzenie jednotlivých pojmov.
- Predstavenie nástrojov vhodných na analyzovanie malwaru a vytváranie obrán pred malwarom.
- Podrobná analýza často používaných techník.
- Ukážky analyzovaných techník spolu s *proof of concept* kódom alebo demonštráciou na vzorkách malwaru.
- Vytvorenie detekčných pravidiel na techniky, pre ktoré to je možné.
- Namapovanie jednotlivých techník na kategórie MITRE ATT&CK frameworku, pre štrukturalizáciu a kategorizáciu detekcií.

Práca je štruktúrovaná do šiestich kapitol. V prvej kapitole práca predstavuje pojmy packer, krypter a protektor. Funkcionality každého z uvedených typov nástrojov sú popísané spolu s bežným a očakávaným využitím týchto nástrojov. Jednotlivé typy nástrojov sú navzájom porovnávané pre lepšie pochopenie jednotlivých, niekedy málo viditeľných, rozdielov. Na záver kapitoly sa predstavuje problematika falošnej pozitivity, ktorá sprevádza nielen detekcie spomínaných nástrojov, ale aj všeobecne akékoľvek detekcie malwaru.

Druhá kapitola predstavuje čitateľovi nástroje a systémy na detekciu a analýzu malwaru. Každý z predstavených nástrojov a systémov bol vytvorený za iným účelom a v kapitole sa načrtne, ako je možné využiť silné stránky jednotlivých nástrojov a skombinovať ich použitie na efektívne vytváranie obrán pred malwarom. Na konci kapitoly sa predstaví rozšírenie jedného z nástrojov, ktoré bolo kľúčové pre efektívnejšie prepojenie funkcionalít zmienených nástrojov. Toto rozšírenie je následne využívané v ďalších kapitolách.

Kapitola číslo tri popisuje vybrané obfuskačné techniky, s ktorými sa je možné stretnúť pri analyzovaní malwaru. V kapitole sú predstavené prvé dva obfuskátory, ktoré je možné často vidieť použité na ochranu malwaru. Obfuskátory sú predstavené v rôznych podkapitolách ako reprezentanti implementácie a použitia popisovaných obfuskačných techník. V kapitole sú predstavené aj prvé detekcie vytvorené za účelom detekovania rozoberaných techník alebo obfuskátorov.

Štvrtá kapitola sa zaoberá slabinami dešifrovacích algoritmov, ktoré používajú obfuskátory. V úvode kapitoly sa pojednáva o výhodách a nevýhodách implementovania vlastných kryptografických algoritmov. Kapitola následne pokračuje predstavením a porovnávaním možných postupov, ako pristúpiť k novým šifrovaným alebo obfuskovaným vrstvám pri analýze programov. Vo zvyšku kapitoly sú analyzované dva vybrané kryptery z hľadiska slabín ich dešifrovacích algoritmov. Kapitola sa zavŕši vytvorením detekcií na predstavené kryptery. Ukáže sa tak, ako analýza slabín a chýb vo vlastných kryptografických algoritmoch môže napomôcť analytikovi k nasledovnej detekcii malwaru.

Piata kapitola je venovaná technikám spustenia finálneho payloadu na infikovanom zariadení. V prvej časti kapitoly sa predstaví technika *process hollowing*, na ktorú bol vyrobený aj ukážkový *proof of concept* program. Analýza techniky je zavŕšená vytvorením detekcie, ktorá je schopná odhaliť túto techniku aj v neznámych vzorkách malwaru. Druhá polovica kapitoly popisuje techniku *self hollowing* a možnosti, ako ju detekovať.

V poslednej kapitole sú rozoberané techniky obrany pred automatizovanou aj manuálnou analýzou. V kapitole sú analyzované tri kryptery, ktoré implementujú takéto techniky. Každý krypter je analyzovaný v samostatnej podkapitole spolu s technikami obrany pred analýzou, s ktorými sa je možné pri danom krypteri stretnúť. V závere kapitoly je možné nájsť popis a zhodnotenie vytvorených detekcií pre predstavené techniky.

Kapitola 1

Packery, kryptery a protektory

Moderný malware je komplexný software, ktorý často pozostáva z viacerých komponentov, kde každý komponent má svoj špecifický účel. [1][2] Pre autorov malwaru je nepraktické prepisovať nanovo komponenty, ktoré obsahujú hlavnú funkcionalitu ich programu (jadro malwaru) vždy, keď ich antivírus detekuje. Preto je dnes už štandardom, že malware je distribuovaný ako niekoľkovrstvová aplikácia, v ktorej nie je možné vidieť kód jadra pri dekompilácii, disasemblovaní alebo statickom skenovaní.[3]

Keďže je z hľadiska autorov malwaru potrebné prvé vrstvy aktualizovať a vyvíjať oveľa aktívnejšie, aby mal malware šancu ostať nedetekovaný, vznikli samostatné aplikácie, ktoré sa začali špecializovať práve na úlohu ochrany kódu v jadre. Tieto programy sa pridávajú do malwaru ako ďalší komponent, ktorý tvorí vonkajšie vrstvy finálnej aplikácie.

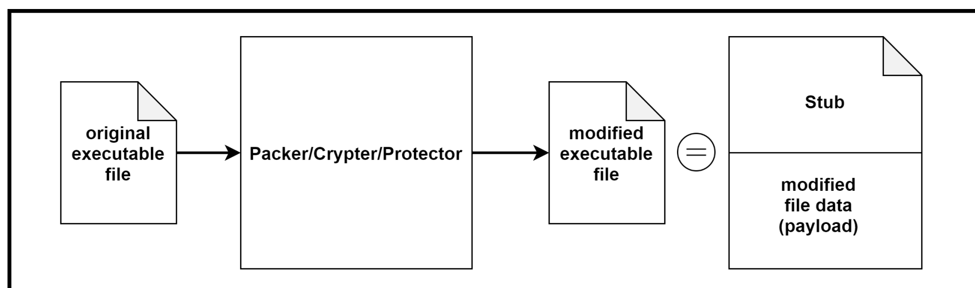
Na druhej strane však zároveň existuje legitímny dopyt po aplikáciách, ktoré vedia skryť a ochrániť kód. O ochranu kódu majú často záujem spoločnosti, ktoré nechcú prísť o konkurenčnú výhodu, pokiaľ sa im podarilo vyvinúť software, ktorý ich dáva do výhody pred konkurenciou. Ďalším pravidelným využívateľom softwaru na ochranu kódu sú aj tvorcovia počítačových hier, ktorí buď nechcú zverejniť herné mechaniky, alebo sa snažia brániť pred útokmi na hru samotnú a možným hackovaním.[4]

Existujúce aplikácie, ktoré ponúkajú možnosti ochrany kódu, majú rôzne funkcionality. Spravidla sa delia na tieto tri kategórie podľa toho, aké služby vedia poskytnúť:

- Packery
- Kryptery
- Protektory

Jedna aplikácia však môže poskytovať služby aj z viacerých vyššie uvedených kategórií zároveň. Vedieť, ako tieto aplikácie fungujú a ako prekonávať ich ochrany, je pre analytika malwaru nutnosťou. Napriek tomu, že analýza v dynamických sandboxoch vie prezradiť o správaní vzoriek veľa, bez možnosti priameho náhľadu do jadra malwaru môže analytik prísť o výrazné množstvo informácií.

Princíp fungovania packerov, krypterov a protektorov je veľmi podobný. Spustiteľný program sa vloží do aplikácie, ktorá spraví zmeny na originálnom programe v závislosti od toho, či sa jedná o packer, krypter alebo protektor. Takto transformovaný program sa uloží ako dáta do nového spustiteľného súboru spolu s ďalším kódom, ktorému sa hovorí *stub*. *Stub* slúži ako obálka pôvodného programu a jeho hlavnou úlohou je spraviť inverzné transformácie uložených dát. Dostane ich tak do originálneho stavu a následne originálny program spustí.



Obr. 1.1: Princíp fungovania packeru, krypteru a protektoru

1.1 Packery

Packer je program, ktorého úlohou je zredukovať veľkosť spustiteľného súboru a zároveň zachovať možnosť pôvodný program priamo spustiť. Redukcia veľkosti sa dosiahne komprimáciou súboru vrátane sekcií so spustiteľným kódom. Kedysi, keď bolo technologicky náročné veľké súbory distribuovať a skladovať na disketách alebo diskoch s malou kapacitou pamäte, bolo používanie packerov bežné. Dnes dostupné rýchlosti internetového pripojenia umožňujú prenášať aj väčšie súbory, a preto môže byť použitie packeru v dnešnej dobe podozrivé viac ako v minulosti. *Stub*-y packerov sú spravidla krátke a obsahujú len nevyhnutné inštrukcie potrebné na dekompresiu originálneho súboru. Zároveň často obsahujú aj informácie o tom, aký packer bol použitý, ako je možné vidieť na obrázku 1.2.[4]

Name	Virtual Size	Virtual Address	Raw Size	Raw Address	Reloc Address	Linenumbers	Relocations N...	Linenumbers ...	Characteristics
Byte[8]	Dword	Dword	Dword	Dword	Dword	Dword	Word	Word	Dword
UPX0	0006F000	00001000	00000000	00000400	00000000	00000000	0000	0000	E0000080
UPX1	0004A000	00070000	00049E00	00000400	00000000	00000000	0000	0000	E0000040
UPX2	00001000	000BA000	00000200	0004A200	00000000	00000000	0000	0000	C0000040

Obr. 1.2: Pomenovanie sekcií po použití UPX packeru

1.2 Kryptery

Úlohou krypteru je šifrovať a obfuskovať spustiteľný súbor. [4] Obfuskácia robí objekt menej zrozumiteľným až nezrozumiteľným. Jedná sa o zmenu objektu, ktorý má byť modifikovaný tak, aby mu bolo náročné porozumieť, ale jeho funkcionalita sa nezmenila. Obfuskovaný môže byť samotný kód programu alebo aj dáta, ktoré program používa. [5]

Už na tomto mieste môže dôjsť k miešaniu pojmov, keďže kompresia je jeden spôsob obfuskácie, ktorý vie spraviť statickú aj dynamickú analýzu náročnejšou, a teda packery môžu byť nazývané aj kryptery. Hlavným rozdielom je, že úlohou packeru nie je skrývanie, teda komprimačný algoritmus býva známy a často je možné nájsť aj dekomprimačný program, ktorý vie staticky vygenerovať zabalený súbor – bez nutnosti spustenia tohto súboru a rizika infekcie. Príkladom takéhoto dekomprimačného programu môže byť `PE Explorer`, ktorý je možné rozšíriť o plugin na rozbaľovanie súborov zabalených s `UPX` packerom.

Kryptery môžu používať známe šifrovacie algoritmy. Jedným z používaných algoritmov je napríklad šifra `RC4`, ktorá je zrejme rozšírená, lebo je funkčná a zároveň nie je náročné a ani zdĺhavé ju implementovať aj priamo v assembleri. Druhou možnosťou autorov je použiť vlastný šifrovací algoritmus, čo vie analytikovi o niečo predĺžiť prácu, kým príde na to, ako algoritmus funguje. Na druhej strane implementácia vlastného šifrovacieho algoritmu je často nedokonalá a stáva sa, že niektoré byty alebo dokonca celé bloky payloadu ostanú nešifrované. Zásadným rozdielom od packerov je, že krypter nemusí nijak zmenšiť veľkosť zabaleného súboru, teda pri šifrovaní môže dôjsť len k transformácii dát v pomere 1:1 alebo môže dôjsť dokonca aj k zväčšeniu pôvodného súboru.

Zatiaľ čo na packery je možné bežne nájsť program, ktorý vie originálny súbor staticky rozbaliť, úlohou krypteru je, aby pre nich takýto program nebol vytvorený. Pokiaľ sa takýto program na rozbalenie vytvorí a rozšíri, krypter je považovaný za znehodnotený a autor musí zmeniť funkcionalitu natoľko, aby boli rozbaľovacie programy znefunkčnené. Práve toto je dôvodom k vytváraniu vlastných šifrovacích funkcií, ktorých parametre alebo vlastnosti

sú v ďalších verziách mierne pozmenené. Takéto funkcie sú však z kryptografického hľadiska často nefunkčné.

Šifrovanie je však len jedna z možných techník obfuskácie, konkrétne obfuskácia uložených dát. Techník na obfuskáciu je mnoho a pri ich vytváraní je hranicou len fantázia autora. Pre analytika je nutnosťou zoznámiť sa s čo najväčším množstvom obfuskačných techník a postupov ako ich rýchlo prekonať, preto sa novoobjavené techniky bežne popisujú a publikujú v odborných publikáciach aj komunitných blogoch. Ďalšie techniky, okrem šifrovania, zahŕňajú aj obfuskácie samotného kódu krypteru, ktoré robia program neprehľadným a ťažko pochopiteľným. Vybrané techniky sú dopodrobna popísané v nasledujúcom texte, avšak príkladom môžu byť:

- *loop unrolling* – rozbaľovanie slučiek a vytváranie dlhých funkcií s repetitívnymi blokmi kódu.
- *junk code* – vkladanie kódu, ktorý nemá v programe žiadnu úlohu. Toto môže zahŕňať zbytočné inštrukcie, ale aj vkladanie nepotrebných API volaní.
- manipulácia s menami premenných alebo funkcií. Táto technika je významná hlavne pre .NET programy.
- *jump in the middle* – vkladanie skokových inštrukcií, ktoré skáču doprostred inštrukcie a zmätú disassembler alebo dekompilátor.

1.3 Protektory

Z trojice packer, krypter a protektor je protektor najkomplexnejší typ programu. Jeho úlohou je chrániť zabalený program pred analýzou, a to akýmikoľvek spôsobmi a technikami. Protektory často používajú techniky packerov a krypterov a pridávajú navyše ďalšie. *Stub*-y protektorov sú relatívne rozsiahle a komplikované, práve kvôli tomu, že implementujú kombinácie mnohých obfuskačných a ochranných techník.[4] Medzi najzákladnejšie techniky, ktorými sa dokážu brániť proti analýze, patria napríklad:

- Kontrolovanie, či je program otvorený v debuggeri.
- Kontrola názvov všetkých spustených programov a v prípade prítomnosti neželanej aplikácie uspatie alebo ukončenie programu.
- Skrývanie podstatných častí programu do pozične nezávislých kódov.
- Kontrola prítomnosti špecifických dynamických knižníc v pamäti.
- Skrývanie/falšovanie/randomizácia metadát a informácií o súbore.

Práve protektory sú často nasadzované v hernom priemysle, aby zabránili hackovaniu hier a vytváraniu tzv. trénerov – programov, ktoré modifikujú správanie hry a umožňujú napríklad prechádzanie cez pevné objekty alebo neobmedzenú viditeľnosť či zdroje.[4]

Na druhej strane sú protektory často vytvárané a zneužívané za účelmi šírenia malwaru. Takéto protektory fungujú ako platená služba, kedy si zákazník najbežnejšie platí za „počet ochránení“ a menej často za časovo obmedzenú licenciu na užívanie. Na šírenie sú používané dva modely. Prvý a starší je distribuovanie bežnej aplikácie (napr. Kazy Crypter), ktorá funguje lokálne, avšak pri novšie vzniknutých rodinách je bežné, že protektor funguje ako webová služba (napr. Cassandra Crypter), kedy zaregistrovaný užívateľ nahrá svoj súbor na stránku a následne stiahne jeho chránenú verziu.

Na obrázku 1.3 je ukážka protektoru s názvom *Aegis Crypter*, ktorý existuje už minimálne od roku 2015[6] a dnes je už spoľahlivo detekovateľný. Aby bol protektor konkurencieschopný, prešiel niekoľkými verziami a implementoval mnoho funkcionalít, ktoré je možné nájsť pri väčšine protektorov. Základnými funkciami sú odklad rozbaľovania alebo skrytie súboru. Ďalšie bežné funkcionality pokrývajú možnosti vyplnenia meta údajov o spustiteľnom súbore. Poslednou, ale veľmi častou a podstatnou súčasťou takýchto protektorov je voľba spôsobu spustenia zabaleného súboru. Protektory ponúkajú možnosti spustiť súbor ako samostatný proces, injektovať súbor do iného procesu a spustiť ho pod názvom iného procesu alebo injektovať súbor do samého seba. Pre ochranu pred malwarom, ktorý je zabalený takýmito protektormi, je potrebné byť oboznámený s funkcionalitami, ktoré protektory ponúkajú. Podrobná analýza, možnosti odhalenia a detekcie spomenutých techník sú detailne vysvetlené v nasledujúcich kapitolách.

Je možné si všimnúť, že napriek tomu, že sa v tejto sekcii pojednáva o protektoroch, názov programu v ukážke na obrázku 1.3 je krypter. Keďže sú si pojmy relatívne blízke, tak sa pojem krypter stal historicky oveľa populárnejší a tvorcovia malwaru ho začali používať aj pre protektory. Následne aj antivírusové spoločnosti reagovali na existujúcu terminológiu a napriek tomu, že sa definične pojmy krypter a protektor líšia, tak priamo v názvoch detekcií je možné vidieť, že sú používané pojmy `crypter`, `cryptic` alebo `kryptik`. Pre presnú terminológiu budú v tejto práci pojmy krypter a protektor rozlišované napriek tomu, že sú často zamieňané.

1.4 Problém falošnej pozitivity

Napriek tomu, že sa na prvý pohľad môže zdať ako praktické a jednoduché riešenie vytvárať detekcie a chrániť užívateľov pred akýmkoľvek programom,

Obr. 1.3: Ukážka funkcionalít protektoru

ktorý používa kryptery, packery alebo protektory, nájde sa mnoho legitímnych programov, ktoré tieto služby využívajú. Pokiaľ by teda antivírusy blokovali všetko, užívatelia by dostávali mnoho falošne pozitívnych hlásení o napadnutí aj z legitímnych a neškodných programov, ktoré bežne používajú a tieto programy by končili mazané zo systému. Toto by spôsobilo dve veci:

- Užívatelia by začali vytvárať výnimky a mohli by to časom robiť už tak automaticky, že by do výnimiek začlenili aj programy, ktoré by reálne mohli predstavovať hrozbu.
- Veľký počet falošne pozitívnych hlásení by väčšinu zákazníkov a užívateľov antivírusového softwaru mohol natoľko obťažovať, že by antivírus odinštalovali a ostali by tak bez ochrany pred malwarom.

Pri chránení je teda potrebné pozorne nastavovať prísnosť detekcií. Zároveň je nutné rozlišovať jednotlivé rodiny protektorov a sledovať ich aktivitu a históriu využívania. Až keď je známe, že protektory sú inzerované na hackerských fórach a hlavnou súčasťou ich propagácie a marketingu je to, že sú tzv. *FUD* (Fully undetectable – nedetekované žiadnymi antivírusmi) alebo sú v nich dlhodobo zabalené len rôzne druhy malwaru, je možné takéto rodiny protektorov plne blokovať.

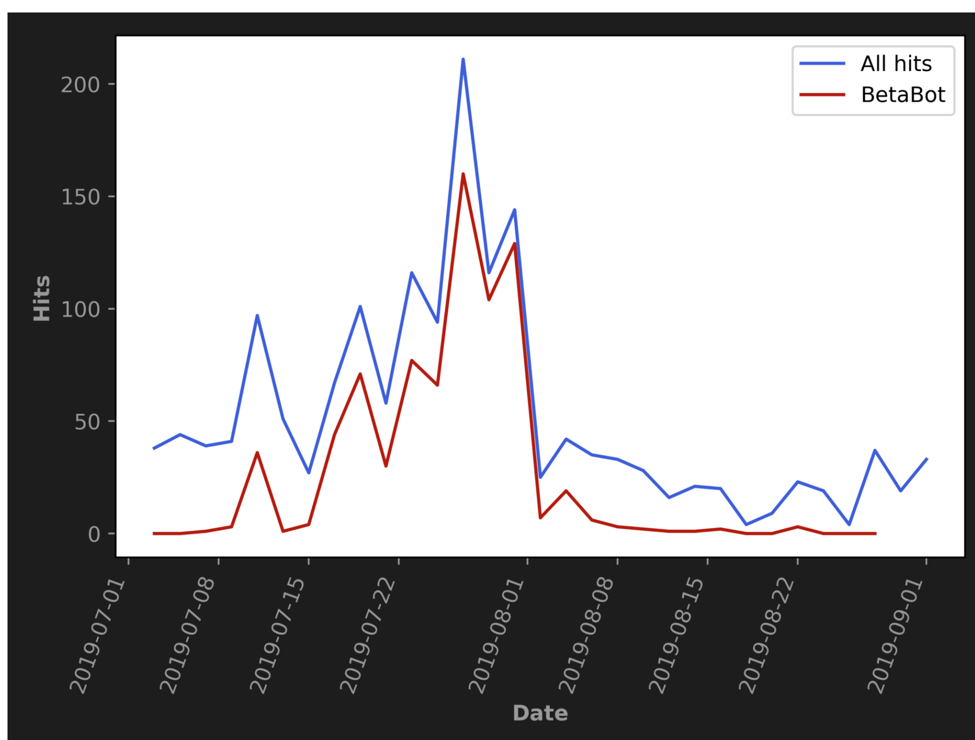
1.5 Význam a prínosy identifikácie a klasifikácie

Kvôli problému falošnej pozitivity krypterov, packerov a protektorov vie byť klasifikácia jednotlivých rodín a správne detekovanie časovo náročné. Napriek tomu je však monitorovanie vývoja tohto typu malwaru prínosná. Prvým dôvodom je, že pokiaľ sa podarí detekovať malware staticky ešte predtým, než sa spustí, tak sa zariadenie nestane infikované a malware bude úspešne zneškodnený. Toto by nebolo možné pre malware zabalený v krypteroch a protektoroch, pokiaľ sa nedetekuje priamo vonkajšia vrstva, teda práve konkrétny krypter alebo protektor.

Druhý dôvod je blízky prvému. Pokiaľ statická detekcia súbor nezneškodnila a súbor sa spustí, stále je možné využiť dynamické detekcie malwaru na to, aby zastavil beh škodlivého programu ešte predtým, ako sa spustí samotný zabalený malware. Teda podobne ako v predošlom bode sa zabráni infikovaniu zariadenia ešte predtým, ako sa spustí zabalený malware.

Ďalší významný dôvod je detekcia ešte neznámeho malwaru a hrozieb. V počítačovej bezpečnosti existuje významný problém, že obranca vie reagovať len na známe útoky a zabezpečiť systém iba pred tým, čo pozná. Na druhej strane, útočníkom kladie medze akurát ich fantázia a nový malware s novými technikami je vyvíjaný neustále. Pokiaľ však útočníci spustia novú

kampaň s novým a neznámym malwarom a použijú protektor, na ktorý už existujú funkčné detekcie, tak sa nová kampaň prejaví ako výrazne zvýšená aktivita výskytu daného protektoru. Takúto aktivitu je možné v systémoch obrancov jednoducho spozorovať a zanalyzovať situáciu. Následne po objavení novej malwarovej rodiny je možné vytvoriť detekcie už priamo na danú rodinu a zastaviť jej šírenie aj v prípade, že by zmenila protektor, ktorý používa. Príkladom takejto situácie je šírenie malwaru *BetaBot* v roku 2019 pomocou OnionCrypter-u. Na grafe 1.4 je možné vidieť, že na prelome júla a augusta v roku 2019 bolo zaznamenané výrazné zvýšenie v počte užívateľov, u ktorých bol detekovaný súbor chránený OnionCrypter-om. Modrý graf zobrazuje počet unikátnych súborov OnionCrypter-u, ktoré boli nájdené na zariadeniach užívateľov a červený graf zobrazuje, koľko z týchto súborov obsahovalo malware *BetaBot*. [7]



Obr. 1.4: Šírenie *BetaBot* malwaru pomocou OnionCrypter-u podľa [7]

Kapitola 2

Nástroje a systémy na detekciu malwaru

Pred samotnou analýzou techník používaných kryptermi, packermi a protektormi je potrebný výber nástrojov. Na jednoduché a prehľadné mapovanie jednotlivých techník bude použitý framework ATT&CK. Na analýzu vzoriek a na vytváranie detekcií sú použité open-source projekty YARA a Cuckoo Sandbox, ktoré používa aj mnoho entít zaoberajúcich sa IT bezpečnosťou ako napríklad Avast, alebo VirusTotal. [8]

2.1 MITRE ATT&CK framework

V roku 2013 vytvorili výskumníci zo spoločnosti MITRE framework s plným názvom *Adversarial Tactics, Techniques, and Common Knowledge*, ktorý je používaný v skrátenej forme ATT&CK. [9] Tento framework bol vytvorený za účelom systematickej kategorizácie správania sa protivníka. Základom frameworku je množina techník, ktoré môže útočník akýmkoľvek spôsobom využiť, aby dosiahol svoj cieľ. Každá technika má priradené informácie, medzi ktoré patria aj názov, popis techniky alebo unikátne ID techniky. Pokiaľ sa jedná o podtechniku, jej ID sa skladá z prefixu, ktorý je ID hlavnej techniky, bodky slúžiacej ako oddeľovač a sufixu pozostávajúceho z trojčíslia unikátneho pre konkrétnu podtechniku. Názov aj ID sú v tomto texte používané pre referenciu konkrétnych techník.

Je potrebné si hneď na začiatku uvedomiť, že framework ponúka množinu použiteľných techník, ktoré nemusia nutne implikovať útok. Napríklad niektoré programy môžu byť používané rovnako administrátorom zariadenia alebo siete, ako aj útočníkom. Preto je potrebné framework vnímať ako zoznam potenciálnych hrozieb a problémov, z ktorého je možné systematicky čerpať. Teda aj detekcie jednotlivých techník, ktoré sú popísané týmto framewor-

kom nebudú slúžiť na klasifikáciu konkrétnych malwarových rodín ani na definitívnu detekciu malwaru. Budú však slúžiť ako veľmi hodnotné indikátory potenciálneho nebezpečenstva, ktoré môžu klasifikácii malwaru výrazne pomôcť.

V súčasnosti sa framework delí na tri časti:

- ATT&CK for Enterprise – zameranie na podozrivé správanie a techniky používané v rámci OS Windows, Linux, macOS.
- ATT&CK for Mobile – zameranie na podozrivé správanie a techniky používané v rámci operačných systémov Android a iOS.
- Pre-ATT&CK – zameranie na správanie sa protivníka pred útokom.

[10]

2.1.1 MITRE ATT&CK matica

Pre jednoduchú a prehľadnú vizualizáciu techník boli vytvorené matice pre časti Enterprise a Mobile, pričom časť Pre-ATT&CK je zakomponovaná v rámci týchto dvoch matíc. Matica zachytáva vzťah medzi cieľmi útočníka a jednotlivými technikami, ako tieto ciele dosiahnuť. Mnohé z cieľov je možné realizovať viacerými spôsobmi a aj mnohé techniky sa často delia na rôzne podtechniky, čo je taktiež zachytené v tejto matici.

Defense Evasion (40 techniques)	**Command and Control (16 techniques)**	**Discovery (29 techniques)**
Deobfuscate/Decode Files or Information	Communication Through Removable Media	Process Discovery
Indirect Command Execution	Data Encoding	Account Discovery
Process Injection	Encrypted Channel	Container and Resource Discovery
Weaken Encryption	Dynamic Resolution	Cloud Service Discovery
Hijack Execution Flow	Non-Standard Port	Network Service Scanning
Obfuscated Files or Information	Protocol Tunneling	Network Sniffing
...	...	...

Tabuľka 2.1: Výrez z MITRE ATT&CK matice[11]

2.1.2 Časti Enterprise matice relevantné pre ďalšiu prácu

Taktík, ktoré môžu útočníci používať, je mnoho, avšak kryptery, packery a protektory majú konkrétne ciele a viažu sa na ne teda konkrétne taktiky a techniky. Z Enterprise matice sú pre problematiku rozoberanú v tejto práci podstatné najmä nasledujúce časti matice:

- *Process Injection (ID T1055)* – táto technika alebo konkrétna podtechnika tejto kategórie je implementovaná a používaná každým packerom, alebo krypterom, keďže primárny účel týchto nástrojov je spustiť zabalený program. Táto technika má v Enterprise matici až jedenásť rôznych podtechník.

- *Obfuscated Files or Information (ID T1027)* – obfuskácia dát je primárny nástroj krypterov a protektorov. Problémom tejto kategórie je, že obfuskácia dát je závislá na kreativite útočníka, takže pokrytie možných podtechník by znamenalo vytváranie detekcií tak špecifických, že by detekovali už len konkrétnu rodinu malwaru, ktorá danú techniku používa. Napriek tomu je možné spraviť výber a pokryť najčastejšie používané techniky, s ktorými sa je možné bežne stretnúť pri analýze malwaru.

- *Deobfuscate/Decode Files or Information (ID T1140)* – technika bežná pre každý packer, krypter a protektor. Podobne ako pri predošlej technike, nie je možné pokryť všetky možné implementácie, ale je možné sa zamerať na tie časté.

- *Data Obfuscation (ID T1001)* – táto technika pojednáva primárne o obfuskácii a ochrane dát používaných pre komunikáciu s *command and control* servermi. Čo sa však týka samotnej implementácie a prejavov tejto techniky, tak má veľký prekryv s predošlými dvomi kategóriami techník. Obfuskované dáta v súboroch a v sieťových prenosoch môžu často používať rovnaký spôsob obfuskácie.

- *Virtualization/Sandbox Evasion (ID T1497)* – mnohé z moderných protektorov ponúkajú možnosť kontroly spúšťania v sandboxoch. Tieto kontroly môžu spočívať v kontrole hardwarových komponentov, spustených programov alebo prítomnosti súborov charakteristických pre konkrétne virtuálne stroje.

Je možné si všimnúť, že niektoré kategórie techník, ako napríklad *Virtualization/Sandbox Evasion (ID T1497)*, sa nachádzajú na viacerých miestach v ATT&CK matici, pretože niektoré techniky môžu byť použité na dosiahnutie viacerých cieľov. V použitom príklade je možné vidieť techniky spadajúce pod *Virtualization/Sandbox Evasion* s cieľom obrany pred detekciou a analýzou, ale aj s cieľom zbierania informácií o systéme. Ako sa ukazuje v článku [12], zbieranie informácii o sandboxoch, v ktorom sa malware spúšťa a analyzuje,

môže útočníkom priniesť cenné informácie na vytvorenie pokročilých obrán proti týmto sandboxom v nových verziách malwaru.

Vybrané techniky sú v ďalšom texte podrobne popísané, rozanalyzované a následne sú na tieto techniky vytvorené detekčné pravidlá. Tieto pravidlá budú potom vedieť poskytnúť analytikovi malwaru rýchly náhľad na skúmaný súbor a skrátiť čas potrebný na jeho analýzu. Techniky sú následne demonštrované na reálnych vzorkách malwaru, prípadne budú vytvorené proof of concept vzorky, na ktorých bude možné detekčné pravidlá otestovať.

Tieto techniky môžu byť realizované v krypteroch, packeroch a protektoroch viacerými spôsobmi v závislosti od toho, v akom programovacom jazyku sú napísané a aké API funkcie používajú. Preto aj detekcia jednotlivých techník musí pozostávať z detekcie jednotlivých podtechník a variánt danej techniky. Výhodou tohto prístupu je, že pri objavení nových a neimplementovaných možností realizácie danej techniky bude jednoduché rozšíriť už existujúci súbor detekčných pravidiel o nové.

2.2 Cuckoo

Cuckoo Sandbox je open-source systém slúžiaci na automatizovanú analýzu malwaru. [13] Tento sandbox poskytuje izolované prostredie, v ktorom je možné bezpečne spustiť podozrivý súbor. Počas celého behu súboru je prostredie monitorované a všetky akcie podozrivého súboru sú zaznamenávané.

Akonáhle sa do Cuckoo vloží súbor na analýzu, súbor sa otvorí v dopredu pripravenom prostredí a spustí sa. Cez webové rozhranie je možné sledovať, čo sa deje na obrazovke sandboxu a dokonca aj systém ovládať. Negatívom tejto funkcionality je, že počas analýzy sa všetky akcie zaznamenávajú a pri interakcii budú všetky úkony užívateľa zaznamenané vo finálnom reporte spolu so všetkými akciami analyzovaného programu. Po ukončení analýzy sa výsledky spracujú na výstup na webové rozhranie. Výsledky obsahujú informácie o všetkých procesoch, ktoré boli počas analýzy spustené, informácie o volaných API funkciách vrátane parametrov s akými boli volané a návratovými hodnotami. Ďalej je možné skúmať záznamy o sieťovej komunikácii, informácie o alokovaných častiach pamäte, prístupy k súborom, prístupy k registrom atď.

2.2.1 Signatúry

Cuckoo umožňuje vytváranie takzvaných signatúr, ktoré popisujú vzor správania charakteristický pre potenciálne škodlivú techniku. Pomocou signatúr je možné kontrolovať napríklad, či analyzovaný súbor hľadal dynamické knižnice charakteristické pre konkrétne virtuálne prostredia, ako napríklad kontrola

prítomnosti `VBoxDisp.dll` by indikovala, že je program emulovaný vo *VirtualBox*-e. Signatúry takto slúžia ako komplementárny zdroj informácií k YARA pravidlám popísaným v podkapitole 2.3.

Keďže je Cuckoo Sandbox open-source, časom vzniklo mnoho signatúr vytvorených komunitou, ktoré je možné nájsť v repozitári [14]. Signatúry vybrané podľa vlastného uváženia a potreby je možné zahrnúť do Cuckoo.

2.2.2 Vytváranie signatúr

Základný popis, ako vytvárať vlastné signatúry, sa nachádza priamo v dokumentácii Cuckoo Sandbox-u [13]. Táto dokumentácia umožňuje pochopiť proces ako vytvoriť jednoduché signatúry, avšak väčšina pokročilejších funkcionalít nie je zdokumentovaná. Pre prehľad všetkých metód triedy `Signature`, ktorá slúži ako abstraktná trieda pre vytvárané signatúry, je možné použiť definície priamo v kóde v oficiálnom repozitári nachádzajúcom sa v [15].

Pre prácu sú podstatné najmä funkcie `on_call` a `on_process`. Tieto funkcie dostávajú niekoľko argumentov, ktorých obsah a štruktúra sa nenachádza v dokumentácii. Keďže Cuckoo Sandbox začne vyhodnocovať jednotlivé signatúry až na konci analýzy, využíva rovnaké dátové štruktúry, ako je možné vidieť v reporte vo formáte `JSON`. Keďže využívanie všetkých dostupných údajov na vytváranie logiky signatúr výrazne rozširuje možnosti autorov signatúr, je žiadúce popísať funkcie a ich parametre, ktoré sú použité v tejto práci.

Funkcia `on_process` sa zavolá jedenkrát pre každý proces vytvorený počas analýzy. Táto funkcia dostane v parametri `process` štruktúru zodpovedajúcu dátam v reporte z analýzy na pozícii `report[behavior][processes][i]`, kde i je iterátor cez objekty údajov jednotlivých procesov.

```
1301        def on_process(self, process):
1302            """Called on process change.
1303
1304            Can be used for cleanup of flags, re-activation of the signature, etc.
1305
1306            @param process: dictionary describing this process
1307            """
1308
```

Obr. 2.1: Definícia metódy `on_process` z abstraktnej triedy `Signature`[15]

Funkcia `on_call` sa zavolá jedenkrát pre každé zavolanie API funkcie, ktoré bolo počas analýzy zaznamenané. Táto funkcia dostane v parametri `process`

rovnakú štruktúru ako funkcia `on_process`. V parametri `call` dostane štruktúru zodpovedajúcu jednej položke zo zoznamu všetkých API volaní jednotlivých procesov, ktorý sa nachádza v `report[behavior][processes][i][calls]`.

```
1282        def on_call(self, call, process):
1283            """Notify signature about API call. Return value determines
1284            if this signature is done or could still match.
1285
1286            Only called if signature is "active".
1287
1288            @param call: logged API call.
1289            @param process: proc object.
1290            """
1291            raise NotImplementedError
```

Obr. 2.2: Definícia metódy `on_call` z abstraktnej triedy `Signature`[15]

Aj keď nie je oficiálna dokumentácia vyčerpávajúca ohľadne manuálu ako vytvárať vlastné signatúry, tak so znalosťou uvedených funkcií je možné s vytváraním signatúr efektívne začať.

2.3 YARA

Na detekovanie malwaru je potrebné nejakým spôsobom popísať jeho vlastnosti. Práve za týmto účelom bol Victorom M. Alvarezom, pracujúcim pod záštitou VirusTotal-u, vytvorený jazyk YARA. Význam tejto skratky je podľa autora „Yet Another Recursive Acronym“, prípadne „Yet Another Ridiculous Acronym“. [16] Tento open-source nástroj umožňuje rýchlo a efektívne popísať statické aj dynamické vlastnosti súborov pomocou pravidiel. Aj napriek tomu, že bola YARA primárne vytvorená za účelom popisovania škodlivých súborov, je možné YARA pravidlá použiť na porovnávanie akýchkoľvek dát, dokonca aj e-mailov alebo sieťového toku. Nástroj YARA aktívne využíva v praxi mnoho veľkých a celosvetovo pôsobiacich spoločností zaoberajúcich sa informačnou bezpečnosťou. [8]

Na obrázku 2.3 je možné vidieť ukážku ako môže YARA pravidlo vyzerať. V tomto prípade sa jedná o pravidlo na detekciu malwaru s názvom *BadMalware*. Je možné vidieť, že pravidlo pozostáva z viacerých častí a má definovanú štruktúru, ktorá je popísaná v nasledujúcom texte.

2.3.1 Štruktúra YARA pravidiel

Yara pravidlo môže pozostávať z troch sekcií – `meta`, `strings` a `condition`. Jediná povinná sekcia v YARA pravidle je sekcia `condition`, ktorá obsahuje logiku daného pravidla. Funkcionality a význam jednotlivých sekcií je popísaný v nasledujúcich podkapitolách.

2.3.1.1 Sekcia `meta`

Sekcia `meta` obsahuje všeobecné informácie o pravidle, ako napríklad meno autora, popis pravidla, údaje o tom, ktorú rodinu pravidlo detekuje a ďalšie. Sekcia obsahuje dvojice identifikátor a hodnota, pričom hodnota môže pozostávať

```
import "pe"
import "math"
import "cuckoo"

rule demonstration_rule
{
    meta:
        author = "Jakub Kaloc"
        description = "Detection based on known sequences and behaviour"
        strain = "BadMalware"
        type = "RAT"
        rule_type = "mixed"
        hash = "1caf8ace48bf138e390be1b2ab2f6094719193c1c2dff6043f0122af5d1593d1"
    strings:
        //string characteristic for BadMalware
        $s00 = "I am BadMalware"
        //sequence characteristic for malware BadMalware
        $h00 = { 00 11 22 33 ?? 55 6? [4-6] BB AA }
        //matches all variants of this string in base64
        $s01 = "top secret information!" base64
    condition:
        (
            any of ($s0*) and
            $h00
        ) or
        (
            for any section in pe.sections :
            (
                section.raw_data_size > 0 and
                math.entropy(section.raw_data_offset, section.raw_data_size) > 6.5

            )
        ) or
        cuckoo.network.http_request(/BadMalwareCnC\.com/)
}
```

Obr. 2.3: ukážka YARA pravidla

z *UTF-8* textových reťazcov, alebo booleovských hodnôt `true/false`. [17] Táto sekcia je nepovinná a funkčnosť pravidla neovplyvňuje, avšak pri systematickom vytváraní a nasledujúcej práci s väčším počtom pravidiel je podstatná. Metadáta môžu byť použité pre pochopenie funkcionality pravidla pri čítaní iným analytikom alebo aj pri ďalšom spracovávaní a klasifikácii vzoriek, na ktoré pravidlá reagovali. Pravidlá môžu obsahovať nasledujúce údaje, ktoré je žiadúce vyplniť, pokiaľ sú pre charakter pravidla relevantné:

- `author` – hodnota pri tomto identifikátore obsahuje meno autora, prípadne autorov, ktorí sa podieľali na vytváraní YARA pravidla.
- `description` – hodnota pri tomto identifikátore obsahuje stručný popis funkcionality pravidla.
- `strain` – hodnota pri tomto identifikátore obsahuje názov rodiny malwaru, ktorú YARA pravidlo detekuje. Táto položka sa používa v prípadoch, že je dané pravidlo zamerané na detekciu konkrétnej rodiny a nie všeobecnej techniky.
- `type` – hodnota pri tomto identifikátore obsahuje typ malwaru. Typ malwaru je všeobecnejší ako daná rodina a popisuje ciele malwaru. Identifikátor môže nadobúdať hodnoty ako bot, RAT[1], ransomware atď.
- `rule_type` – hodnota pri tomto identifikátore slúži na rozlíšenie statických a dynamických pravidiel. Dynamické pravidlá sú také YARA pravidlá, ktoré využívajú reporty z dynamickej analýzy v sandboxe.
- `hash` – hodnota pri tomto identifikátore obsahuje hash analyzovanej vzorky, na základe ktorej bolo pravidlo vytvorené.
- `reference` – hodnota pri tomto identifikátore obsahuje odkaz na dodatočné informácie. Tento identifikátor je vhodné využívať pri detekcii konkrétnych techník malwaru na odkázanie sa na zdroj informácií o danej technike.

2.3.1.2 Sekcia `strings`

V sekcii `strings` je možné definovať reťazce. Základným typom reťazca je obyčajný text v úvodzovkách. Čo však pomáha z YARY robiť silný nástroj sú ďalšie možné modifikátory textových reťazcov. K ľubovoľnému textovému reťazcu je možné pridať napríklad tieto kľúčové slová, ktoré majú nasledujúce efekty[17]:

- `ascii` – hľadá zhodu len s *ASCII* verziou reťazca

[1]Remote Access Trojan

- `wide` – hľadá zhodu s reťazcom uloženým v dvojbytovom formáte. Príkladom je reťazec „YARA“ uložený ako `Y\x00A\x00R\x00A\x00`[2]. S takýmito reťazcami sa je možné stretnúť pri vzorkách napísaných pomocou .NET.
- `base64` – hľadá všetky 3 možné base64 verzie reťazca. Tento modifikátor je užitočný najmä v rámci obfuskácií, kde je časté, že sa dáta transformujú práve na formát base64.
- `xor` – hľadá všetkých 256 možných verzií reťazca, na ktorý bola aplikovaná operácia XOR s jedným bytom.
- `nocase` – hľadá všetky verzie reťazca bez rozlišovania veľkých a malých písmen.

Ďalšia možnosť je zadávať reťazce v hexadecimálnej forme. Pridanou hodnotou je možnosť zadávať do takýchto reťazcov zástupné znaky (wildcard characters), ktoré reprezentujú ľubovoľný byte alebo aj intervaly, ktoré reprezentujú sekciu s variabilným počtom ľubovoľných bytov. Reťazce v takejto forme majú veľké využitie, pokiaľ chce analytik vytvárať detekciu na konkrétne časti samotného programu, kde popíše inštrukcie, a variabilitu v adresách a skokoch vyrieši práve zástupnými znakmi.

Posledná možnosť, ako zadávať reťazce, je zadefinovať popis reťazcov pomocou regulárneho výrazu. Hľadanie zhody pre regulárny výraz je zo všetkých troch doteraz uvedených spôsobov zadávania najpomalší. Na druhú stranu je však vďaka regulárnym výrazom možné popísať aj zložité štruktúry a pravidlo spraviť dostatočne všeobecné, aby pokrývalo čo najobšírnejšie možnú variabilitu medzi vzorkami. Nástroj YARA používa svoje vlastné spracovávanie regulárnych výrazov. Väčšina prvkov regulárnych výrazov je prebraná z knižnice PCRE (Perl Compatible Regular Expressions), ale niektoré prvky sú prebrané z POSIX regulárnych výrazov.[17]

2.3.1.3 Sekcia `condition`

Sekcia `condition` popisuje, kedy YARA pravidlo niečo detekuje a kedy nie. Celá sekcia je vyhodnocovaná ako jeden pravdivostný výrok, pričom pravidlo detekuje vzorku práve vtedy, keď je výrok vyhodnotený ako pravdivý. Vo výroku je možné používať negáciu, konjunkciu, disjunkciu alebo určovanie prednosti vyhodnotenia pomocou zátvoriek. V podmienke je možné používať aj matematické alebo bitové operácie, čo je užitočné najmä pri práci s konkrétnymi adresami (napr. `adress_of_entrypoint == 0x123`). V podmienke je ďalej možné určovať minimálne a maximálne počty reťazcov vyskytujúcich sa v súbore. Napríklad `3 of ($s0, $s1, $s2, $s3, $s4)` je vyhodnotené

[2] zápis ako v jazyku C

ako pravda, pokiaľ sa v súbore nachádzajú aspoň 3 z uvedených reťazcov zadefinovaných v sekcii `strings`. Medzi posledné výrazné funkcionality, ktoré sú využiteľné v tejto sekcii, patria `for` cykly, kedy je možné iterovať napríklad cez jednotlivé reťazce a dotazovať sa na ich početnosť.

2.3.2 Moduly a rozšírenia

Okrem základných vlastností a funkcionalít ponúka YARA možnosť rozširovania pomocou modulov. Moduly umožňujú definíciu vlastných štruktúr alebo funkcií, s ktorými je následne možné efektívne a jednoducho pracovať pri vytváraní zložitejších podmienok. Viacero modulov je zdokumentovaných a spravovaných priamo VirusTotal-om, avšak priamo v dokumentácii je možné nájsť podrobný návod, ako si vytvoriť vlastné moduly podľa potreby. [18]

2.3.2.1 Modul PE

Jeden z najčastejšie používaných modulov je modul `PE`. Ako už názov modulu naznačuje, modul je schopný pracovať s `PE` formátom a implementuje dátové štruktúry, pomocou ktorých je možné pracovať s mnohými hodnotami z `PE` hlavičky. Príkladom môže byť dotazovanie sa na názvy sekcií, iterovanie cez jednotlivé sekcie, práca s adresou entry pointu, importami alebo zdrojmi.

2.3.2.2 Modul Cuckoo a behaviorálna detekcia

Veľkou slabinou YARY je jej obmedzenie na statické skenovanie súborov a úplné vynechanie informácií o tom, ako sa súbor správa po spustení. Túto slabinu rieši Cuckoo modul, ktorý umožňuje vytvárať behaviorálne pravidlá. S týmto modulom YARA sama o sebe stále nemení princíp na akom funguje a nestáva sa z nej nový druh sandboxu, ale využíva reporty vygenerované sandboxom Cuckoo. Tento sandbox je schopný vygenerovať report vo formáte `JSON`, ktorý podrobne popisuje správanie sa programu. YARA s modulom Cuckoo je následne schopná spracovať tento report a dohľadávať v ňom konkrétne informácie. Týmto rozšírením vie YARA všestranne popísať škodlivé súbory, a keďže sa jedná o open-source nástroj, akékoľvek obmedzenia alebo chýbajúce vlastnosti je možné zlepšiť vlastnými, prípadne komunitnými, rozšíreniami.

2.3.3 Rozšírenie Cuckoo modulu

Pre potreby práce bol Cuckoo modul v nástroji YARA rozšírený o funkciu s názvom `signatures`. Táto funkcia prijíma ako argument textový reťazec s názvom signatúry a vyhodnotí sa ako `true`, pokiaľ sa daná signatúra nachádza v reporte analyzovanej vzorky. Kód rozšírenia je možné nájsť v prílohách v priečinku `attachments/yara_modifications`. Funkcia je deklarovaná nasledovne:

```
begin_declarations
    ...
    declare_function("signature", "s", "i", signature_match)
end_declarations
```

V Cuckoo module sa ukazovateľ na dekódovaný JSON report z analýzy nachádza v štruktúre `YR_OBJECT* module_object` v rámci `module_object->data`. K tejto štruktúre sa v rámci funkcie `signature_match` pristúpi a načíta sa z nej zoznam objektov reprezentujúcich jednotlivé signatúry. Tieto objekty sa následne prehľadávajú a hľadá sa zhoda s názvom signatúry, ktoré je v YARA pravidle. Kľúčové časti funkcie vyzerajú nasledovne:

```
json_t * signatures = json_object_get(
                        module_object->data,
                        "signatures");
...
char * received_signature_name = string_argument(1);
...
json_array_foreach(signatures, index, value)
{
    json_t * signature_name_json = json_object_get(value, "name");
    ...
    if (strcasecmp(received_signature_name, signature_name_str)
            == 0)
    {
        result = 1;
        return_integer(result);
    }
}
...
```

Toto rozšírenie je vložené do zdrojového kódu Cuckoo modulu, ktorý sa nachádza v súbore `yara/libyara/modules/cuckoo/cuckoo.c`

Inštalácia rozšíreného modulu spočíva v spustení príkazu `make` a potom druhého príkazu `sudo make install`. Podrobnejšie informácie je možné nájsť priamo v dokumentácii [19].

Po rozšírení Cuckoo modulu v nástroji YARA sa otvárajú nové možnosti, ako je možné písať YARA pravidlá. Všetko správanie programu zistené počas dynamickej analýzy v Cuckoo je možné popísať signatúrami a následne tieto signatúry zahŕňať do pravidiel. Vďaka tomu je možné skombinovať napríklad podozrivé statické sekvencie, ktoré však samé o sebe len vzbudzujú podozrenie na škodlivý súbor spolu so signatúrami, ktoré môžu popisovať napríklad

techniku injekcie. Takýmto kombinovaním je možné vytvárať oveľa presnejšie a efektívnejšie detekcie a popísať v YARE techniky, ktoré by dovtedy nebolo možné popísať.

2.3.4 YARA v Cuckoo

Jednou z funkcionalít Cuckoo Sandboxu je schopnosť využívania YARY. V rámci analýzy vie Cuckoo preskenovať pôvodne vložený súbor a súbory, ktoré analyzovaný program vytvoril v systéme. Pokiaľ Cuckoo zachytilo prácu s pamäťou, vie preskenovať aj zachytené časti operačnej pamäte. Vďaka tejto funkcionalite je možné identifikovať typ a rodinu malwaru, pokiaľ pre ňu existujú spoľahlivé YARA pravidlá. Bežnou praxou ransomwaru je vytvorenie súboru v napadnutom systéme s informáciami o tom, čo sa stalo a ako zaplatiť požadované výkupné na obnovenie súborov. Pokiaľ existujú YARA pravidlá popisujúce takéto súbory, malware bude pomocou týchto pravidiel a Cuckoo analýzy rýchlo a plne automaticky identifikovaný.

2.3.5 Monitorovanie nových pravidiel

Keďže vo svete existujú milióny súborov a tisíce nových pribúdajú každý deň, je očakávateľné, že niektoré vytvorené detekčné YARA pravidlá budú zachytávať aj nečakané súbory. Keďže môže ísť aj o detekciu legitímnych a neškodných súborov, je žiadúce nové pravidlá monitorovať. Doba monitorovania sa môže výrazne líšiť v súvislosti s počtom súborov, na ktorých sa pravidlo testuje. Jeden z možných spôsobov testovania je pomocou služby *VT Hunting*. Jedná sa o službu[20], ktorá používa YARU nad datasetom, ktorý má VirusTotal. Tento dataset je možné považovať za dostatočne rozsiahly na testovanie pravidiel a pri súboroch je možné rýchlo pozorovať, ktoré antivírusové riešenia detekujú daný súbor. Služba so sebou prináša aj možnosť stiahnuť si detekované súbory na vlastnú lokálnu analýzu.

Pokiaľ sa ukáže, že pravidlo detekuje súbory, ktoré nemá, je potrebné pravidlo prerobiť. Jednoduchým riešením je bližšie špecifikovať súbory pridaním ďalších podmienok a orezaním priestoru súborov pomocou pridaných príznakov a vlastností.

Kapitola **3**

Obfuskačné techniky

Obfuskovanie častí programu prináša pre tvorcov krypterov a protektorov zásadnú výhodu, a to dlhší čas bez analýzy a detekcie ich programu. Keďže je *stub* protektoru prvou vrstvou škodlivého programu, práve táto vrstva podlieha všetkým typom analýz – statická, automatická, dynamická a manuálna. Pokiaľ je kód dobre čitateľný, tak analytik vie zhodnotiť o aký program ide oveľa rýchlejšie ako keby bol obfuskovaný. Zároveň je jednoduchšie vytvoriť detekčné YARA pravidlá na originálny neobfuskovaný kód, ktorý je rovnaký naprieč všetkými vzorkami, ako na kód, ktorý sa pri kvalitnom obfuskovaní mení v každej vzorke.

Obfuskovanie súborov alebo informácií má v MITRE ATT&CK matici vlastnú kategóriu s ID `T1027`, ktorá je ďalej delená na niekoľko podtechník podľa druhu obfuskácie.[11]

3.1 Steganografia

Jedna z techník, ktoré môžu kryptery a protektory používať na ukrývanie payloadu je steganografia. Jedná sa o spôsob ukrývania informácie do objektu, ktorý nie je tajný a nevzbudzuje dojem, že ukrýva informácie.[21] Steganografia má vlastnú podkategóriu v rámci MITRE ATT&CK matice s ID `T1027.003`. [11] Ako sa ukáže na príklade, v praxi je možné stretnúť sa s kryptermi a protektormi, ktoré ukrývajú payload v rámci obrázkov.

3.1.1 Cassandra Crypter

Jednoduchou, no veľavravnou ukážkou použitia techník steganografie je protektor *Cassandra Crypter* alebo inak nazývaný aj *CyaX Crypter*. Tento protektor bol popísaný už koncom roku 2019 v článku [22] a o rok na to znovu v článku [23], avšak protektor je stále aktívny a autori generujú nové verzie.

3.1.1.1 Analýza ukrývania informácií

V novšej verzii protektoru priloženej v prílohe, ktorá bola prvýkrát videná v polovici roku 2021, je možné vidieť, že autori protektoru si dávajú námahu vytvárať relatívne veľké programy, ktoré majú pôsobiť na prvý pohľad vierohodne. V práci je analyzovaná vzorka umiestnená v prílohách v priečinku `attachments/samples/Cassandra Crypter/`:

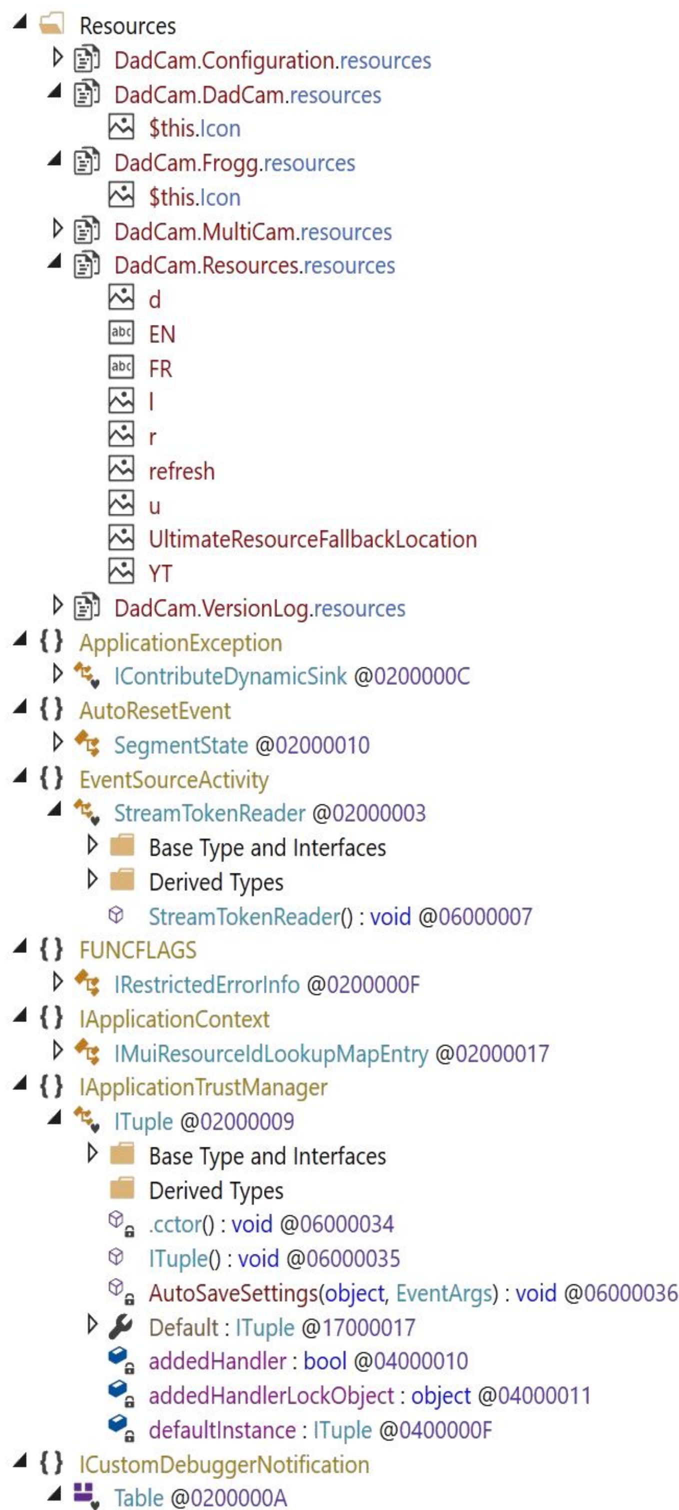
`fc40a65deca750ecaf6d9f0d69abe788ba2febc8f366cdd55fcb97cafe171b3d`

Po otvorení vzorky v nástroji *CFF Explorer* je možné vidieť, že má vyplnené informácie o assembly, kde sa píše, že program sa volá `DadCam` a autorské práva sa odkazujú na web `frogg.fr`. Po otvorení vzorky v *dnSpy* a náhľade do zdrojov si je možné všimnúť, že obrázky, ktoré sa tam nachádzajú, je vidieť aj na zadanej webovej adrese. Okrem toho sa v zdrojoch nachádzajú XML súbory obsahujúce nezaujímavé textové reťazce potrebné k behu programu `DadCam` a aj rýchly náhľad na funkcie povie len to, že program zrejme nerobí nič zaujímavé, keďže nie je vidieť žiadne obfuskácie mien funkcií. Jediné, čo zaujme analytika, ktorý vie, čo môže očakávať, je zdanlivo pokazený a zašumený obrázok s názvom `UltimateResourceFallbackLocation` a zdroj s názvom `YT`, kde sa obrázok na prvý pohľad nezobrazuje, pretože obrázok má výšku len jeden pixel, takže ho takmer nevidieť.

Ako sa ukáže pri podrobnejšej analýze, tak zdroj `YT` je použitý vo funkcii `IContributeDynamicSink.IContributeDynamicSink`, ktorú je možné vidieť na obrázku 3.1. Funkcia prejde každý pixel obrázku, z RGBA hodnôt každého pixelu si vyberie hodnotu červenej farby a tú vkladá postupne do poľa. Takýmto spôsobom protektor získa dynamickú knižnicu, ktorú načíta do pamäte a spustí. Pri analýze si je možné všimnúť aj fakt, že sa zdroj s názvom `UltimateResourceFallbackLocation` nepoužíva v programe nikde.

V spustenej dynamickej knižnici sa následne vykoná funkcia, ktorá má názov `SimpleUI::MDI.SelectorX`. Táto funkcia si načíta do pamäte zdroj, ktorý sa volá `UltimateResourceFallbackLocation`. Obrázok sa interpretuje pixel za pixelom a RGBA zložky sú ukladané do alokovaného poľa. Toto pole spolu s kľúčom následne dostane dešifrovacia funkcia, ktorá dešifruje každý získaný byte z obrázku pomocou operácie `xor` s kľúčom a `xor` s hodnotou odvodenou z posledného bytu v dátach.

Ako je možné vidieť, šifrovací algoritmus na ukrytie dát nie je kryptograficky silný, ale autor programu sa snažil primárne ukryť jednak podozrivý kód medzi nevinne vyzerajúce funkcie a ukryť ďalšie etapy programu medzi iné, obyčajné obrázky. Ako sa píše v článku [23], tento protektor používa viacero jednoduchých variánt šifrovacích algoritmov, pričom sú známe aj pokročilejšie



Obr. 3.1: náhľad na *stub* Cassandra Crypter-u č. 1

steganografické varianty, kde sa dáta ukrývajú ako najmenej významný bit RGB zložiek jednotlivých pixelov.

3.1.1.2 Detekcia techniky

Niektoré techniky ukrývania dát do obrázkov je možné detekovať pomocou nástroja YARA na základe entropie obrázkov. Je treba mať na mysli, že úspešnosť detekcie závisí aj od spôsobu uloženia obfuskovaných dát. Entropiu je totiž možné znižovať vkladaním blokov prázdnych dát, napríklad striedanie blokov nulových bytov s pôvodnými obfuskovanými dátami.

Na detekciu tejto techniky boli vytvorené statické YARA pravidlá, ktoré je možné nájsť v prílohách v priečinku `attachments/yara_rules` v súbore s názvom `T1027_003_Steganography.yar`. V súbore je možné vidieť pravidlo zodpovedajúce štruktúre MITRE ATT&CK matice, ktoré má v podmienke privátne pravidlo. Detekcia je riešená týmto spôsobom kvôli možnej prehľadnej rozšíriteľnosti.

```
public IContributeDynamicSink(bool c1)
{
    int num = 0;
    Bitmap yt = ComDefaultInterfaceAttribute.YT;
    byte[] array = new byte[51713];
    int num2 = 0;
    checked
    {
        do
        {
            int num3 = 0;
            do
            {
                Color pixel = yt.GetPixel(num2, num3);
                int num4 = ColorTranslator.ToWin32(pixel);
                array[num] = (byte)num4;
                num3++;
            }
            while (num3 <= 0);
            num++;
            num2++;
        }
        while (num2 <= 51711);
        this.MessageSurrogateFilter(array, new Font("Arial", 12f), "", 0f);
    }
}
```

Obr. 3.2: náhľad na *stub* Cassandra Crypter-u č. 2

Privátne pravidlo s názvom `DOTNET_HIGH_IMAGE_ENTROPY` detekuje obrázky uložené v .NET zdrojoch. V pravidle sa iteruje cez všetky zdroje programu a vyhľadáva v nich signatúra bitmapového obrázku. Pokiaľ sa nájde, spočíta sa entropia tohto zdroja. YARA počíta entropiu v rámci modulu `math`, pričom vrátený výsledok je `float` hodnota reprezentujúca počet bitov informácie na byte, teda maximálna entropia je 8.0. Pokiaľ spočítaná entropia obrázku presiahne hranicu 6.5 (na základe článku [24]), pravidlo vráti hodnotu `True`.

Rozširovať tieto pravidlá ďalej je možné nasledujúcimi spôsobmi:

- Pridávanie nových signatúr pre ďalšie formáty obrázkov do privátneho pravidla `DOTNET_HIGH_IMAGE_ENTROPY`.
- Vytvorenie nových privátnych pravidiel pre súbory iných formátov.

Pri vytváraní detekcií je vždy potrebné mať na mysli, aké súbory môžu detekovať a aké sú ich možnosti využitia. Potenciálnym problémom tejto detekcie môžu byť komprimované obrázky. Kompresia slúži na zníženie veľkosti obrázku, a teda rovnaký obsah informácií sa po kompresii musí zmestiť do menšieho objemu dát, čím dochádza k zvyšovaniu priemernej entropie. Pokiaľ bude nejaký .NET program obsahovať v zdrojoch takéto obrázky, tak budú spúšťať vytvorené YARA pravidlo, aj keď sa v nich nenachádza nič škodlivé. Pre tento dôvod nie je vhodné používať toto pravidlo ako jediný rozhodujúci faktor pri určovaní, či je súbor škodlivý alebo nie. Pravidlo však môže slúžiť ako veľmi dobrá doplňujúca podmienka pri vytváraní iných pravidiel. Pokiaľ sa podarí popísať správanie programu kombináciou niekoľkých vlastností, ktoré samé o sebe nemusia byť unikátne, ale zároveň ich používa len malware alebo dokonca len konkrétna rodina malwaru, majú takéto pravidlá veľký význam a využitie.

Detekcia bola otestovaná na analyzovanej vzorke a následne na troch ďalších nezávislých vzorkách, ktoré sú chránené rovnakým protektorom, ale obsahujú rôzne payloady. Tieto testovacie vzorky je možné nájsť v prílohách v priečinku `attachments/samples/Cassandra Crypter/test/`. Detekcia funguje na všetkých testovaných vzorkách.

3.2 Obfuskácia názvov v .NET-e

Programy, ktoré boli pôvodne písané v jazyku C# alebo Visual Basic, je možné disasemblovať do assembly jazyku MSIL. Tento jazyk je čitateľný pre ľudí, avšak zároveň obsahuje všetky informácie potrebné na úspešnú dekompiláciu do zdrojového kódu. [25] V súčasnosti existuje viacero programov, ktoré sú schopné .NET binárny súbor dekompilovať, avšak v práci je na tento účel

používaný nástroj *dnSpy*[3]. V tomto nástroji je možné prezerať a staticky analyzovať disasemblovanú a dekompilovanú verziu programu a zároveň aj program spustiť a analyzovať ho dynamicky.

Keďže v dekompilovanej verzii .NET programov je možné vidieť pôvodné názvy funkcií, parametrov funkcií a aj premenných, tak bez ochrany týchto informácií sa analytikovi značne uľahčuje práca. Práve preto sa takéto programy často obfuskujú. Základné obfuskovanie .NET programov spočíva v premenovaní pôvodných názvov na rôzne a často náhodné textové reťazce. Tieto reťazce môžu pozostávať z nezobraziteľných znakov, ktoré dekompilátor následne vypíše iba ako reťazec hexadecimálnych hodnôt (napr. v podkapitole 3.3). Ďalšou možnosťou je použiť reťazce pozostávajúce zo znakov iných abecied (ako napríklad čínske znaky), ktoré analytikov neovládajúcich danú abecedu mýlia ešte viac, keďže pri náhodnom reťazci je stále možné rozlišovať jednotlivé reťazce a znaky medzi sebou, ale pri čínskych názvoch je tento úkon oveľa náročnejší pre podobnosť jednotlivých znakov. Takéto obfuskácie používa napríklad Morpheus Crypter, ktorý je analyzovaný v podkapitole 4.3.

3.2.1 Prekonanie obfuskácie názvov v .NET-e

Jednou z možných techník deobfuskovania je manuálne premenovávanie častí programu, ktoré analytik už zanalyzoval, avšak tento proces je zdĺhavý a náročný. Efektívne vie pomôcť open-source nástroj *de4dot*, ktorý sa snaží obnoviť obfuskované programy do pôvodnej formy. V súčasnosti podporuje devätnásť známych packerov a obfuskátorov[26], ktorých chránené programy vie *de4dot* vrátiť do pôvodnej formy. Pokiaľ nástroj nepozná použitý obfuskátor, premenuje systematicky obfuskované názvy podľa toho, čo reprezentujú a koľké premenovanie v poradí to je – `Class0, Class1, string_0, string_1`... Tento nástroj je používaný aj pri analýze obfuskovaných vzoriek v rámci tejto práce.

3.3 ConfuserEx 2

ConfuserEx 2 je open-source nástroj pre .NET aplikácie verejne dostupný na GitHub-e.[27] Tento nástroj je vytvorený na základe svojich dvoch predchodcov – protektorov *ConfuserEx* a *Confuser*. [28] Ako už preklad názvu nástroja naznačuje, jeho hlavným cieľom je obfuskácia a ochrana .NET aplikácií. Zatiaľ čo niektoré kryptery a protektory implementujú vlastné obfuskácie, aby sťažili analýzu, autori iných protektorov si môžu uľahčiť túto prácu využitím už vyvinutého obfuskátora. Napriek tomu, že *stub* protektoru môže (a pravdepodobne aj bude) implementovať vlastné techniky obrany pred analýzou, nástroj ako ConfuserEx 2 pridá ešte ďalšiu vrstvu obrany.

[3]dostupný z `https://github.com/dnSpy/dnSpy`

```
namespace 曦羵磹竊礵艷艄竃磬纏禠粡搝曦礫磋箘磝礲磄舭禠獕襀曦
{
    // Token: 0x02000003 RID: 3
    internal class 曦纚箕磬礰磝矘礛礀艷磄穜磖磜磓磕磖禨禮籣囊礀礀
    {
        // Token: 0x06000003 RID: 3
        [DllImport("矘筳粠磩稿磩裺襀粡艒曦囊竃磝礀囊鲍禥艴艷艒曦筳艷褚靨
        磨艪禠瞖羵騊竊磩羵穜礐羵曦磖竡曦粿笺襀瞖磄礰.dll")]
        private static extern string 矘褚艞驋粡磓羅笏艒粡襜礒磽竃褸鲍樲
        簹籣磼礝鲜磋艒粡艞礲曦靨砥艪磼纊磜搝笏簹礲礦矘礐礲艳礛礀粠礵篙雙
        稿艷粡磬羵磈窣(IntPtr 曦羵艄艭艭羅磈磖磬襜糒磋礝磄穜穦裺鲜粿褚笏
        砥磋磬礀礒稿骳礛磈纊稿搝磕羵粠磈舭筳禠纚矓, byte 砥磋磬礀礒稿骳礛
        磈纊稿搝磕羵粠磈舭筳禠纚矓曦羵艄艭艭羅磈磖磬襜糒磋礝磄穜穦裺鲜粿
        褚笏);

        // Token: 0x06000004 RID: 4
        [DllImport("矘筳粠磩稿磩裺襀粡艒曦囊竃磝礀囊鲍禥艴艷艒矘褚艞驋粡
        磓羅笏艒粡襜礒磽竃褸鲍樲簹籣磼礝鲜磋艒粡艞礲曦靨.dll")]
        private static extern string[] 矘筳粠磩稿磩裺襀粡艒曦囊竃磝礀
        囊鲍禥艴艷艒矘褚舭竡曦礝礒簹磜礰礝礦竡粠礛搝礂礲笏磩糒禥磄礲笐穦
        纊(bool[] 曦磽笺磈磖礀竊瞖礛礦纚矘筳磝竡雙舭艪瞖羅箵礵穜礝襜磄穜
        襀磄曦羵艄艭艭羅磈磖磬襜糒磋礝磄穜穦裺鲜粿褚笏, char 曦羵艄艭艭羅
        磈磖磬襜糒磋礝磄穜穦裺鲜粿褚笏曦磽笺磈磖礀竊瞖礛礦纚矘筳磝竡雙舭
        艪瞖羅箵礵穜礝襜磄穜襀磄);

        // Token: 0x06000005 RID: 5
        [DllImport("矘褚艞驋粡磓羅笏艒粡襜礒磽竃褸鲍樲簹籣磼礝鲜磋艒粡艞
        礲曦靨矘筳粠磩稿磩裺襀粡艒曦囊竃磝礀囊鲍禥艴艷艒.dll")]
        private static extern int 矘磬磄稿禥曦礦磽褸稿箘艴舭磼骳籣笐羅
        粿磋羅磽磩矘粿礦襀磄礐襀磕曦矘笺樲礛笐艴穦礛磕礒瞖囊磓襀窣窣
        (StringComparer 砥磋磹靨礲穜礀粿箘磹褚瞖窣礒磄磼粻羅磃鲜磃艭艭笺
        曦籣襀礐曦磽笺磈磖礀竊瞖礛礦纚矘筳磝竡雙舭艪瞖羅箵礵穜礝襜磄穜襀
        磄, long 曦磽笺磈磖礀竊瞖礛礦纚矘筳磝竡雙舭艪瞖羅箵礵穜礝襜磄穜襀
        磄砥磋磹靨礲穜礀粿箘磹褚瞖窣礒磄磼粻羅磃鲜磃艭艭笺曦籣襀礐);

        // ...
    }
}
```

Obr. 3.3: náhľad na obfuskované názvy Morpheus Crypteru

3.3.1 Prehľad funkcionalít

Funkcionality nástroja ConfuserEx 2 je možné rozdeliť do nasledujúcich štyroch úrovní podľa toho, ako ich je možné navoliť priamo v programe:

- Minimálna ochrana
- Normálna ochrana
- Agresívna ochrana
- Maximálna ochrana

Každá z úrovní ochrany je náročnejšia na analýzu ako tá predošlá, pretože každá úroveň pridáva nové ochrany. Pri každej úrovni je možné ďalej meniť dodatočné nastavenia ochrán a niektoré buď úplne vylúčiť alebo ich naopak pridať.

Keďže na vytvorenie detekcie je potrebná analýza chránených vzoriek, tak pre potreby tejto sekcie boli vytvorené dva jednoduché programy napísané v jazyku C#. Programy sa volajú `HelloWorld` a `HelloWorld2` a je možné nájsť ich zdrojový kód aj skompilovaný binárny súbor v prílohách. Programy obsahujú ukážkové funkcie, triedy, premenné a textové reťazce, aby bolo možné pozorovať, ako sa súbory zmenia po použití nástroja ConfuserEx 2. V nasledujúcom texte sú analyzované tieto vybrané funkcionality za účelom poskytnutia informácií potrebných na ich prekonanie:

- Ochrana pred debuggovaním
- Premenovanie názvov
- Ochrana konštánt
- Pridanie proxy funkcií
- Zmena toku riadenia
- Ochrana pred modifikáciami programu

Je veľmi nepravdepodobné, že by sa z nástroja ConfuserEx používali len jednotlivé funkcionality, už len preto, že konfigurácia je zložitejšia. Oveľa jednoduchšie je používať už predvolené konfigurácie v rámci jednotlivých úrovní ochrany. Napriek tomu je pre zrozumiteľnosť a pochopiteľnosť analýza robená na vzorkách, kde sa vždy používa práve jedna ochrana naraz. Naopak, pri vytváraní detekcie pomocou jazyka YARA boli použité vzorky, ktoré boli chránené jednotlivými úrovňami ochrany, kde je možné vidieť komplexitu jednotlivých ochrán a náročnosť ich analýzy.

3.3.1.1 Ochrana pred debuggovaním

Po otvorení vzorky `HelloWorld2_antidbg` v nástroji *dnSpy* je možné vidieť, že bol do konštruktoru modulu pridaný nový kód. Využiť konštruktor modulu na pridanie dodatočného kódu do programu je efektívny nápad ako spustiť pridaný kód ešte pred spustením samotného programu. Pokiaľ teda tento pridaný kód obsahuje nejaké ochrany pred analýzou, tak môže dôjsť k ukončeniu programu ešte pred spustením chráneného kódu.

Pri analýze kódu v konštruktore je možné vidieť, že pribudla anti-debuggovacia ochrana. Ako prvá sa kontroluje premenná prostredia `COR_ENABLE_PROFILING`, ktorá kontroluje prítomnosť profilovacieho nástroja. Profilovacie nástroje slúžia na monitorovanie priebehu aplikácií. Bežné profilovacie nástroje môžu monitorovať výkon jednotlivých častí aplikácie, ako napríklad čas strávený v jednotlivých funkciách alebo využívanie pamäte. [29] Pokiaľ je hodnota premennej `COR_ENABLE_PROFILING` nastavená na 1, program sa ukončí.

Po tejto kontrole sa spustí nové vlákno, ktoré v pravidelných časových intervaloch kontroluje prítomnosť debuggeru. Pokiaľ je prítomnosť debuggeru zistená, tak sa celý program ukončí.

3.3.1.2 Premenovanie názvov

Táto funkcionalita premenuje názvy premenných, parametrov, funkcií a tried na *UNICODE* reťazce, ktoré dekompilátor nedokáže zobraziť. Napriek tomu, že sa informácia o pôvodných názvoch stratí, pre zvýšenie čitateľnosti kódu je možné použiť nástroj *de4dot*, ktorý nezobraziteľné znaky odstráni, ako je vysvetlené v podkapitole 3.2.

3.3.1.3 Ochrana konštánt

Ochrana konštánt je funkcionalita, ktorá zašifruje všetky konštanty v programe. Funkcionalita vie značne sťažiť prácu analytikovi, keďže pri analýze stratí náhľad na všetky textové reťazce, bez ktorých je oveľa náročnejšie rýchlo odhadnúť, čo program alebo jednotlivé funkcie robia. Táto ochrana vie byť účinná aj pred statickými detekciami, ktoré môžu byť založené na výskyte niekoľkých známych textových reťazcov v programe.

Keďže sú textové reťazce, použité v programe, veľmi nápomocné pri analýze, je dobré vedieť túto ochranu prekonať. Jedna možnosť je statické dešifrovanie konštánt a prípadná následná automatizácia skriptom. Keďže je však obfuskácia relatívne komplikovaná a pre analytikov je potrebné minimalizovať investovaný čas do analýz, je jednoduchšie dostať z programu textové reťazce dynamicky.

V konštruktore modulu je niekoľko funkcií, ktoré sa líšia len konštantami a používajú funkciu `Encoding.UTF8.GetString`. Pokiaľ sa na tieto miesta umiestni breakpoint, tak pri návrate z `Encoding.UTF8.GetString` je možné vidieť v premennej `result` dešifrovaný textový reťazec.

```
result = (-)((object)string.Intern(Encoding.UTF8.GetString(<Module>.
\u202E\u200E\u206C\u202B\u206C\u202D\u202A\u200B\u202B\u200E\u206E\u
202D\u200C\u200C\u200C\u206F\u200D\u202A\u206D\u202C\u202A\u200E\u20
0D\u200B\u202C\u200B\u202B\u206E\u206A\u200B\u206E\u206B\u202B\u202D
\u206B\u200F\u200F\u202C\u200E\u206F\u202E, A_0 + 4, count)));
```

Obr. 3.4: extrakcia textových reťazcov zo vzorky ConfuserEx 2

3.3.1.4 Pridanie proxy funkcií

Táto ochrana identifikuje v programe miesta, kde sa volajú známe API funkcie a vytvorí pre tieto funkcie novú samostatnú funkciu s obfuskovaným názvom, ktorá predá parametre originálnej funkcii. Prvá analyzovaná ochrana (premenovanie názvov) tieto názvy z kódu neodstraňovala. Táto ochrana ich tiež síce neodstráni úplne, no ukryje ich, aby sa takto analytikovi pridal ďalší krok, ktorý musí spraviť pre pochopenie kódu.

Ochranu je možné prekonať premenovaním proxy funkcií podľa názvov funkcií, ktoré obsahujú. Toto však môže byť pri veľkom počte funkcií značne pracné.

3.3.1.5 Zmena toku riadenia

Pri použití ochrany, ktorá zmení tok riadenia, sa analýza pre neskúseného analytika výrazne sťaží. Pri nahliadnutí do pôvodného kódu aplikácie je možné vidieť, že sa kód výrazne zmenil. V pôvodných funkciách došlo k obfuskácii toku riadenia a aj veľmi jednoduchá a krátka funkcia ako `Main` v programe `HelloWorld2` sa stala výrazne dlhšou a neprehľadnou. Zároveň boli medzi časti pôvodného kódu vložené nové inštrukcie, ktoré nemajú žiadny vplyv na originálnu funkcionalitu, avšak zväčšujú množstvo kódu, kde sa pôvodné inštrukcie môžu ľahko stratiť.

Ako je možné vidieť na obrázku 3.5, z obfuskovaného kódu je veľmi náročné zistiť, ktoré inštrukcie sa majú vykonať ako prvé a aj ktoré inštrukcie sú podstatné pre program a ktoré boli len vložené ako súčasť obfuskácie. Pôvodná, neobfuskovaná verzia kódu je naopak veľmi stručná a pochopiteľná:

```
private static void Main(string[] args)
{
    Console.WriteLine("HelloWorld!");
    int num = Program.foo(123);
}
```

Takáto obfuskácia takmer znemožní statickú analýzu. Analytik môže jedine skúmať názvy funkcií, aké premenné a konštanty sa nachádzajú v analyzovanom mieste programu a skúšať odhadovať, čo sa v programe deje na základe volania systémových funkcií, ktorých názvy obfuskované nie sú. Takáto analýza je však často nedostatočná a v kombinácii s predošlými obfuskáciami názvov a pridaním proxy funkcií výrazne sťažená. Preto je vhodné pri použití takejto ochrany analyzovať program dynamicky.

3.3.1.6 Ochrana pred modifikáciami programu

Posledná z analyzovaných funkcionalít je ochrana pred modifikáciami programu. Cieľom tejto ochrany je zabrániť analytikovi alebo komukoľvek inému meniť obsah programu. Táto funkcionalita je veľmi účinná v synergii s premenovaním názvov a pridaním proxy funkcií. Pokiaľ by bol na takto chránený program použitý nástroj *de4dot* alebo by analytik manuálne menil názvy v programe, pôvodný program by sa nespustil. Napriek tomu, že nejde priamo o obfuskačnú techniku, je to technika, ktorá priamo chráni obfuskácie, aby ich nebolo možné jednoducho odstrániť.

Vzhľadom na to, ako je táto funkcionalita implementovaná, tak ako vedľajší efekt pridáva ďalšiu silnú ochranu. Po otvorení programu chráneného touto funkcionalitou nie je možné v dekompilátore vidieť žiadny pôvodný kód, teda statická analýza je nemožná.

Kód ochrany pred modifikáciami programu sa opäť nachádza v konštruktore modulu. Hneď v úvode programu je možné vidieť, že program získa počiatočnú adresu, kde je program načítaný v pamäti. Počnúc od tejto adresy program začne parsovať PE hlavičku s cieľom získať informácie o počte sekcií a ukazovateľ na zoznam hlavičiek sekcií. V nasledujúcej ukážke kódu je možné vidieť okomentovanú realizáciu získania uvedených informácií z PE hlavičky:

```
Module module = typeof(<Module>).Module;
//get image base address
byte* ptr = (byte*)((void*)Marshal.GetHINSTANCE(module));
//get e_lfanew  value - Nt Headers address
byte* ptr2 = ptr + *(uint*)(ptr + 60);
//get number of sections
```

```
ushort num = *(ushort*)(ptr2 + 6);
//get size of Optional Header
ushort num2 = *(ushort*)(ptr2 + 20);
//get address of Section Headers
uint* ptr4 = (uint*)(ptr2 + 24 + num2);
```

Keďže nástroj *dnSpy* neposkytuje vhodný a prehľadný náhľad na parsovanú PE hlavičku, je pri analýze takýchto funkcií dobré použiť nástroj *CFF Explorer*, ktorý toto umožňuje. Po otvorení programu v tomto nástroji je možné vidieť, že v programe sa nachádzajú sekcie `.text`, `.rsrc`, `.reloc`, ktoré sú v programoch bežné, sekcia `WZ4"yG^` a sekcia bez zobrazeného mena. Posledné dve sekcie bežné nie sú. Po nahliadnutí na hexadecimálnu podobu hlavičky sekcií si je možné všimnúť, že názov sekcie `WZ4"yG^` je v skutočnosti dlhý 8 znakov, avšak v poradí šiesty znak má hexadecimálnu hodnotu $0x1F$, ktorá nie je bežne zobraziteľným znakom. Sekcia bez názvu má na mieste názvu samé nulové byty.

Následne sa inicializujú štyri lokálne premenné dopredu nadefinovanými celými číslami, ktoré budú neskôr slúžiť ako počiatočné seed hodnoty.

Funkcia ďalej pokračuje cyklom, ktorý iteruje cez všetky sekcie v zozname hlavičiek sekcií. Na začiatku každej iterácie sa vezme názov sekcie ako dve štvorbytové celé čísla, ktoré sa spolu vynásobia. Následne sa podľa výsledku násobenia rozlišujú tri typy sekcií pomocou podmienky s preddefinovanými hodnotami:

- Sekcia s názvom pozostávajúcim z nulových bytov
- Sekcia s názvom `WZ4"yG^`
- Ostatné sekcie

V prípade sekcie s názvom `WZ4"yG^` sa do premenných uloží adresa začiatku sekcie a dĺžka sekcie. Sekcia s názvom pozostávajúcim z nulových bytov je vynechaná. Nakoniec pri ostatných sekciách sa získa z hlavičky sekcií adresa začiatku danej sekcie a jej dĺžka a vo vnorenom cykle sa začne iterovať cez danú sekciu. V každej iterácii sa použijú štyri byty zo sekcie na mixovanie štyroch lokálnych premenných, ktoré boli na začiatku nadefinované na konkrétne celé čísla. Každé mixovanie pozostáva z nasledujúcich inštrukcií, kde `ptr5` je ukazovateľ na aktuálnu pozíciu v sekcii a čísla `num4`, `num5`, `num6`, `num7` sú spomínané premenné:

```
uint num11 = (num4 ^ *(ptr5++)) + num5 + num6 * num7;
num4 = num5;
num5 = num7;
num7 = num11;
```

```
private static void
\u200F\u200C\u206D\u202E\u200F\u206A\u200F\u206A\u200E\u206F\u202C\u202B\u202B\u200B\u2
00C\u200B\u200B\u200F\u200B\u202D\u200C\u206E\u200E\u200D\u202D\u206D\u200B\u206D\u200C
\u202E\u206F\u206C\u202C\u202E\u200F\u206D\u200B\u200F\u200E\u200F\u202E(string[] A_0)
{
    for (;;)
    {
        IL_01:
        uint num = 1359505401U;
        for (;;)
        {
            uint num2;
            switch ((num2 = (num ^ 960234177U)) % 5U)
            {
            case 0U:
                num = (num2 * 3905876816U ^ 2560750904U);
                continue;
            case 1U:
            {
                int num3 =
                \u202E\u202D\u202C\u202C\u200F\u206D\u202B\u202A\u206E\u202E\u202B\u200
                C\u202C\u202B\u200F\u206F\u206F\u200C\u202D\u200F\u200F\u200B\u200B\u20
                0F\u200B\u202A\u202A\u206E\u202A\u202E\u202E\u200B\u202C\u206B\u202A\u2
                02C\u202E\u202A\u206F\u206B\u202E.
                \u202B\u200F\u200F\u202C\u202B\u206D\u206B\u206F\u206F\u200D\u206B\u202
                D\u206B\u200F\u202D\u200B\u206A\u200D\u206F\u202A\u202E\u206B\u200F\u20
                2E\u206A\u200B\u202B\u200F\u200F\u202E\u200B\u202A\u206F\u202E\u200C\u2
                06D\u200C\u206A\u206F\u200D\u202E(123);
                num = (num2 * 2005349099U ^ 2556792130U);
                continue;
            }
            case 2U:
                goto IL_01;
            case 3U:
                \u202E\u202D\u202C\u202C\u200F\u206D\u202B\u202A\u206E\u202E\u202B\u200
                C\u202C\u202B\u200F\u206F\u206F\u200C\u202D\u200F\u200F\u200B\u200B\u20
                0F\u200B\u202A\u202A\u206E\u202A\u202E\u202E\u200B\u202C\u206B\u202A\u2
                02C\u202E\u202A\u206F\u206B\u202E.
                \u202C\u200E\u206B\u200B\u200D\u202A\u200D\u200D\u202C\u202D\u206A\u200
                E\u200F\u206F\u206A\u200F\u200F\u200E\u200D\u202A\u206F\u200E\u202B\u20
                2C\u200B\u200D\u206F\u206F\u202E\u206A\u202A\u200D\u202E\u202E\u206C\u2
                02C\u200F\u202D\u200F\u206D\u202E(<Module>.
                \u202C\u206A\u206C\u206B\u202B\u206C\u202A\u200C\u206D\u206A\u202C\u202
                B\u202B\u202E\u202B\u206F\u206F\u200C\u200F\u202A\u206D\u200E\u202B\u20
                0B\u206B\u206C\u202B\u200F\u206F\u206B\u202B\u206C\u206F\u202B\u200F\u2
                06D\u202E\u206C\u206C\u202D\u202E<string>(433074626));
                num = 2079930U;
                continue;
            }
            return;
        }
    }
}
```

Obr. 3.5: Obfuskácia toku riadenia nástrojom ConfuserEx 2

V nasledujúcich niekoľkých jednoduchých krokoch sa vytvoria dve celočíselné polia s dĺžkou šestnásť prvkov, ktoré sa inicializujú na základe štyroch hodnôt, ktoré boli v predošlom cykle mixované. Výstupom tohto inicializačného procesu je prvé z dvoch polí, ktoré je ešte raz modifikované pomocou hodnôt v druhom poli.

Predposledný krok spočíva v zavolaní API funkcie `VirtualProtect`. Táto funkcia nastaví sekcii s názvom `WZ4"yG^` konštanty na ochranu pamäte na hodnotu $0x40$, čo je známa konštanta `PAGE_EXECUTE_READWRITE`; adresa začiatku sekcie a jej veľkosť bola zistená a uložená v predošlých krokoch.

Po tom, čo si program zaistí, že vie sekciu čítať a zapisovať do nej, začne cyklus, v ktorom sa iteruje postupne cez celú sekciu s názvom `WZ4"yG^`. V každej iterácii sa na štyri byty v sekcii naxoruje hodnota z poľa inicializovaného v predošlých krokoch. Použitá hodnota z tohto poľa je potom zmenená na novú na základe dešifrovanej hodnoty a konštanty. Priebeh dešifrovania sekcie je možné vidieť v nasledujúcom kóde, pričom `num3` je dĺžka sekcie `WZ4"yG^` a `ptr3` je adresa začiatku sekcie:

```
uint num13 = 0U;
for (uint num14 = 0U; num14 < num3; num14 += 1U)
{
    //decrypt 4 bytes
    *ptr3 ^= array[(int)(num13 & 15U)];
    //mix array and increase pointer to data
    array[(int)(num13 & 15U)] =
        (array[(int)(num13 & 15U)] ^ *(ptr3++)) + 1035675673U;
    //increase counter
    num13 += 1U;
}
```

Funkcionalita na ochranu pred modifikáciami teda spočíva v zašifrovaní pôvodného kódu programu, čo vysvetľuje fakt spomenutý na začiatku, že sa takto chránený kód nedá dekompilovať v *dnSpy*. Pokiaľ sa však analytik dostane pri dynamickej analýze za túto ochranu do bodu, kedy sa exekúcia presunie do pôvodného chráneného kódu, v *dnSpy* sa zobrazí nová assembly, ktorá má túto ochranu odstránenú a je možné analyzovať kód chránených funkcií.

3.3.2 Detekcia

Na pozorovanie zmien a možné odlíšenie randomizovaných a staticky generovaných hodnôt použitých pri jednotlivých ochranných funkcionalitách, ako aj identifikovanie náhodne generovaného kódu, bol každý z testovacích programov `HelloWorld` a `HelloWorld2` obfuskovaný viackrát. Tento prístup pomohol

aj pri písaní detekcií, ktoré sú odolné voči zmenám v adresách a náhodne generovaných hodnotách v jednotlivých vzorkách.

Pri reálnom použití nástroja ConfuserEx 2 je potrebné očakávať, že sú použité viaceré z popísaných funkcionalít zároveň. Pre simulovanie bežného použitia tohto nástroja bol každý z programov `HelloWorld` a `HelloWorld2` obfuskovaný päťkrát na každej úrovni ochrany, ktoré sú prednastavené a ponúkané programom.

Na detekovanie nástroja bolo vytvorené YARA pravidlo, ktoré je možné nájsť v prílohách. Pravidlo je zamerané na detekciu statických sekvencií inštrukcií, prípadne textového reťazca, ktorý do programu nástroj pridá. Sekvencie inštrukcií sú prevzaté priamo z častí kódu viacerých modulov a ochrán, ktoré ConfuserEx 2 pridá do obfuskovaného programu. Pravidlo je zostavené tak, že je schopné zachytiť vzorky obfuskované každou z piatich prednastavených úrovní ochrany.

Na zníženie možných falošne pozitívnych detekcií boli vybrané čo najdlhšie sekvencie inštrukcií. Tento krok je nutný špeciálne pre .NET programy, keďže operačné kódy MSIL inštrukcií majú väčšinou len jeden byte. Ďalej bola do podmienky vytvoreného YARA pravidla pridaná špecifikácia, že pravidlo má detekovať iba súbory, ktoré sú .NET programy. Táto podmienka je vytvorená v rámci samostatného privátneho pravidla, pretože sa jedná o silnú vylučovaciu podmienku, ktorá bude použitá v rovnakej podobe aj v ďalších YARA pravidlách. Pri tejto podmienke je potrebné mať na mysli, že YARA spracováva akékoľvek súbory, bez ohľadu na ich formát alebo obsah, takže zameranie sa výhradne na .NET súbory pomôže odstrániť falošne pozitívne detekcie.

Je možné si všimnúť, že na vytváranie aj testovanie detekcií boli obfuskované jednoduché a neškodné programy, teda detekcia sa spúšťa nezávisle na programe, ktorý je nástrojom ConfuserEx 2 chránený. Pri detekovaní krypterov a obfuskátorov je toto očakávané správanie. Takéto detekcie sú využiteľné jednak ako informácia pre analytika, aby vedel aké ochrany boli na skúmaný program použité a jednak ako informácia pre klasifikáciu alebo ďalšie, komplexnejšie detekčné pravidlá. Pravidlo sa taktiež dá ďalej monitorovať a pokiaľ sa ukáže, že vzorky, ktoré pravidlo detekuje obsahujú len malware, je možné na odhalenie takýchto vzoriek použiť len toto detekčné pravidlo.

Kapitola 4

Slabiny dešifrovacích algoritmov

Účel šifrovania payloadu alebo častí krypteru má rovnaký význam ako použitie steganografie – ukrytie dát pred analytikom a statickým skenovaním. Následne je úlohou analytika prísť na to, ako šifrovanie v rámci vzorky funguje, ako ho prekonať a zistiť, ako krypter alebo protektor funguje a aký malware ukrýva. Napriek všeobecne známemu odporúčaniu „nevymýšlajte si vlastnú kryptografiu“ je možné pozorovať trend svedčiaci o tom, že autori protektorov tak robia (podkapitoly 3.1.1, 4.2 a 4.3). Dôvodov, prečo sú autori malwaru motivovaní implementovať vlastné kryptografické algoritmy, je hneď niekoľko:

1. Na neznámu a nekonvenčnú šifru nie je známy dešifrovací algoritmus a pokiaľ by ho chcel niekto vytvoriť, tak je potrebné zanalyzovať a zreverzovať celú šifru.

2. Bežný analytik zoznámený s tradičnými šiframi môže stráviť oveľa viac času nad analýzou kódu, ktorý ešte nevidel.

3. Šifra nebude lúštená ale reverzovaná a/alebo dešifrovaná. Preto šifra nemusí byť matematicky odolná voči útokom a rôznym lúštiacim technikám. Navyše aj keby bola šifra odolná voči lúšteniu, jej kľúč bude dostupný priamo v programe, keďže program musí byť spustiteľný na infikovanom zariadení.

Vymýšľanie vlastných šifrovacích algoritmov môže mať vážne dopady na kvalitu ukrytia dát a spolu s tým na možnosti detekcie malwaru. Zároveň použitie nekonvenčného šifrovacieho algoritmu môže byť oveľa ľahšie detekovateľné, keďže konvenčné šifrovacie algoritmy sú často prítomné aj v legitímnych programoch. Ako sa ukáže v nasledujúcich podkapitolách, vlastné kryptografické algoritmy aj v platených krypteroch a protektoroch môžu zlyhať a ukrývané dáta odhaliť.

4.1 Prístup k novej vrstve v .NET-e

Na demonštráciu slabín šifrovacích a dešifrovacích algoritmov, ktoré je možné vidieť v krypteroch a protektoroch, boli vybrané dva protektory, ktoré sú napísané v .NET frameworku. Pre analýzu teda bude primárne použitý nástroj *dnSpy*. Keďže protektory svoje časti alebo samotný payload ukrývajú, je možné rozlišovať v behu protektoru jednotlivé vrstvy, prípadne etapy. Za vrstvu (etapu) bude považovaná každá časť programu, ktorá bola dešifrovaná a jej kód bol následne spustený.

Pre prístup k novej vrstve je možné postupovať viacerými spôsobmi:

1. Po analýze dešifrovacieho algoritmu je možné staticky dešifrovať šifrované dáta pomocou vlastného programu, alebo skriptu. Dešifrované dáta je následne možné načítať do dnSpy a pokračovať v analýze nasledujúcej vrstvy.

2. Analyzovať program dynamicky v *dnSpy* a po prebehnutí dešifrovania je možné zvoliť lokálnu premennú s dešifrovanými dátami a uložiť tieto dáta na disk ako súbor. Následne je možné načítať tento súbor do *dnSpy* a pokračovať v analýze.

3. Analyzovať program dynamicky v *dnSpy*, nechať prebehnúť dešifrovanie a počkať, kým sa dešifrované dáta načítajú do pamäte ako modul. V analyzovanom programe môže byť toto načítanie zabezpečené funkciou `AppDomain.CurrentDomain.Load` alebo `Assembly.LoadModule`.

Každý z uvedených spôsobov nesie so sebou niekoľko výhod a nevýhod. Je na zvážení analytika, ktorý spôsob si vyberie, podľa situácie a potreby.

4.1.1 Prvá metóda – statická extrakcia

Výhoda prvej metódy spočíva v tom, že extrakcia druhej vrstvy môže prebehnúť staticky, bez toho, aby bol malware spustený. Táto extrakcia sa dá jednoducho automatizovať skriptom a je teda možné pracovať aj s veľkým množstvom vzoriek zároveň. Výraznou nevýhodou tejto metódy je časová náročnosť, keďže je potrebné zanalyzovať a zreplikovať dešifrovací algoritmus. Pokiaľ je však očakávané extrahovanie z väčšieho množstva vzoriek v budúcnosti, je táto technika výhodná pre zredukovanie času potrebného na budúce extrakcie až takmer na nulu.

4.1.2 Druhá metóda – dynamická extrakcia pomocou uloženia obsahu premennej

Druhá metóda je oproti prvej vhodnejšia pre analýzu jedného alebo malého počtu vzoriek, pretože každú novú vrstvu je potrebné manuálne vyextrahovať. Ďalšia nevýhoda spočíva v tom, že malware je nutné spustiť a analyzovať ho dynamicky. Na to je nutné bezpečné prostredie – sandbox, pretože môže ľahko dôjsť k spusteniu malwaru. Zároveň je vhodné rátať s tým, že sa pri analýze (najmä komplexnejších programov) mohla prehliadnuť nejaká nebezpečná časť, ktorá sa vykoná ešte pred samotným dešifrovaním.

Na druhej strane, pre tento postup nie je potrebná žiadna analýza samotného dešifrovacieho algoritmu a stačí väčšinou len rýchla prvotná analýza, ktorá identifikuje miesto v programe, kde dochádza k dešifrovaniu dát. Zároveň nie je potrebné vytvárať vlastný dešifrovací skript, teda táto metóda je jednoznačne časovo menej náročná, pokiaľ chce analytik vyextrahovať ďalšiu vrstvu a nezaoberať sa prvou.

Realizácia tejto metódy spočíva v nájdení momentu v programe, kedy sú dešifrované dáta uložené v premennej (napr. bytové pole). Premenné sú v nástroji *dnSpy* monitorované a ich obsah je možný počas behu programu monitorovať, meniť, alebo aj uložiť. Pre potreby druhej metódy stačí v *dnSpy* zvoliť správnu premennú v správnom momente a uložiť jej obsah.

4.1.3 Tretia metóda – dynamický prístup pomocou načítania modulu

Tretia metóda má rovnakú nevýhodu ako druhá metóda – malware je nutné analyzovať dynamicky a teda je nutné ho spustiť, pri čom hrozí riziko infikovania systému. Takisto ako druhá metóda aj táto je vhodná na analýzu malého počtu vzoriek, keďže k novej vrstve sa je potrebné dostať manuálne.

Realizácia tretej metódy spočíva v nájdení momentu, kedy sa zavolá funkcia, ktorá načíta nový modul do programu. Pre nájdenie modulu je potrebné si v *dnSpy* zapnúť zobrazovanie modulov v menu `Debug` → `Windows` → `Modules`. Potom, čo sa modul načíta, zobrazí sa v zozname modulov a je možné ho načítať do `Assembly Explorer`-u.

Najpodstatnejší rozdiel medzi druhou a treťou metódou je fakt, že pri použití tretej metódy je možné pokračovať v dynamickej analýze ďalej, bez ukončenia rozbehnutého programu. Pokiaľ nová vrstva pozostáva z dynamickej knižnice, ktorá nefunguje ako samostatný program, nebude ju možné s použitím druhej metódy samostatne spustiť a dynamicky analyzovať. Môže nastať prípad, kedy sa funkcie dynamickej knižnice volajú z predošlej vrstvy a zároveň sa

v rámci funkcií takejto dynamickej knižnice využívajú zdroje, ktoré obsahuje predošlá vrstva. V takom prípade sú vrstvy na sebe závislé a nemusí byť vhodné používať druhú metódu. Keďže však závislosť nie je vždy možné vylúčiť hneď, je vhodné používať tretiu metódu a po potvrdení nezávislosti si novú vrstvu uložiť.

Pokiaľ je však nová vrstva samostatná a nijakým spôsobom nezávisí na prvej vrstve, má druhá metóda výhodu, pretože si analytik môže vytvoriť kontrolný bod, od ktorého môže pokračovať v analýze ďalej. Toto je užitočné pri programoch, ktoré sú silno zaobfuskované a natoľko komplexné, že je časovo efektívne nezačínať dynamickú analýzu vždy od úplného začiatku, ale len od začiatku určitej vrstvy.

4.2 Kazy Crypter

Kazy Crypter je napísaný v jazyku C#. *Stub* je po vygenerovaní obfuskovaný – menné priestory, názvy funkcií, premenných a aj názvy položiek v zdrojoch sú premenované na nezmyselné textové reťazce pozostávajúce z *ASCII* znakov. Na rozdiel od Cassandra Crypter-u však *stub* nie je veľký, neobsahuje žiadny kód navyše, ktorý by mal spomalovať analýzu, a teda nie je potrebné cielene hľadať relevantné časti kódu.

4.2.1 Analýza dešifrovacieho procesu

V texte je analyzovaná vzorka z `attachments/samples/Kazy Crypter/`, ktorej SHA256 je:

`2da0df3f820ac03b2975c68c423b9a3d9bcc9ffc3eaa99d2fe13dc593d5bb154`

Prvý náhľad na vzorku pomocou nástroja *dnSpy* ukazuje, že program obsahuje len dve funkcie, ale až 70 položiek v zdrojoch, ktoré sa nepoužívajú ani v jednej z dvoch funkcií programu.

4.2.1.1 Prvá vrstva

Prvá vrstva obsahuje funkciu `Main` a funkciu `IgpotMLcnngpQyIE`. Druhá funkcia zabezpečuje dešifrovanie poľa, kde je kľúč textový reťazec predaný ako parameter. Táto funkcia v prvom kroku premení kľúč vo formáte textového reťazca na bytové pole, pomocou:

```
byte[] bytes = Encoding.Unicode.GetBytes(key);
```

Následne prebehne `for` cyklus, v ktorom sa jednotlivé byty kľúča `xor`-ujú (bitový `xor`) na bytové pole obsahujúce zašifrované dáta tak, že na i-tý byte šifrovaných dát sa naxoruje hodnota kľúča na pozícii i mod key_length.

Podľa dokumentácie funkcia `Encoding.Unicode.GetBytes` vráti kódovanie pre *UTF-16* v little endian-e [30]. Keďže však funkcia na vstupe dostáva *ASCII* textový reťazec, tak sa na zakódovanie každého znaku kľúča využije len sedem zo šestnástich bitov a zvyšných deväť bude vždy nulových. V prípade analyzovanej vzorky to bude vyzerať nasledovne:

```
string key = "sXVwyrJiZXHEM"
ASCII values of key = { 0x73 0x58 0x56 0x77 0x79 0x72 0x4a 0x69
                        0x5a 0x58 0x48 0x45 0x4d }
UTF-16 little endian key = { 0x73 0x00 0x58 0x00 0x56
                         0x00 0x77 0x00 0x79 0x00 0x72 0x00
                         0x4a 0x00 0x69 0x00 0x5a 0x00 0x58
                         0x00 0x48 0x00 0x45 0x00 0x4d 0x00 }
```

Táto konverzia kľúča spôsobí, že každá druhá hodnota pri dešifrovaní bude xorovaná s nulou. Keďže je operácia `xor` symetrická a pre šifrovanie musel byť použitý rovnaký kľúč ako pre dešifrovanie, tak sa každý druhý byte šifroval xorovaním s nulou. Keďže podľa definície platí, že `xor` akejkoľvek hodnoty s nulou danú hodnotu nezmení, tak každý druhý byte šifrovaných dát je nezašifrovaný.

Vo funkcii `Main` je možné vidieť, že dešifrovacia funkcia je volaná trikrát. Všetky tri volania používajú rovnaký kľúč a dešifrujú tri statické polia. Dve zo statických polí sa dešifrujú na textové reťazce „KazyLoader.Loader“ a „Start“. Tretie statické pole obsahuje šifrovanú dynamickú knižnicu `KazyLoader.dll`. Po dešifrovaní sa z knižnice zavolá funkcia `Start` z menného priestoru s názvom `KazyLoader` a triedy `Loader`.

4.2.1.2 Druhá vrstva

Druhá vrstva obsahuje štyri funkcie, pričom ako prvá bude vykonávaná funkcia `Start`, ktorá volá ostatné tri funkcie. Na začiatku funkcie `Start` sa načítajú zdroje. Jedna zo sedemdesiatich položiek s názvom `IJfXF` je následne interpretovaná ako zoznam textových reťazcov. Tento zoznam obsahuje všetky názvy zvyšných položiek v zdrojoch v určitom poradí. Podľa tohto poradia sú následne vo funkcii `JoinBytes` obsahy všetkých zdrojov poskladané do jedného bytového poľa.

Pole obsahujúce poskladané obsahy je v ďalšom kroku interpretované ako bitmapový obrázok. Funkcia `GetBytesFromImage` následne vyextrahuje pixely z obrázku a vloží ich do bytového poľa, aby sa nepracovalo s hlavičkou a inými metadátami. Posledným dešifrovacím úkonom pred spustením ďalšej vrstvy je vloženie tohto poľa spolu s textovým reťazcom do funkcie s názvom `Encrypt`, ktorá má rovnakú funkcionalitu ako dešifrovacia funkcia v prvej vrstve.

Z druhej vrstvy je možné vidieť, že payload bol pri vytváraní chráneného súboru najskôr zašifrovaný, následne bol interpretovaný ako bitmapový obrázok, rozsekaný na veľký počet častí a poradie jednotlivých častí bolo permutované. V tejto fáze analýzy je k dispozícii dostatočné množstvo informácií na vytvorenie detekcie. Tretia vrstva, ktorá je posledná vrstva pred spustením finálneho payloadu, obsahuje najmä ochrany pred analýzou, a teda analýza vzorky pokračuje ďalej v podkapitole 6.1, ktorá je tematicky relevantná.

4.2.2 Vytvorenie detekcie

Analýza vzorky a nájdenie nedokonalostí v dešifrovacom procese pomohla vytvoriť statickú detekciu v nástroji YARA. Detekcia je založená na fakte, že každý druhý byte dynamickej knižnice uloženej v statickom poli nie je šifrovaný. Táto knižnica je jadro protektoru, o ktorom si autor myslel, že bude ukryté, pretože sa túto knižnicu nesnažil autor ďalej chrániť alebo obfuskovať. Preto je očakávané, že je táto knižnica použitá v rovnakej forme vo viacerých vzorkách.

Alternatívou by bolo zamerať sa na prvú vrstvu. Problémom je, že prvá vrstva je obfuskovaná a nie je vhodné z nej vyberať textové reťazce, pretože tieto reťazce budú unikátne pre danú vzorku. Alternatívne by bolo možné detekovať charakteristické sekvencie inštrukcií, napríklad dešifrovaciu funkciu, avšak cyklus, v ktorom sa robí jednoduchá operácia `xor`, nemusí byť unikátny. Pokiaľ sa k tejto potenciálne neunikátnej sekvencii pridá fakt, že operačné kódy MSIL inštrukcií sú vo väčšine jednobytové, vzniká reálne riziko falošne pozitívnych detekcií.

Prihliadnuc na uvedené fakty, je vhodné vybrať niekoľko sekvencií z dešifrovanej dynamickej knižnice, vhodne zvoliť zástupné znaky a zostaviť YARA pravidlo z nich. Je potrebné mať na mysli, že pri náhrade každého druhého znaku za zástupný znak môže dôjsť k spomaleniu skenovania veľkých súborov, preto boli do YARA pravidla zahrnuté ďalšie podmienky, ktoré obmedzujú stavový priestor a podmienka so zástupnými znakmi je presunutá na koniec, aby sa vyhodnocovala ako posledná. Na obmedzenie potenciálnych falošne pozitívnych detekcií bolo aj v tomto pravidle použité privátne pravidlo, ktoré obmedzí možné detekcie ľubovoľných súborov len na .NET programy. Vytvorené YARA pravidlo je možné nájsť v súbore s názvom `KazyCrypter.yar`.

Vytvorená detekcia bola otestovaná na analyzovanej vzorke a následne na troch ďalších nezávislých vzorkách, ktoré obsahujú rovnaký protektor, ale rôzne payloady. Tieto testovacie vzorky je možné nájsť v rámci príloh v priečinku `attachments/samples/Kazy Crypter/test/`. Detekcia funguje na všetkých testovaných vzorkách.

V druhej fáze bola detekcia testovaná na vzorke, ktorá chráni neškodný program `HelloWorld`. Vzorku je možné nájsť v prílohách v rámci priečinku s názvom `attachments/samples/Kazy Crypter/test/legitimate/`. Keďže detekcia reaguje na protektor samotný, tak podľa očakávania je aj táto vzorka vytvoreným pravidlom detekovaná. Detekcia legitímnych programov chránených protektorom môže spôsobovať problémy pre legitímne protektory, ktoré sú používané na ochranu legitímneho softwaru. Kazy Crypter však nie je predávaný cez vierohodné stránky a pri vyhľadávaní tohto protektoru cez webové vyhľadávače je možné vidieť odkazy smerujúce primárne na rôzne hackerské fóra. Na základe uvedených faktov je možné očakávať, že tento protektor bude používaný na ochranu malwaru. Ako pri každom novom detekčnom pravidle, je určite odporúčané pravidlo dočasne monitorovať. Na základe monitorovania bude možné potvrdiť predpoklady, že protektor je používaný výhradne na ochranu malwaru a pravidlá začať plnohodnotne využívať na detekovanie aj neznámeho malwaru zabaleného v tomto protektore.

4.3 Morpheus Crypter

Morpheus Crypter je napísaný v jazyku C# a svoje *stub*-y obfuskuje. Pre analýzu bola použitá vzorka z `attachments/samples/Morpheus Crypter/`, ktorej SHA256 je:

`d51afe5ec0766f40b36b4050b79ab2b6d645f75ffae045305cfdafb403d8475c`

Autor pri obfuskácii stavil na zlú rozoznateľnosť jednotlivých čínskych znakov analytikom, ktorý čínštinu neovláda. Pre odstránenie obfuskácií bol použitý nástroj *de4dot*, ktorý síce nerozoznal žiadny známy obfuskátor, ale napriek tomu zmenil nerozoznateľné názvy a výrazne sprehľadnil funkciu, ako je možné vidieť na obrázkoch 4.1 a 4.2. Pokračovanie analýzy prebiehalo na deobfuskovanej vzorke.

4.3.1 Prvá vrstva – prvotná analýza

Rýchla prvá analýza prvej vrstvy ukazuje, že len dve funkcie sú podstatné. Na začiatku sa vo funkcii `Main` načíta do bytového poľa obsah jediného zdroja a ten sa predá ako argument do funkcie `smethod_0`. V rámci funkcie `smethod_0` sa bytové pole s resource xoruje s nejakou (zatiaľ bližšie neurčovanou) hodnotou, následne sa načíta ako assembly a zavolá sa entry point.

4.3.2 Prvá vrstva – analýza do hĺbky

Ako je vidieť v predošlej sekcii, na pochopenie a prekonanie prvej vrstvy stačila veľmi zbežná a rýchla analýza. Takáto analýza je podstatná pre šetrenie času. Nie je v silách analytika analyzovať vždy všetko, a preto je potrebné

```
public static byte[] 曦磯纛羅矇箕礑糲纆礑礒纆篭褚礞豔笏鞣浞磋艄曦窗(byte[] 矇靨磺掩磳纗穪磔碱艭礑碰碱笺礵艳磔艭糲荏礐磈, string 矇䒾艷浞纆䐁鶌荏糲鶌矌碰纆礤纐碧䒾曦艙䒾磺艷窜磁矓)
{
    byte[] bytes = Encoding.ASCII.GetBytes(矇䒾艷浞纆䐁鶌荏糲鶌矌碰纆礤纐碧䒾曦艙䒾磺艷窜磁矓);
    for (int i = 0; i < 矇靨磺掩磳纗穪磔碱艭礑碰碱笺礵艳磔艭糲荏礐磈.Length; i++)
    {
        int num = i;
        矇靨磺掩磳纗穪磔碱艭礑碰碱笺礵艳磔艭糲荏礐磈[num] ^= (byte)(bytes[i % 矇䒾艷浞纆䐁鶌荏糲鶌矌碰纆礤纐碧䒾曦艙䒾磺艷窜磁矓.Length] >> i + 5 + bytes.Length & 255);
    }
    AppDomain.CurrentDomain.Load(矇靨磺掩磳纗穪磔碱艭礑碰碱笺礵艳磔艭糲荏礐磈).EntryPoint.Invoke(null,
    new object[]
    {
        曦纗箕碧礤礅矌礒磵艷磺穜磁礤碰礑磁癠禮籓纛磵磵.曦艳礎曦矇礒䴈羅羅礤礒鞣磺禮禮粔笏磺糲礎矇荏礞鶌礦
    });
    Guid guid = Guid.NewGuid();
    string[] array = new string[]
    {
        guid.ToString("B"),
        guid.ToString("b"),
        guid.ToString("B")
    };
    foreach (string text in array)
    {
    }
    return 矇靨磺掩磳纗穪磔碱艭礑碰碱笺礵艳磔艭糲荏礐磈;
}
```

Obr. 4.1: Morpheus Crypter – pred deobfuskáciou

```
public static byte[] smethod_0(byte[] byte_0, string string_1)
{
    byte[] bytes = Encoding.ASCII.GetBytes(string_1);
    for (int i = 0; i < byte_0.Length; i++)
    {
        int num = i;
        byte_0[num] ^= (byte)(bytes[i % string_1.Length] >> i + 5 + bytes.Length & 255);
    }
    AppDomain.CurrentDomain.Load(byte_0).EntryPoint.Invoke(null, new object[]
    {
        Class0.string_0
    });
    Guid guid = Guid.NewGuid();
    string[] array = new string[]
    {
        guid.ToString("B"),
        guid.ToString("b"),
        guid.ToString("B")
    };
    string[] array2 = array;
    for (int j = 0; j < array2.Length; j++)
    {
    }
    return byte_0;
}
```

Obr. 4.2: Morpheus Crypter – po deobfuskácii

vedieť, aký je cieľ. Pokiaľ je cieľom prekonať ochrany protektoru, dostať sa k malwaru, ktorý ukrýva a analyzovať ho, prvotná analýza je postačujúca pre ďalší postup v práci. Keďže však je momentálne cieľom analyzovať protektor samotný, je vhodné sa podrobnejšie pozrieť, ako samotný dešifrovací algoritmus funguje.

Funkcia `smethod_0` (zobrazená na obrázku 4.2) má dva parametre – bytové pole `byte_0` a textový reťazec `string_1`. O premennej `byte_0` už je z prvotnej analýzy známe, že obsahuje šifrované dáta. Textový reťazec `string_1` je transformovaný pomocou funkcie `Encoding.ASCII.GetBytes` na bytové pole `bytes`. Hodnoty z tohto bytového poľa sú v dešifrovacej slučke xorované so šifrovanými dátami, teda reťazec `string_1` slúži ako kľúč.

Samotný dešifrovací cyklus prechádza byte po byte pole `byte_0`. Pre znázornenie priority operátorov a lepšie pochopenie dešifrovacieho procesu sa i-tá hodnota poľa `byte_0` dešifruje nasledovne, pričom pre zlepšenie čitateľnosti a zachovanie štandardného pomenovávania je premenná `bytes` pomenovaná ako `key`, dĺžka premennej `bytes` ako `key_len` a premenná `byte_0` je pomenovaná ako `ct`:

```
ct[i] = ct[i] ^ ((key[i % key_len] >> (i + 5 + key_len)) & 0xFF)
```

Prvá vec, ktorú je možné si všimnúť, je, že bitový posun doprava bude o hodnotu minimálne šesť, pokiaľ by kľúč pozostával len z jedného bytu a s rastúcim iterátorom i bude hodnota bitového posunu len rásť. Toto by mohlo naznačovať, že v najlepšom prípade, kedy bude kľúč dlhý len jeden byte, bude len pri prvých dvoch iteráciách bitový posun menší ako osem, teda maximálne prvé dva byty budú xorované s niečím iným ako nula. Tento prípad sa však nedeje. Podľa dokumentácie bitového shiftu sa v tomto prípade bitový posun určuje podľa najnižších piatich bitov pravého operandu [31]. Z toho vyplýva, že napriek tomu, že hodnota `i + 5 + key_len` bude každou iteráciou rásť o jedna, tak sa každých 32 iterácií budú najnižšie byty tohto výrazu opakovať. Toto spôsobí, že bitové posuny doprava budú postupne o $0, 1, 2, ..., 32, 0, 1, ...$ bitov, pričom prvá hodnota závisí od dĺžky kľúča.

Ďalej si je možné uvedomiť, že hodnota kľúča, ktorá je bitovo posúvaná, je osembitová hodnota, teda po posune doprava väčšom ako osem bude výsledok vždy nula. Toto zapríčiní, že maximálne osem po sebe idúcich bytov `ct` môže byť xorovaných s niečím iným ako nula, teda len osem po sebe idúcich bytov z každého bloku 32 bytov je šifrovaných.

Na zavŕšenie analýzy sa ostáva pozrieť na samotný kľúč. Autor chcel očividne byť konzistentný vo svojich obfuskáciách a ako kľúč vygeneroval reťazec pozostávajúci zo sedemdesiatjeden čínskych znakov. Na konverziu textu na bytové pole bola však použitá funkcia `Encoding.ASCII.GetBytes` a čínske znaky

sa v *ASCII* tabuľke nenachádzajú. Podľa dokumentácie [32] sa všetky znaky, ktoré sa nenachádzajú v rozmedzí U+0000 až U+007F skonvertujú na znak „?“. Tento znak má *ASCII* hodnotu 0x3F, alebo bitovo 0b0011_1111. Táto konverzia má niekoľko vážnych dôsledkov:

1. Všetky byty kľúča majú rovnakú hodnotu 0x3F.
2. Hodnota, ktorá je bitovo posúvaná doprava, má vždy dolných šesť bitov s hodnotou 1 a horné dva bity sú nulové. Preto sa pri posunoch doprava väčších ako šesť celá hodnota kľúča vynuluje a teda len šesť po sebe idúcich bytov z každého bloku 32 bytov je šifrovaných.
3. Bloky šifrovaných bytov sú šifrované pomocou operácie xor s dopredu známymi hodnotami 0b00111111, 0b00011111, 0b00001111, 0b00000111, 0b00000011, 0b00000001, a to presne v poradí, v akom sú uvedené.

Vďaka tejto analýze je možné pochopiť, ako vyzerajú šifrované dáta a ako ich dešifrovať. Ďalej sa ukáže v podkapitole 4.3.4, že takáto chyba vie napomôcť k vytvoreniu detekcie.

4.3.3 Prvotná analýza druhej vrstvy

Pre prístup k druhej vrstve bola použitá dynamická analýza a načítanie modulu. Druhá vrstva obsahuje osem funkcií, pričom sa na začiatku spustí funkcia `Main`. Je možné vidieť, že vrstva má položku v zdrojoch, ktorej názov je obfuskovaný pomocou čínskych znakov. Názvy funkcií a premenných už obfuskované nie sú, čo naznačuje, že sa autor spoliehal na ochránenie prvou vrstvou.

Názvy funkcií naznačujú, že program obsahuje niekoľko obranných techník. Podrobná analýza týchto funkcií a techník prebehne v podkapitole 6.2, ktorá je tejto problematike venovaná. Analýza je rozdelená na dve časti aj z dôvodu, že sa ukáže, že na detekovanie Morpheus Crypter-u na základe slabín v šifrovacom algoritme stačí analýza ukázaná v tejto časti textu.

Posledná neznáma funkcia má názov `BB3` a po jej otvorení je možné vidieť, že sa jedná o dešifrovaciu funkciu, kde dešifrovanie funguje úplne rovnako ako v prvej vrstve. Po kliknutí pravým tlačidlom v *dnSpy* na názov tejto funkcie je možné zvoliť možnosť `analyze` a vidieť, že táto funkcia je zavolaná len jedenkrát z funkcie `Main`. Po náhľade do funkcie `Main` je možné vidieť, že ako kľúč bol znova použitý reťazec čínskych znakov, teda znova nebude väčšina dát zašifrovaných.

4.3.4 Detekcia

Aj v tomto prípade pomohla analýza a nájdenie chýb v dešifrovacom procese vytvoriť statickú detekciu v nástroji YARA. Pre vytvorenie detekcie boli

vybrané textové reťazce nachádzajúce sa v druhej vrstve. Podobne ako v podkapitole 4.2, boli uprednostnené reťazce nachádzajúce sa v druhej vrstve, pred reťazcami a bytovými sekvenciami nachádzajúcimi sa v prvej vrstve. Aj pri tomto protektore je možné vidieť, že druhá vrstva už nie je obfuskovaná a je možné očakávať zmeny v tejto vrstve s oveľa menšou frekvenciou ako v prvej vrstve, ktorá bude vďaka obfuskáciám obsahovať unikátne textové reťazce pre každú vzorku. Alternatívou by ešte mohlo byť detekovanie dešifrovacieho cyklu, ktorý však podobne ako pri Kazy Crypter-i nie je veľmi zložitý a v kombinácii s krátkymi operačnými kódmi MSIL inštrukcií, by mohlo hroziť riziko detekcie neškodných súborov a programov.

Do vytvorenej detekcie boli zakomponované tri textové reťazce z druhej vrstvy. Väčší počet reťazcov bol zvolený kvôli zamedzeniu nesprávnych detekcií, keďže je možné, že jeden rovnaký reťazec sa bude v inom programe nachádzať, ale nie všetky. Vybrané boli dva názvy funkcií a jeden textový reťazec. V prípade textového reťazca je treba mať na mysli, že keďže ide o .NET program, text bude uložený vo formáte *UTF-16*, teda za každým písmenom je treba počítať s nulovým znakom. Na bližšiu konkretizáciu detekcie bola aj v tomto detekčnom YARA pravidle pridaná podmienka, ktorá obmedzuje detekciu len na .NET programy, podobne ako v podkapitole 4.2 alebo 3.3. Táto podmienka je aj tu prítomná, aby znížila riziko možných falošne pozitívnych detekcií.

Vytvárať manuálne všetky možnosti, ako mohol byť vybraný textový reťazec zašifrovaný, je časovo náročné a náchylné na zanesenie chýb. Preto bol vytvorený skript v jazyku Python, ktorý zo zadaného textového reťazca v premennej `word` vygeneruje všetky možnosti ako môže vyzerať šifrovaná verzia tohto textového reťazca. Pre uľahčenie práce skript pri výpise aj formátuje vytvorené výsledky tak, aby ich bolo možné priamo skopírovať do YARA pravidla. Súbor s pravidlom je nazvaný `MorpheusCrypter.yar` a spomínaný Python skript je možné nájsť v adresári `attachments/other_code/` pod názvom `morpheus_gen_yara.py`.

Detekcia bola otestovaná na analyzovanej vzorke a následne na troch ďalších nezávislých vzorkách, ktoré obsahujú rovnaký protektor, ale rôzne payloady. Vzorky je možné nájsť v `apendix/samples/Morpheus Crypter/test/`. Detekcia funguje na všetkých testovaných vzorkách.

Následne bola detekcia testovaná aj na vzorke, ktorá obsahuje ako payload neškodný `HelloWorld` program. Túto vzorku je možné nájsť v prílohách v priečinku `attachments/Morpheus Crypter/test/legitimate/`. Keďže je detekcia vytvorená tak, aby reagovala na protektor samotný, tak podľa očakávania je aj táto vzorka úspešne YARA pravidlom detekovaná. Toto správanie je v poriadku, pokiaľ má analytik, ktorý pravidlo vytvára, na mysli jeho vlastnosti a zvolí vhodné použitie. Podobne ako pri Kazy Crypter-i je možné vo

webovom vyhľadávači vidieť, že je tento protektor často spomínaný na webových lokalitách, ktoré sa zaoberajú malwarom. Aj pri tomto protektore je teda očakávané, že bude možné vzorky označiť ako škodlivé len na základe použitého protektoru, avšak podobne ako pri ostatných detekčných pravidlách je najskôr potrebné monitorovanie na potvrdenie správnosti dohadov. Pokiaľ by sa ukázalo, že protektor zvykne chrániť aj legitímne programy, pravidlo je stále možné použiť v rámci ďalších, komplexnejších pravidiel, alebo pre klasifikáciu.

Kapitola 5

Techniky spustenia payloadu

Okrem dôkladného ukrytia payloadu je úlohou každého krypteru a protektoru payload úspešne spustiť. Vo všeobecnosti je možné odlišovať dva hlavné typy techník:

1. Techniky, ktoré využívajú pre spustenie payloadu miesto na disku.

2. Techniky, ktoré využívajú výlučne pamäť a na disk neukladajú nič.

Prvý typ techník spočíva v tom, že krypter, resp. packer vytvorí počas dešifrovacieho procesu súbor na disku. Z hľadiska packerov je toto legitímny a mnohokrát aj želaný efekt, keďže užívateľovi na konci ostane rozbalený pôvodný súbor. Pokiaľ sa však jedná o kryptery alebo protektory, v momente, kedy sa chránený kód alebo dáta ocitnú v originálnej podobe na disku, prestanú byť chránené a v prípade malwaru budú s veľkou pravdepodobnosťou staticky detekované.

Keďže z hľadiska ochrany pred malwarom nemá veľký význam zaoberať sa technikami prvého typu, sú pre ďalší text podstatné práve techniky spustenia payloadu, pri ktorých sa všetko odohráva len v pamäti. Tieto techniky sú náročnejšie na detekciu, pretože s pamäťou programu sa počas behu neustále manipuluje a skenovať celý pamäťový priestor programu na prítomnosť malwaru po každej operácii zápisu by bola natoľko náročná operácia, že by obmedzovala užívateľa v bežnej práci. Preto je nutné zvoliť momenty skenovania pamäte rozumne. Podrobná znalosť jednotlivých techník spustenia payloadu vie nielen pomôcť identifikovať momenty pre cielený sken pamäte a zvýšiť tak šancu identifikovať malware, ale aj zároveň vytvoriť detekčné pravidlá, ktoré budú vedieť rozpoznať vzorky využívajúce tieto techniky a označiť ich minimálne ako podozrivé a vhodné na podrobnejšie analyzovanie.

Z hľadiska klasifikácie podľa MITRE ATT&CK frameworku je možné techniky rozoberané v tejto časti práce zaradiť do kategórie s ID `T1055` – Process Injection.

5.1 Process Hollowing

Jedna z vídaných techník na spustenie payloadu spočíva v injektovaní škodlivého kódu do cudzieho, nevinne vyzerajúceho procesu. Technika sa v rámci MITRE ATT&CK frameworku označuje s ID `T1055.012`. Pre túto techniku bol vytvorený aj *proof of concept* program, ktorého funkcionalita je popisovaná v nasledujúcom texte.

5.1.1 Popis realizácie techniky

V prvom rade potrebuje krypter alebo protektor proces, ktorý bude slúžiť ako obeť. Tento proces môže byť niektorý z už bežiacich procesov v systéme, alebo si protektor spustí svoju inštanciu nejakého bežného procesu, ktorý nie je ničím podozrivý, ako napríklad `notepad.exe`.

Program `demo.py` slúžiaci ako proof of concept program, ktorý je možné nájsť v prílohách v `attachments/samples/process_hollowing/` používa pre ukážku dva ďalšie programy. Jedným je program s názvom `HelloWorld.exe`, ktorý obsahuje len kód vykresľujúci okno s názvom a textom „Hello World!“. Druhý program je `ResourceHacker.exe`, freeware program slúžiaci na prehliadanie zdrojov Windows aplikácií, ktorý je možné získať z webu [33]. Vytvorený ukážkový program najskôr spustí `HelloWorld.exe` ako program, ktorý bude obeťou a následne sa bude snažiť druhý program injektovať do spusteného procesu prvého programu. Realizácia prebieha v nasledujúcich krokoch:

1. Proces, ktorý je obeťou, sa musí dostať do `Suspended` stavu. Keďže tento proces je vytvorený v rámci ukážky, je možné použiť pri vytváraní funkciu `CreateProcessA`, ktorá pri nastavení parametru `dwCreationFlags` vytvorí proces v `Suspended` stave.

2. V priebehu tejto techniky bude potrebné využiť funkcie z knižnice `ntdll`, konkrétne sa jedná o funkciu `NtQueryInformationProcess` a o funkciu `NtUnmapViewOfSection`. Preto program pokračuje tým, že pomocou `GetProcAddress` získa adresu týchto funkcií.

3. V ďalšom kroku sa získava adresa, od ktorej je nahratý program do pamäťového priestoru – adresa `ImageBase`. V tomto momente sa využíva prvá z funkcií, ktorých adresa bola získaná v predošlom kroku – funkcia `NtQueryInformationProcess`. Funkcia vráti pre daný handle procesu štruktúru `PROCESS_BASIC_INFORMATION`. Podľa dokumentácie [34] sa v tejto štruktúre nachádza informácia o adrese PEB. S touto adresou je

možné pomocou `ReadProcessMemory` načítať PEB. Napriek tomu, že sa v oficiálnej dokumentácii táto informácia nenachádza, tak podľa [35] je na offsete 0x008 hľadaná ImageBase adresa a v kóde sa získa ako

```
void * ptrImageBase = (void*)((DWORD*)ptrPEB)[2];
```

4. Pokračovaním programu sa načíta program v roli útočníka – payload. V tomto bode by v reálnom krypteri bol payload nejakým spôsobom ukrytý a zašifrovaný, avšak pre ukážku sa jedná o načítanie nešifrovaného spustiteľného externého binárneho súboru do pamäte.

5. Akonáhle je payload v pamäti, je možné zistiť cez DOS hlavičku, kde sa nachádza štruktúra `IMAGE_NT_HEADERS`. Z tejto štruktúry je možné zistiť informácie potrebné v nasledujúcich krokoch, ako napríklad hodnota `ImageBaseAddress`, počet sekcií, adresy sekcií alebo veľkosť jednotlivých sekcií.

6. V programe, ktorý bude injektovaný, sa v tomto momente je možné zbaviť existujúceho kódu a dát – krok, kde sa reálne robí „hollowing“. Na tento účel je použitá funkcia `NtUnmapViewOfSection`.

7. Po uvoľnení priestoru je možné alokovať nové miesto pre payload pomocou funkcie `VirtualAllocEx`.

8. Pokiaľ je adresa začiatku alokovanej pamäte z predošlého kroku iná, ako preferovaná `ImageBaseAddress`, je nutné spraviť *rebase* operáciu. Jedna zo sekcií PE súboru je sekcia `.reloc`, ktorá slúži na opravenie virtuálnych adries v programe, pokiaľ bol systémom (napríklad kvôli ASLR) nahraný na inú ako preferovanú adresu. Táto sekcia obsahuje bloky, kde každý blok má na začiatku hlavičku a zoznam relokačných záznamov. Hlavička pozostáva z dvoch bytov, kde jeden z nich udáva veľkosť bloku a druhý udáva offset na začiatok stránky, od ktorého sa budú počítať nasledujúce záznamy. Záznam pozostáva taktiež z dvoch bytov, avšak len posledných 12 bitov určuje, kde presne sa na stránke nachádza adresa, ktorú treba opraviť. Pri *rebase* operácii je teda potrebné prejsť všetky bloky v sekcii a v rámci každého bloku prejsť všetky záznamy a na adresách, ktoré udávajú, je potrebné prepočítať adresy. Prepočítanie pozostáva z pričítania rozdielu preferovanej hodnoty `ImageBaseAddress` a adresy, na ktorú bol reálne program nahraný. [36]

9. Potom, čo je spravený *rebase* programu, alebo je aspoň skontrolované, že rebase nie je potrebný, je treba program reálne nahrať do alokovanej pamäte v cieľovom procese. Na tento krok sa použije funkcia `WriteProcessMemory`, ktorá nahrá do pamäte hlavičku a následne všetky sekcie. Pri nahrávaní sekcií je nutné dodržiavať adresy začiatkov sekcií z hlavičky, ktoré boli zistené v kroku 5.

10. Pred finálnym spustením programu je nutné spraviť poslednú úpravu, a to nastaviť hodnotu v kontexte vlákna vytvoreného procesu. V kontexte je nutné nastaviť hodnotu `EAX` na adresu entry pointu injektovaného programu pomocou funkcie `SetThreadContext`. Po nastavení tejto hodnoty je možné spustiť proces pomocou funkcie `ResumeThread`.[37]

5.1.2 Detekcia techniky

Túto techniku je možné monitorovať a detekovať v Cuckoo Sandboxe, pretože technika je závislá na konkrétnych API funkciách, ktoré Cuckoo monitoruje a ktorých kombinované použitie počas behu programu môže indikovať injekciu payloadu. Vytvorené detekčné pravidlá a signatúry pre túto techniku by však mali slúžiť ako indikátor podozrivého správania a nie ako dôkaz škodlivej aktivity.

Vzorka pracuje s dvomi programami `HelloWorld.exe` a `ResourceHacker.exe`, ktoré sa nachádzajú v prílohách. Keďže je možné do Cuckoo Sandboxu pridať len jednu vzorku do jednej analýzy, tak na otestovanie vzorky v Cuckoo Sandboxe bol vytvorený skript v jazyku Python. Tento skript obsahuje všetky potrebné spustiteľné súbory, ktoré uloží na disk a spustí PoC program.

Na detekciu rozoberanej techniky bola vytvorená signatúra, ktorú je možné nájsť v súbore s názvom `process_hollowing.py`. Pre vytvorenie signatúry bolo nutné zistiť, ktoré API funkcie Cuckoo monitoruje. Jedná sa primárne o API funkcie z knižnice `ntdll`, keďže iné knižnice ako `kernel32`, alebo `user32` sa v implementácii svojich API funkcií odkážu na funkcie z knižnice `ntdll`. V konečnom dôsledku bolo v signatúre implementované monitorovanie nasledujúcich funkcií:

- `CreateProcessInternalW`
- `NtUnmapViewOfSection`
- `NtAllocateVirtualMemory`
- `NtGetContextThread`
- `NtSetContextThread`
- `NtResumeThread`

Po každom zavolaní jednej z uvedených funkcií sa názov funkcie priradil k procesu podľa hodnoty PID. Po dokončení iterovania cez API volania v signatúre prebieha kontrola, ktorý z procesov zavolal všetky z uvedených funkcií. Pokiaľ taký existuje, signatúra sa aktivuje a vráti PID všetkých procesov, ktoré použili všetky z uvedených funkcií.

Akonáhle bola vytvorená signatúra pre Cuckoo Sandbox, bolo možné vytvoriť aj behaviorálne YARA pravidlo, ktoré detekuje túto techniku. Pravidlo je možné nájsť v prílohách pod názvom `T1055_012_ProcessHollowing.yar` a report z Cuckoo Sandboxu je možné nájsť v `attachments/cuckoo_reports`.

Vytvorená signatúra aj pravidlo boli následne otestované na troch vzorkách malwaru, ktorý túto techniku používa a samozrejme aj na samotnom PoC programe. Vzorky malwaru je možné nájsť v prílohách v rámci priečinku `attachments/samples/process_hollowing/test/`. Reporty všetkých vzoriek po spustení v Cuckoo Sandboxe obsahovali vytvorenú signatúru a relevantné YARA pravidlo s dodaným reportom vzorky úspešne detekovalo. Keďže sa jedná o detekciu všeobecnej techniky, tak je možné, že detekované budú úplne neznáme rodiny malwaru, alebo aj istá podmnožina legitímnych a neškodných programov (ako je to aj v prípade vytvoreného PoC programu), ktoré túto techniku používajú. Vytvorené pravidlo je vhodné používať pre vytváranie komplexnejších detekčných pravidiel, alebo ako rýchly zdroj informácií pre analytika. Pokiaľ analytik vie, aký druh injekcie vzorka používa, môže rýchlejšie a cielene vyextrahovať finálny payload.

5.2 Self Hollowing

Druhá technika na spúšťanie payloadu, ktorá je rozoberaná v tejto práci, sa nazýva self hollowing a na rozdiel od process hollowing-u, spočíva v injektovaní kódu do svojho vlastného procesu. Túto techniku používa napríklad Onion Crypter, podľa ktorého bude technika analyzovaná. Keďže podrobná analýza Onion Crypter-u bola popísaná už v článku [7], v práci sú rozoberané len časti relevantné pre pochopenie danej techniky a následné vytvorenie detekcie.

5.2.1 Pozične nezávislý kód

Ako sa ukáže, pre úspech techniky self hollowing je nutné použiť pozične nezávislý kód. Jedná sa o strojový kód, ktorý je možné vykonať nezávisle na tom, kde sa v operačnej pamäti nachádza. Na rozdiel od pozične závislého kódu, tento kód používa výhradne relatívne skoky a odkazy na svoje dáta alebo funkcie. [38]

5.2.2 Analýza techniky

Pre analýzu a vytváranie detekcie bola použitá vzorka, ktorú je možné nájsť v priečinku `attachments/samples/Onion Crypter/` so SHA256:

`8b85a4d9df1140d25f11914ec4e429c505bd97551ede19197d2b795c44770afe`

V prvom kroku realizácie self hollowing-u sa alokuje pamäť. Toto je možné dosiahnuť napríklad funkciami `VirtualAlloc`, `HeapAlloc` alebo `GlobalAlloc`.

Pamäť je potrebné alokovať s ochranou pamäte nastavenou na hodnotu `RWX`, alebo bude potrebné v najbližších krokoch túto hodnotu nastaviť, napríklad pomocou funkcie `VirtualProtect`. Analyzovaná vzorka používa na alokovanie pamäte funkciu `HeapAlloc`, ktorej predchádzalo vytvorenie haldy pomocou funkcie `HeapCreate` a ochranou pamäte nastavenou na hodnotu `RWX`.

V nasledujúcom kroku sa dešifruje pozične nezávislý kód a vloží sa do alokovanej pamäte. Akonáhle je tento kód pripravený, je možné ho zavolať a predať mu kontrolu riadenia programu.

V treťom kroku je potrebné získať adresy niektorých API funkcií, ktoré budú potrebné pre úspešné dokončenie techniky. Tieto adresy si musí pozične nezávislý kód získať sám, keďže tieto adresy nie sú dopredu známe. Na získanie týchto adries je možné najskôr načítať adresu TIB (Thread Information Block). Táto dátová štruktúra obsahuje informácie o aktuálne bežiacom vlákne a je k nej možné pristúpiť pomocou hodnoty v registri `FS` [39]. Z tejto štruktúry je možné následne získať adresu štruktúry PEB, spomínanú už pri process hollowing-u. Táto štruktúra obsahuje na offsete $0x0C$ adresu štruktúry s názvom `PEB_LDR_DATA`, ktorá obsahuje informácie o načítaných moduloch daného procesu vo forme spojového zoznamu. [40] Informácie v jednotlivých položkách zoznamu obsahujú aj názov načítaného modulu a základnú adresu modulu, vďaka čomu je možné nájsť základnú adresu dynamickej knižnice s názvom `kernel32.dll`. V prípade analyzovanej vzorky je možné vidieť ešte medzikrok, ktorý slúži ako obfuskačná technika. Krypter spočíta *CRC32* kontrolný súčet názvu modulu a porovnáva ho s predpočítaným súčtom, aby sa vyhol ukladaniu textových reťazcov.

Akonáhle má krypter adresu knižnice `kernel32.dll`, je možné vyparsovať jej PE hlavičku a získať adresu tabuľky exportov. Z tejto tabuľky je následne možné získať adresu funkcie `GetProcAddress` a pomocou nej načítať všetky potrebné funkcie z tejto knižnice. Pokiaľ by bolo potrebné využívať aj iné knižnice, je možné získať adresu API funkcie s názvom `GetModuleHandle`, ktorá sa taktiež nachádza v `kernel32.dll` a použiť ju na získanie prístupu k ďalším dynamickým knižniciam. Je potrebné podotknúť, že vzorka Onion Crypter-u v rámci obfuskovania textových reťazcov nepoužíva API funkciu `GetProcAddress`, ale iteruje cez tabuľku exportov a porovnáva *CRC32* kontrolné súčty názvov funkcií s predpočítanými kontrolnými súčtami.

V tomto momente má analyzovaný program k dispozícii všetky prostriedky potrebné na dokončenie injekcie do samého seba. Vzorka Onion Crypteru dešifruje ešte jednu vrstvu pozične nezávislého kódu, zopakuje doterajší proces získavania adries funkcií aj v ňom, ale toto nie je relevantný krok pre porozumenie self hollowing-u, takže nebude dopodrobna rozoberaný.

Krypter môže v tejto fáze pripraviť finálny payload – teda pokiaľ bol nejakým spôsobom šifrovaný alebo komprimovaný, tak sa payload pripraví do finálnej podoby. Akonáhle je payload pripravený, je možné ho injektovať. Najskôr je potrebné pomocou funkcie `VirtualProtect` zmeniť práva pre prístup k pamäti všetkých sekcií a hlavičky procesu tak, aby bolo možné do tejto pamäte zapisovať. Keď je pamäť pripravená, tak je možné namapovať payload na miesto pôvodného programu. Toto mapovanie prebieha podobne ako pri process hollowing-u, teda je potrebné skopírovať hlavičku programu a následne všetky jeho sekcie. Na záver je potrebné nastaviť správne pamäťové práva, tak ako sú v PE hlavičke nového, injektovaného programu. Po zavolaní entry pointu sa spustí úplne nový, injektovaný program.

Name	Virtual Size	Virtual Address	Raw Size	Raw Address
Byte[8]	Dword	Dword	Dword	Dword
.text	00077EB8	00001000	00078000	00000400
.rdata	00006F4C	00079000	00007000	00078400
.data	00003364	00080000	00001800	0007F400
.rsrc	0000DB18	00084000	0000DC00	00080C00
.reloc	00003152	00092000	00003200	0008E800

Obr. 5.1: Onion Crypter – hlavičky sekcií pred prevedením self hollowing-u

Name	Virtual Size	Virtual Address	Raw Size	Raw Address
Byte[8]	Dword	Dword	Dword	Dword
.text	00004450	00001000	00004600	00000400
.rdata	00000EDA	00006000	00001000	00004A00
.data	000006CC	00007000	00000400	00005A00
.bss	000006CF	00008000	00000800	00005E00
.rsrc	0005A000	00009000	00059A00	00006600

Obr. 5.2: Onion Crypter – hlavičky sekcií po prevedení self hollowing-u

Výsledok techniky self hollowing je zachytený na obrázkoch 5.1, 5.2, 5.3 a 5.4.

Obrázky ukazujú porovnanie medzi PE hlavičkami programu pri spustení a toho istého programu po prevedení self hollowing-u a nahradení pôvodného programu úplne iným programom. Je možné si všimnúť rozdielne hodnoty entry pointu, názvy sekcií alebo aj veľkosti sekcií.

Member	Offset	Size	Value
Magic	00000118	Word	010B
MajorLinkerVersion	0000011A	Byte	09
MinorLinkerVersion	0000011B	Byte	00
SizeOfCode	0000011C	Dword	00078000
SizeOfInitializedData	00000120	Dword	00019600
SizeOfUninitializedData	00000124	Dword	00000000
AddressOfEntryPoint	00000128	Dword	000076A3
BaseOfCode	0000012C	Dword	00001000
BaseOfData	00000130	Dword	00079000
ImageBase	00000134	Dword	00400000

Obr. 5.3: Onion Crypter – PE Optional Header pred self hollowing-om

Member	Offset	Size	Value
Magic	00000118	Word	010B
MajorLinkerVersion	0000011A	Byte	09
MinorLinkerVersion	0000011B	Byte	00
SizeOfCode	0000011C	Dword	00004600
SizeOfInitializedData	00000120	Dword	00002200
SizeOfUninitializedData	00000124	Dword	00000000
AddressOfEntryPoint	00000128	Dword	000010E7
BaseOfCode	0000012C	Dword	00001000
BaseOfData	00000130	Dword	00006000
ImageBase	00000134	Dword	00400000

Obr. 5.4: Onion Crypter – PE Optional Header po self hollowing-u

5.2.3 Detekcia techniky

Prítomnosť tejto techniky je možné detekovať v Cuckoo Sandboxe, keďže program implementujúci túto techniku určite potrebuje alokovať pamäť, kam zapíše a následne spustí kód. Taktiež potrebuje nastaviť ochranu pamäte tak, aby do nej mohol zapisovať a aby mohla injekcia prebehnúť. Pre detekciu tejto techniky bola vytvorená Cuckoo signatúra s názvom *self_hollowing*, ktorú je možné nájsť v priečinku `attachments/cuckoo_signatures`. Táto signatúra monitoruje nasledujúce dve API funkcie:

- `NtAllocateVirtualMemory`
- `NtProtectVirtualMemory`

Obe API funkcie sú priraďované k jednotlivým procesom, ktoré boli spustené počas analýzy. Zároveň sú monitorované argumenty, ktoré tieto funkcie dostali. V prípade API funkcie `NtAllocateVirtualMemory` sú zaznamenávané volania, ktoré dostali handle na svoj proces a konštantu na ochranu pamäte nastavenú na hodnotu $0x40$, čo zodpovedá konštante s názvom `PAGE_EXECUTE_READWRITE`. Druhá monitorovaná funkcia s názvom `NtProtectVirtualMemory` bola zaznamenávaná tiež, keď dostala handle na svoj proces a mala nastaviť ochranu pamäte na `PAGE_EXECUTE_READWRITE`. Pokiaľ ľubovoľný proces splnil tieto dve podmienky, tak je vyhodnotený ako podozrivý kvôli pravdepodobnému použitiu techniky self hollowing.

Vytvorená signatúra, a teda aj YARA pravidlo monitoruje dve často používané API funkcie a parametre, s akými sú tieto funkcie volané. Je možné, že kombináciu týchto API funkcií aj s parametrami, ktoré nastavujú ochranu častí pamäte na `RWX`, budú používať aj programy, ktoré neboli písané podľa konceptu najnižších možných oprávnení. Preto je potrebné vnímať detekciu tejto techniky ako návod a nápovedu pre analytika, aby vedel rýchlejšie zhodnotiť, čo program robí a či je škodlivý. Druhé možné využitie je ako ďalší príznak v komplexnejšom YARA pravidle, ktorý slúži na obmedzenie možných falošne pozitívnych detekcií.

Technika self hollowing spadá pod MITRE ATT&CK kategóriu s názvom Process Injection a s ID `T1055`. Táto technika na rozdiel od techniky *process hollowing* nemá vlastnú podkategóriu. Vytvorené YARA pravidlo odráža štruktúru MITRE ATT&CK matice a má názov `t1055_process_injection`. Pravidlo je možné nájsť v priečinku `attachments/yara_rules/`.

Vytvorená signatúra a aj YARA pravidlo boli otestované na troch vzorkách obsahujúcich malware, ktoré používajú techniku self hollowing. Testované vzorky je možné nájsť v priečinku `attachments/Onion Crypter/test/`. Všetky testované vzorky obsahovali v reporte z Cuckoo Sandboxu vytvorenú signatúru.

Zároveň aj YARA pravidlo s daným reportom úspešne detekovalo prítomnosť tejto techniky vo vzorkách. Podobne ako pri technike process hollowing, vytvorená signatúra a pravidlo budú detekovať aj neznámy malware, alebo legitímny software, ktorý používa túto techniku. Pravidlá je teda aj v tomto prípade možné použiť ako zdroj informácií pre analytika, aby mohol cielene a rýchlo nájsť payload, alebo ako vlastnosť vzorky pri klasifikácii.

Kapitola **6**

Techniky obrany pred analýzou

Rozvinutejšie kryptery a protektory môžu obsahovať obrany pred automatizovanou aj manuálnou analýzou. Techniky týkajúce sa takejto obrany spadajú do MITRE ATT&CK kategórie s názvom *Virtualization/Sandbox Evasion* s ID `T1497`. Ako je možné vidieť napríklad na obrázku 1.3, pri chránení aplikácie protektorom si je možné navoliť, či má chránený program obsahovať aj obrany pred analýzou. Možnosť je voliteľná, pretože na jednej strane tieto techniky môžu úspešne ochrániť vzorku pred analýzou v sandboxe, alebo spomaliť analytika, ale na druhej strane pri detekcii prítomnosti takejto techniky pôsobí vzorka podozrivejšie a môže byť klasifikovaná ako malware. V nasledujúcom texte sú analyzované vybrané protektory, ktoré takéto techniky implementujú. Na základe vykonanej analýzy sú na konci kapitoly popísané vytvorené detekcie.

6.1 Kazy Crypter

Začiatok analýzy Kazy Crypter-u bol prevedený už v predošlej kapitole 4.2, kde boli analyzované prvé dve vrstvy. Táto kapitola nadviaže na analýzu v tretej vrstve tohto protektoru. Pre zjednodušenie prístupu a uloženie si progresu predošlej analýzy bola táto vrstva dynamicky extrahovaná a uložená (vrstva nie je závislá na predošlých). K tretej vrstve sa je teda možné dostať pomocou pôvodnej vzorky, alebo je ju možné rovno otvoriť v *dnSpy* zo súboru v priečinku `attachments/samples/Kazy Crypter/` so SHA256:

`78aae0b93ea6233805e185262624dbaae090269cde8e4b6c18c362eaa194f831`

Keďže pre prístup do tejto vrstvy je potrebné prekonať dve predošlé vrstvy, autor do tejto vrstvy neimplementoval žiadne obfuskácie a ochrany kódu. Po otvorení funkcie `Main`, na ktorú ukazuje `entry_point`, je možné vidieť, že ako prvé sa volajú funkcie, ktoré majú chrániť pred analýzou (pokiaľ boli zvolené adekvátne nastavenia pri chránení programu).

6.1.1 Obrany protektoru

Prvá z funkcií slúžiacich na obranu pred analýzou sa volá `DetectSandboxie`. Kontrola spočíva v zavolaní funkcie `GetModuleHandle` z dynamickej knižnice `kernel32.dll` s parametrom `SbieDll.dll`. Funkcia `GetModuleHandle` vráti v prípade úspechu handle na zadaný modul a nulu v prípade neúspechu. Pokiaľ funkcia úspešne vráti nenulovú hodnotu, program vyhodnotí, že je emulovaný v sandboxe Sandboxie a ukončí sa.

Druhá z funkcií slúžiacich na obranu pred analýzou sa volá `DetectWireshark`. V tejto funkcii sa zavolá systémová funkcia `Process.GetProcesses`, ktorá vráti pole s objektami `Process`, pre každý proces bežiaci na počítači. Následne program iteruje cez obdržané pole a porovnáva hodnotu v atribúte s názvom `MainWindowTitle` objektu `Process` s hodnotou „The Wireshark Network Analyzer". Pokiaľ je teda na počítači spustený program, ktorý má uvedený text v názve okna, program detekuje prítomnosť programu *Wireshark*, ktorý slúži na monitorovanie sieťovej komunikácie a ukončí sa.

Tretia z funkcií slúžiacich na obranu pred analýzou sa volá `DetectWPE` a funguje rovnako ako predošlá, ale kontroluje prítomnosť programu s názvom okna „WPE PRO". Touto cestou sa program snaží zistiť, či je na počítači spustený program *Winsock Packet Editor Pro* slúžiaci na modifikáciu dát na TCP vrstve. Rovnako ako v predošlej funkcii, program sa pri detekcii prítomnosti programu *Winsock Packet Editor Pro* ukončí.

Posledná z funkcií slúžiacich na obranu pred analýzou je `DetectEmulation`. Funkcia si zistí hodnotu `TickCount`, ktorá udáva počet milisekúnd od štartu systému a uspí program. Po skončení spánku program znova zistí hodnotu `TickCount` a zisťuje, či rozdiel zistených hodnôt je väčší alebo rovný dobe spánku. Jedna z možných obrán pred automatickou analýzou môže zahŕňať dlhé uspatie programu (pokojne aj na niekoľko dní), aby vypršal limit pri automatickej analýze vzorky. Aby sa tomuto sandboxy bránili (vrátane Cuckoo Sandboxu), implementovali funkcionalitu, ktorá vie preskočiť dobu spánku a nechať program pokračovať [41]. Pokiaľ by sa však pri tomto preskočení spánku nezmenil aj čas v systéme vrátane hodnoty `TickCount`, tak program zistí, že spánok bol preskočený a ukončí sa.

6.1.2 Dokončenie analýzy

Protektor má ďalej ešte niekoľko ďalších funkcionalít. Protektor podporuje zabalenie viacerých súborov naraz, ktoré všetky postupne spustí. Protektor je možné nakonfigurovať tak, aby chránená vzorka stiahla z predom nakonfigurovanej internetovej lokácie súbor(y) a tie spustila. Ďalšia funkcionalita protektoru spočíva v možnosti nakonfigurovania spustenia programu pri štarte

systému. Poslednou zmienenou funkcionalitou je možné nastavenie atribútov súboru na skrytý a systémový. Takéto súbory bežný užívateľ v prehliadači súborov nevidí, pokiaľ si to špecificky nenastaví. Keďže tieto funkcionality nie sú relevantné pre témy jednotlivých kapitol a vytváranie detekcií, nie sú rozoberané ďalej do hĺbky. Na konci funkcie `Main` sa zavolá funkcia `Run`, ktorá spustí finálny payload.

6.2 Morpheus Crypter

Počiatočná analýza Morpheus Crypter-u bola vykonaná v kapitole 4.3, kde sa analyzovala prvá vrstva a prebehla aj prvotná analýza druhej vrstvy. Nasledujúca sekcia nadviaže na spomínanú kapitolu so zameraním sa na obranné techniky prítomné v protektore. Pre zjednodušenie prístupu k druhej vrstve bola táto vrstva exportovaná z programu a je možné ju nájsť v adresári `attachments/samples/Morpheus Crypter/` ako súbor so SHA256:

`1d25d532cb7a1e48af1612b9cfb796235096e9ffa47c35347ec0d8afc95b2f9d`

6.2.1 Obrany protektoru

Druhá vrstva začína vo funkcii `Main`, kde je možné vidieť hneď v úvode niekoľko obranných funkcií, ktoré vedia spôsobiť ukončenie programu. Tri z týchto obranných funkcií s názvami `DWireshark`, `DEmulation` a `DSandboxie` majú rovnakú funkcionalitu ako takmer rovnomenné funkcie Kazy Crypter-u, ktoré boli analyzované v predošlej sekcii.

Štvrtá z obranných funkcií zisťuje prítomnosť debuggeru, pomocou volania API funkcie `IsDebuggerPresent` z knižnice `kernel32.dll`.

Posledná z obranných funkcií sa volá `DVM`, čo je podľa funkcionality pravdepodobne akronym pre „Detect Virtual Machine“. Funkcia vytvorí inštanciu triedy `ManagementObjectSearcher`, ktorá umožňuje spraviť WMI dotaz v jazyku WQL. Dotaz je do menného priestoru `root/cimv2`, čo je priestor, ktorý obsahuje triedy týkajúce sa hardwaru počítača a jeho konfigurácie. Program dotazom zisťuje informácie o výrobcovi matičnej dosky. Pokiaľ program zistí, že výrobca matičnej dosky je „microsoft corporation“ (porovnávanie prebieha s normalizáciou textu na malé písmená), tak sa program ukončí. Podľa článku [42] je možné zistiť, že táto obrana je cielená na virtuálne stroje bežiace na `Hyper-V`. Takéto virtuálne stroje majú nastavenú hodnotu výrobcu matičnej dosky na „Microsoft Corporation“.

6.2.2 Dokončenie analýzy

V ďalších krokoch si protektor načíta zdroj, ktorý rozdelí na viacero častí pomocou oddeľujúceho textového reťazca `MORPHEOUS`. Druhú položku z výsledku delenia dešifruje a načíta ako dynamickú knižnicu s názvom `RunNet.dll`. Táto knižnica obsahuje viacero funkcionalít vo funkciách, ktoré sú podľa konfigurácie zvolenej pri vytváraní vzorky volané z aktuálne analyzovanej funkcie `Main`.

Medzi funkcionality, ktoré obsahuje knižnica `RunNet.dll`, patrí:

- Ukončovanie neželaných procesov – kontrola iteruje cez názvy spustených procesov, ktoré, ak obsahujú zadaný reťazec, tak sú ukončené.
- Stiahnutie súborov z internetu a ich následné spustenie. Tieto súbory musia byť dopredu nakonfigurované užívateľom protektoru.
- Spustenie vlákna na perzistenciu procesu. Vlákno kontroluje spustené procesy a pokiaľ nenájde hľadaný proces, ktorý má kontrolovať, tak ho spustí znovu.

Keďže tieto funkcionality nebudú slúžiť na vytvorenie detekcie a rámcovo zasahujú mimo témy kapitoly, nebudú ďalej rozoberané do väčšej hĺbky.

6.3 Aegis Crypter

Na rozdiel od predošlých dvoch protektorov, Aegis Crypter nebol v predošlých častiach práce analyzovaný. Jedná sa o protektor, ktorý je napísaný v C++, teda na analýzu bol použitý nástroj IDA, v ktorom sa program disasembloval. Napriek tomu, že vzorka obsahuje viacero funkcionalít, analýza je kvôli relevantnosti zameraná na časť, kde sa protektor snaží detekovať virtuálne prostredie. Analýza prebiehala na vzorke, ktorú je možné nájsť v priečinku `attachments/samples/Aegis Crypter/` ako súbor so SHA256:

`24be4f115ff0132ba56f2aab8e76e76782afae13e5a0b2ce25897c334dfbd509`

Vzorka je zabalená v UPX packeri, čo je možné vidieť už pri náhľade na názvy sekcií v nástroji *CFF Explorer*. Prvým krokom pre prístup k podstatným častiam protektoru je prekonanie UPX packeru. Návodov, ktoré popisujú prekonanie tejto ochrany, bolo vytvorených už mnoho, je možné použiť napr. [43].

Po extrahovaní zabaleného programu je možné nájsť v programe funkciu obsahujúcu všetky kľúčové funkcionality protektoru. Pre rýchle vyhľadanie je možné použiť vyhľadávanie v nástroji IDA a hľadať hexadecimálnu sekvenciu bytov `55 8B EC B8 D8 D6 00 00 E8`, ktorá je na začiatku hľadanej funkcie.

Funkcia začína veľkým množstvom mov inštrukcií, ktoré vkladajú do lokálnych premenných preddefinované sekvencie bytov, ktoré v zásobníku vytvoria textové reťazce. Jedná sa o obfuskačnú techniku, ktorej cieľom je ukryť v programe textové reťazce. Následne funkcia načítava adresy vybraných funkcií z dynamických knižníc pomocou kombinovaného použitia funkcií LoadLibraryA, GetModuleHandleA a GetProcAddress. Ide o ďalšiu obfuskačnú techniku, ktorá ukryje volania API funkcií, keďže sa budú volať pomocou adries uložených v lokálnych premenných.

V rámci tejto funkcie sa nachádza miesto, kde sa začnú presúvať pomocou mov inštrukcií textové reťazce VBoxService.exe a vmtoolsd.exe. Program VBoxService je možné vidieť spustený vo virtuálnom stroji VirtualBox od spoločnosti Oracle. Druhý textový reťazec je program, ktorý beží na virtuálnych strojoch VMware od rovnomennej spoločnosti. Už pri výskyte takýchto textových reťazcov môže analytik spozornieť a očakávať implementovanú obrannú techniku.

V nasledujúcom kroku sa zavolá funkcia CreateToolhelp32Snapshot s hodnotou prvého parametru 0x00000002, čím sa do vytvoreného snapshotu zahrnú všetky procesy. Snapshot je následne využitý v cykle, kde sa pomocou dvoch API funkcií Process32First a Process32Next iteruje cez všetky procesy zachytené v snapshote. Funkcie vrátia štruktúru ProcessEntry32Structure.[44] V rámci tejto štruktúry sa nachádza pole s názvom szExeFile obsahujúce názov spustiteľného súboru procesu. [45] V cykle, ktorý je zachytený aj na obrázku 6.1, je možné vidieť, že sa pomocou funkcie strstr zisťuje prítomnosť procesov vmtoolsd.exe a VBoxService.exe. Pokiaľ sú také procesy nájdené, program vie, že je spustený vo virtuálnom prostredí a ukončí sa.

6.4 Detekcia analyzovaných techník

Všetky techniky analyzované v tejto kapitole spadajú do kategórie s ID T1497 MITRE ATT&CK frameworku. Na základe predošlej analýzy vzoriek boli vytvorené signatúry pre Cuckoo Sandbox, ktoré boli následne použité na vytvorenie YARA pravidiel reflektujúcich MITRE ATT&CK maticu.

6.4.1 Kontrola prítomnosti dynamických knižníc

Pre detekciu tejto kontroly bola vytvorená Cuckoo signatúra, ktorá kontroluje všetky volania funkcie LdrGetDllHandle. Táto funkcia má ako jeden z argumentov názov modulu, ktorý signatúra porovnáva so zoznamom známych dynamických knižníc pre jednotlivé virtuálne stroje. Tento zoznam je implementovaný v premennej sandbox_dlls, kde sa aktuálne kontrolujú tri dynamické knižnice pre virtuálne prostredia Sandboxie, VMware a VirtualBox. Signatúra je jednoducho rozširovateľná pridávaním položiek do zoznamu, takže

Obr. 6.1: Aegis Crypter – obrana pred virtualizovaným prostredím

po analyzovaní nových vzoriek malwaru, ktoré sa dotazujú na zatiaľ nezaradené dynamické knižnice, je možné tieto knižnice pridať do signatúry a rozšíriť jej funkcionalitu.

Vytvorená Cuckoo signatúra je zaradená do MITRE ATT&CK podkategórie Systémové kontroly s ID `1497.001`. V YARA pravidlách je signatúra použitá na vytvorenie privátneho pravidla, ktoré je možné nájsť v prílohách v súbore pod názvom tejto podkategórie. Následne je toto privátne pravidlo použité

v YARA pravidle detekujúcom celú MITRE ATT&CK podkategóriu s ID `1497.001`.

6.4.2 Kontrola prítomnosti debuggeru

Pre detekciu kontroly debuggeru bola vytvorená Cuckoo signatúra kontrolujúca použitie API funkcie `IsDebuggerPresent`. Signatúra bola následne použitá na vytvorenie privátneho YARA pravidla, ktoré bolo pridané do pravidla detekujúceho MITRE ATT&CK podkategóriu s názvom *kontrola aktivít užívateľa* a s ID `T1497.002`.

6.4.3 Kontrola spustených procesov

Na základe analýzy vzorky, v ktorej bola technika použitá, je možné pracovať s faktom, že kontrola spustených procesov si vyžaduje použitie minimálne nasledujúcich troch API funkcií `CreateToolhelp32Snapshot`, `Process32FirstW` a `Process32NextW` v dopredu známom poradí. Na základe týchto poznatkov nadobudnutých pri analýze bolo možné vytvoriť Cuckoo signatúru, ktorá kontroluje, či ktorýkoľvek z procesov spustených počas analýzy mal možnosť kontrolovať spustené procesy.

Signatúra bola opäť použitá na vytvorenie privátneho YARA pravidla, ktoré rozšírilo MITRE ATT&CK podkategóriu s ID `T1497.001`. V tomto momente je možné demonštrovať výhody použitia privátnych pravidiel, ktoré spočívajú v možnej jednoduchej rozšíriteľnosti. Pri budúcom rozširovaní pravidla pre celú MITRE ATT&CK podkategóriu o nové techniky, je možné jednoducho vytvoriť nové, nezávislé privátne pravidlá a tie len pridať do YARA pravidla pre danú podkategóriu.

6.4.4 Kontrola otvorených okien

Podobne ako pre predošlé techniky, aj pre túto bola vytvorená Cuckoo signatúra, ktorá detekuje techniku na základe použitia API funkcií. Signatúra bola použitá na vytvorenie privátneho pravidla, ktoré následne rozšírilo možnosti detekovania techník kategórie s ID `T1497.001`, do ktorej spadajú kontroly aktivity užívateľa.

6.4.5 Kontrola virtualizácie cez WMI dotazy

Detekovanie techniky, ktorá kontroluje prítomnosť virtuálneho prostredia pomocou WMI dotazov, nebolo možné implementovať priamočiaro, pretože sandbox Cuckoo priamo dané dotazy nevidí. V behaviorálnom reporte je možné vidieť, že vzorka si načítala dynamickú knižnicu s názvom `wminet_utils.dll` a následne z nej zisťovala adresy viacerých funkcií. Použitie týchto funkcií však

v Cuckoo reporte nie je zachytené, keďže tieto funkcie Cuckoo Sandbox nemonitoruje.

Pre indikáciu použitia WMI dotazov bola vytvorená Cuckoo signatúra, ktorá monitoruje, či vzorka načítava dynamickú knižnicu `wminet_utils.dll`. Táto signatúra bola následne použitá v privátnom YARA pravidle, ktoré rozširuje MITRE ATT&CK podkategóriu s ID `T1497.001` zameranú na systémové kontroly.

6.4.6 Detekcia preskakovania spánku

Po spustení vzoriek Kazy Crypter-u a Morpheus Crypter-u v Cuckoo Sandboxe bolo na základe reportu z analýzy zistené, že Cuckoo Sandbox nemonitoruje API funkciu `nativeGetTickCount`. Keďže je táto funkcia použitá na detekovanie preskakovania spánku sandboxom, nebolo možné vytvoriť signatúru presne popisujúcu správanie vzorky počas použitia tejto techniky. Namiesto toho bola vytvorená všeobecnejšia signatúra, ktorá monitoruje volania funkcie `sleep`. Táto signatúra detekuje aj techniky, ktoré sa snažia spomaliť analýzu uspatím programu natoľko, že sa škodlivý kód vôbec nespustí.

Napriek tomu, že vytvorená signatúra nie je dostatočne špecifická pre pôvodne analyzovanú techniku, dokáže ju detekovať. Táto signatúra bola zaradená do MITRE ATT&CK podkategórie s ID `1497.003`, ktorá je venovaná obranám pred sandboxom závislým na čase. Do tejto kategórie spadá aj technika spomaľovania analýzy a snaha o dosiahnutie timeoutu sandboxu, takže aj keď je signatúra všeobecnejšie zameraná, funguje na detekciu techník v tejto kategórii.

6.4.7 Zhodnotenie detekcií

Všetky Cuckoo signatúry, ktoré boli popísané v predchádzajúcich sekciách, je možné nájsť v priečinku `attachments/cuckoo_signatures/` a vytvorené YARA pravidlá sa nachádzajú v priečinku `attachments/yara_rules/` pomenované podľa názvu a ID jednotlivých MITRE ATT&CK kategórií, ktoré pokrývajú. Vytvorené YARA pravidlá, ktoré reflektujú MITRE ATT&CK maticu, sú jednoducho rozšíriteľné o ďalšie funkcionality v prípade ďalších analýz.

YARA pravidlá aj Cuckoo signatúry boli otestované na analyzovaných vzorkách, ktoré je možné nájsť v `attachments/samples/signature_test/`. Reporty z analýzy v Cuckoo Sandboxe je možné nájsť v prílohách v priečinku `attachments/cuckoo_reports/`. Jednotlivé Cuckoo reporty sú pomenované podľa hashe súboru, ktorý bol analyzovaný. Vytvorené signatúry úspešne reagujú na techniky použité v jednotlivých vzorkách a poskytujú nielen informácie, ktoré sú spracované YARA pravidlami, ale ako je vidieť na obrázku

6.2, tak už priamo v rozhraní Cuckoo Sandboxu je možné pozorovať podozrivé správanie sa vzorky, ktoré je popísané signatúrami.

Vytvorené signatúry a YARA pravidlá boli testované aj na rôznych vzorkách, ktoré je možné nájsť v priečinku `attachments/samples/signature_test/` obsahujúcich malware aj legitímny software. V priložených reportoch z analýz v Cuckoo Sandboxe je vidieť, že signatúry úspešne reagujú aj na vzorky malwaru a aj na vzorky s neškodnými programami. Napriek tomu, že všetky detekcie vytvorené v tejto kapitole slúžia na zisťovanie prítomnosti techník, ktoré využíva malware, tak mnohé techniky môžu byť použité aj legitímnym softwarom. Príkladom môže byť už len signatúra, ktorá detekuje preskakovanie spánku a timeoutovanie sandboxu. Legitímne programy sa môžu uspať z rôznych dôvodov, ako napríklad čakanie na udalosť spustenú užívateľom alebo iným programom. Preto je potrebné vytvorené detekcie vnímať ako pomôcky na rýchle porozumenie rôznych funkcionalít analyzovanej vzorky alebo ako upozornenie na potenciálne podozrivé techniky. Ďalšia možnosť využitia je v klasifikátoroch, alebo iných YARA pravidlách, ktoré sa snažia čo najšpecifickejšie popísať správanie malwaru za účelom minimalizovania falošne pozitívnych detekcií.

Napriek tomu, že nebolo do signatúry, ktorá detekuje preskakovanie spánku sandboxom, možné použiť API funkciu na priamu detekciu tejto techniky, bola získaná informácia o nemonitorovanej API funkcii. Pokiaľ sú nedostatky sandboxov známe, je možné sandbox ďalej zlepšovať a nedostatky postupne odstraňovať. V tomto prípade by bola možná implementácia monitorovania novej API funkcie, keďže je na základe analýzy potreba tohto monitorovania opodstatnená. Naopak horším prípadom by bolo, pokiaľ sú nedostatky monitorovaného sandboxu neznáme a malware má šancu ukryť svoje funkcionality alebo spôsobiť škodu, preto aj nájdenie nedostatku sandboxu je považované za priaznivý výsledok analýzy.

Sample delays execution, probably to evade automatic analysis. (2 events)

Execution delay	t = 1.0 [s]	Time delay in seconds
Execution delay	t = 0.5 [s]	Time delay in seconds

Sample probably does WMI queries using a WMI related DLL (1 event)

Time & API	Arguments	Status	Return	Repeated
LdrLoadDll Nov. 23, 2021, 8:06 p.m.	module_name: `C:\Windows\Microsoft.NET\Framework\v2.0.50727\\wminet_utils.dll` basename: `wminet_utils` stack_pivoted: `0` flags: `0` module_address: `0x6a310000`	1	0	0

One or more potentially interesting buffers were extracted, these generally contain injected code, configuration data, etc.

Sample may check titles of opened windows to detect analytical tools. (1 event)

Possible window title check	PID = 3032	Process with this PID probably checked for opened analytical tools

Sample tries to detect the presence of a debugger. (1 event)

Debugger detection	IsDebuggerPresent	Using this API function the sample tries to detect the presence of a debugger.

Sample loads known DLLs of various VMs, probably to detect if it's ran inside VM. (1 event)

Virtual environment detection	SbieDll.dll	Process tried to load a library specific for a VM, probably to detect if it's ran inside VM

Injects code into a process and executes it (1 event)

Process Hollowing	PID = 3032	Process with this PID injected code into another process and executed it

Obr. 6.2: Signatúry spustené vzorkou Morpheus Crypteru

Záver

Cieľom práce bolo preskúmanie, analyzovanie a popísanie techník, ktoré používajú kryptery a protektory za účelom ochrany malwaru. Zároveň bolo cieľom práce vytvoriť ochranu pred analyzovanými obfuskátormi a technikami, ktoré tieto obfuskátory používajú.

Práca splnila svoj hlavný cieľ a aj všetky vedľajšie ciele. V prvej časti práce boli predstavené kryptery, packery a protektory, o ktorých práca pojednávala. V rámci budovania teoretických podkladov potrebných vo zvyšku práce bola predstavená problematika falošnej pozitivity, na ktorú je potrebné myslieť pri vytváraní detekcií malwaru.

Práca pokračovala predstavením nástroja Cuckoo Sandbox, ktorý umožňuje automatizovať analýzu malwaru a vie poskytnúť analytikovi cenné informácie, ktoré vedia urýchliť nasledujúcu analýzu vzoriek. Ďalej práca predstavila nástroj YARA, ktorý umožňuje popisovanie statických aj behaviorálnych vlastností malwaru a vytváranie detekčných pravidiel. Tento nástroj bol úspešne rozšírený, čím sa zvýšila možnosť spolupráce nástrojov Cuckoo Sandbox a YARA. Zlepšenie spolupráce spočíva v možnosti využívať signatúry vytvorené pre Cuckoo Sandbox v YARA pravidlách. Týmto sa nástroju YARA výrazne rozšírili možnosti využívať k vytváraniu detekcií aj informácie z dynamickej analýzy.

V ďalšom texte prebehla analýza obfuskačných techník. Najskôr sa v texte pojednávalo o steganografii, pričom pre ukážku a analýzu tejto techniky v praxi bol vybraný Cassandra Crypter. Následne sa práca zaoberala nástrojom ConfuresEx 2, ktorý poskytuje viacero funkcionalít slúžiacich na obfuskáciu. Jednotlivé techniky a funkcionality boli postupne analyzované a popísané v texte. Výstupom z analýz sú YARA pravidlá, ktoré slúžia na detekciu možnej prítomnosti steganografie a prítomnosti nástroja ConfuserEx 2.

Práca sa ďalej zamerala na vlastné kryptografické funkcie implementované v malwari. Na ukážku boli analyzované dva protektory, ktoré obsahujú nedokonalý kryptografický algoritmus. V texte je popísaná ukážková analýza týchto nástrojov a ich dešifrovacích algoritmov. Na základe tejto ukážkovej analýzy bolo následne možné vytvoriť YARA pravidlá detekujúce tieto kryptery, bez ohľadu na to, aký malware sa v nich bude ukrývať.

Nasledujúci text práce pojednáva o možnostiach spustenia payloadu na infikovanom zariadení. V práci boli vybrané dve reprezentatívne techniky, pričom na jednu z nich bol aj vytvorený ukážkový PoC program a druhá technika bola analyzovaná na reálnej vzorke protektoru obsahujúcej malware. Obe techniky boli popísané Cuckoo signatúrou a následne YARA pravidlom tak, aby odrážali štruktúru MITRE ATT&CK matice.

Posledná časť práce bola venovaná technikám, ktoré malware používa na obranu pred manuálnou aj automatizovanou analýzou. V tejto časti práce boli použité tri protektory, ktoré boli analyzované so zameraním sa na takéto techniky. Na základe analýzy spomínaných krypterov a protektorov boli vytvorené Cuckoo signatúry, ktoré boli úspešne využité pre vytvorenie YARA pravidiel.

Práca otvára viacero možností a námetov na budúci výskum a prácu. Prvou možnosťou je rozširovať ďalej implementáciu detekcie jednotlivých techník, ktoré sa nachádzajú v MITRE ATT&CK matici. Takýmto spôsobom sa kategorizácia techník efektívne premietne priamo na jednotlivé súbory, o ktorých je potrebné rozhodnúť, či sú škodlivé alebo nie. Informácie z kategorizácie techník je ďalej možné využiť aj ako príznaky pre prípadnú klasifikáciu súborov, alebo len rozšírenie už existujúcich klasifikátorov.

Ďalšie možné rozšírenie práce vyplýva z faktu, kedy bolo v práci zistené, že niektoré techniky, ktoré malware používa, nie je možné efektívne detekovať v Cuckoo Sandboxe, pretože tento sandbox nemonitoruje všetko, čo sa v systéme deje. Rozšírenie by spočívalo v prieskume techník, ktoré Cuckoo Sandbox nevidí a rozšírenie tohto open-source sandboxu o monitorovanie nových API funkcií. Rozširovanie tohto nástroja pomôže ďalej automatizovať analýzu malwaru, čo je pri dnešnom počte potenciálne škodlivých súborov, ktoré sa šíria cez internet, nepopierateľne potrebné.

Literatúra

[1] Szappanos, G.: MyKings: The Slow But Steady Growth of a Relentless Botnet. [online], december 2019, [cit. 7. septembra 2021]. Dostupné z: https://subscription.packtpub.com/book/application-development/9781788620604/1/ch01lvl1sec13/3-malware-components

[2] Mohanta, A.: Preventing Ransomware - 3. Malware components. [online], marec 2018, [cit. 7. septembra 2021]. Dostupné z: https://subscription.packtpub.com/book/application-development/9781788620604/1/ch01lvl1sec13/3-malware-components

[3] Budd, C.: Avast researchers identify OnionCrypter, a key malware component since 2016. [online], marec 2021, [cit. 7. septembra 2021]. Dostupné z: https://blog.avast.com/onioncrypter-threat-research-avast

[4] Arntz, P.: Explained: Packer, Crypter, and Protector. [online], marec 2017, [cit. 7. septembra 2021]. Dostupné z: https://blog.malwarebytes.com/cybercrime/malware/2017/03/explained-packer-crypter-and-protector/

[5] Zahradnický, T.; Kokeš, J.; Jirkal, M.: Disassembling a obfuskace. [online], september 2021, [cit. 10. novembra 2021]. Dostupné z: https://courses.fit.cvut.cz/NI-REV/media/lectures/rev04cz.pdf

[6] aegiscrypt: Aegis Crypter. [online], [cit. 29. novembra 2021]. Dostupné z: https://www.facebook.com/aegiscrypt/

[7] Kaloč, J.: Hidden menace: Peeling back the secrets of OnionCrypter. [online], marec 2021, [cit. 7. júna 2021]. Dostupné z: https://decoded.avast.io/jakubkaloc/onion-crypter/

[8] VirusTotal: The pattern matching swiss knife for malware researchers (and everyone else). [online], [cit. 14. decembra 2021]. Dostupné z: https://virustotal.github.io/yara/

[9] McAfee: What Is the MITRE ATT&CK Framework? [online], [cit. 10. júna 2021]. Dostupné z: https://www.mcafee.com/enterprise/en-us/security-awareness/cybersecurity/what-is-mitre-attack-framework.html

[10] Strom, B. E.; Applebaum, A.; Miller, D.; aj.: MITRE ATT&CK®: Design and Philosophy. [online], júl 2018, [cit. 10. júna 2021]. Dostupné z: https://attack.mitre.org/docs/ATTACK_Design_and_Philosophy_March_2020.pdf

[11] MITRE: Enterprise Matrix. [online], [cit. 27. novembra 2021]. Dostupné z: https://attack.mitre.org/matrices/enterprise/

[12] Microsoft: PROCESSENTRY32 structure (tlhelp32.h). [online], august 2016, [cit. 27. novembra 2021]. Dostupné z: https://www.blackhat.com/docs/us-16/materials/us-16-Bulazel-AVLeak-Fingerprinting-Antivirus-Emulators-For-Advanced-Malware-Evasion.pdf

[13] Project, T. H.: Cuckoo Sandbox Book. [online], [cit. 10. júna 2021]. Dostupné z: https://cuckoo.sh/docs/index.html

[14] cuckoosandbox: Community Repository. [online], jún 2020, [cit. 10. júna 2021]. Dostupné z: https://github.com/cuckoosandbox/community

[15] Project, T. H.: cuckoosandbox repository. [online], [cit. 8. novembra 2021]. Dostupné z: https://github.com/cuckoosandbox/cuckoo/blob/073c5aab0ff1e4065f665045472309fc4064e354/cuckoo/common/abstracts.py

[16] Alvarez, V. M.: Tweet. [online], september 2016, [cit. 9. júna 2021]. Dostupné z: https://twitter.com/plusvic/status/778983467627479040

[17] VirusTotal: Writing YARA rules. [online], [cit. 9. júna 2021]. Dostupné z: https://yara.readthedocs.io/en/stable/writingrules.html

[18] VirusTotal: Modules. [online], [cit. 10. júna 2021]. Dostupné z: https://yara.readthedocs.io/en/stable/modules.html

[19] VirusTotal: Writing your own modules. [online], [cit. 10. júna 2021]. Dostupné z: https://yara.readthedocs.io/en/stable/writingmodules.html

[20] VIRUSTOTAL: What's VT Hunting? [online], [cit. 28. novembra 2021]. Dostupné z: https://support.virustotal.com/hc/en-us/articles/360000363717-What-s-VT-Hunting-

[21] Wikipedia: Steganography. [online], november 2021, [cit. 27. novembra 2021]. Dostupné z: https://en.wikipedia.org/wiki/Steganography

[22] Sanchez, W. G.; Chen, J. C.: Capesand Exploit Kit's Tools Seen in KurdishCoder Campaign. [online], December 2019, [cit. 10. septembra 2021]. Dostupné z: https://www.trendmicro.com/en_us/research/19/l/obfuscation-tools-found-in-the-capesand-exploit-kit-possibly-used-in-kurdishcoder-campaign.html

[23] Proofpoint Threat Research Team: Commodity .NET Packers use Embedded Images to Hide Payloads. [online], December 2020, [cit. 10. septembra 2021]. Dostupné z: https://www.proofpoint.com/us/blog/threat-insight/commodity-net-packers-use-embedded-images-hide-payloads

[24] ntinfo: Entropy and the distinctive signs of packed PE files. [online], jún 2014, [cit. 10. novembra 2021]. Dostupné z: http://n10info.blogspot.com/2014/06/entropy-and-distinctive-signs-of-packed.html

[25] PreEmptive Solutions: Reverse Engineering. [online], [cit. 17. septembra 2021]. Dostupné z: https://www.preemptive.com/dotfuscator/pro/userguide/en/protection_reverse_engineering.html

[26] Readme.md. [online], [cit. 17. septembra 2021]. Dostupné z: https://github.com/de4dot/de4dot

[27] mkaring: ConfuserEx. [online], november 2021, [cit. 27. novembra 2021]. Dostupné z: https://github.com/mkaring/ConfuserEx

[28] mkaring: ConfuserEx 2. [online], [cit. 27. novembra 2021]. Dostupné z: https://mkaring.github.io/ConfuserEx/

[29] Microsoft: Profiling Overview. [online], september 2021, [cit. 14. novembra 2021]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/framework/unmanaged-api/profiling/profiling-overview

[30] Microsoft: Encoding.Unicode Property. [online], [cit. 10. septembra 2021]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/api/system.text.encoding.unicode?view=net-5.0

[31] Microsoft: Bitwise and shift operators (C# reference). [online], [cit. 21. septembra 2021]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/bitwise-and-shift-operators

[32] Microsoft: How to use character encoding classes in .NET. [online], [cit. 21. septembra 2021]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/standard/base-types/character-encoding

[33] Johnson, A.: Resource HackerTM. [online], november 2020, [cit. 7. septembra 2021]. Dostupné z: http://www.angusj.com/resourcehacker/

[34] PEB-Process-Environment-Block. [online], máj 2015, [cit. 7. septembra 2021]. Dostupné z: https://www.aldeid.com/wiki/PEB-Process-Environment-Block

[35] Microsoft: NtQueryInformationProcess function (winternl.h). [online], máj 2018, [cit. 7. septembra 2021]. Dostupné z: https://docs.microsoft.com/en-us/windows/win32/api/winternl/nf-winternl-ntqueryinformationprocess

[36] Anders: Base relocation table. [online], január 2015, [cit. 7. septembra 2021]. Dostupné z: http://research32.blogspot.com/2015/01/base-relocation-table.html

[37] m0n0ph1: Process Hollowing. [online], september 2018, [cit. 7. septembra 2021]. Dostupné z: https://github.com/m0n0ph1/Process-Hollowing

[38] Wikipedia: Position-independent code. [online], máj 2021, [cit. 27. novembra 2021]. Dostupné z: https://cs.wikipedia.org/wiki/Position-independent_code

[39] Wikipedia: Thread Information Block. [online], apríl 2021, [cit. 27. novembra 2021]. Dostupné z: https://en.wikipedia.org/wiki/Win32_Thread_Information_Block

[40] ALDEID: PEB LDR DATA. [online], máj 2015, [cit. 27. novembra 2021]. Dostupné z: https://www.aldeid.com/wiki/PEB_LDR_DATA

[41] Bremer, J.: Components. [online], [cit. 20. novembra 2021]. Dostupné z: https://cuckoo-monitor.readthedocs.io/en/latest/components.html

[42] tonysoper_MSFT: Profiling Overview. [online], máj 2010, [cit. 21. novembra 2021]. Dostupné z: https://social.technet.microsoft.com/wiki/contents/articles/942.hyper-v-how-to-detect-if-a-computer-is-a-vm-using-script.aspx

[43] Upadhyay, S.: How to unpack UPX packed malware with a SINGLE breakpoint. [online], jún 2021, [cit. 22. novembra 2021]. Dostupné z: https://infosecwriteups.com/how-to-unpack-upx-packed-malware-with-a-single-breakpoint-4d3a23e21332

[44] Microsoft: Process32First function (tlhelp32.h). [online], október 2021, [cit. 22. novembra 2021]. Dostupné z: `https://docs.microsoft.com/en-us/windows/win32/api/tlhelp32/nf-tlhelp32-process32first`

[45] Microsoft: PROCESSENTRY32 structure (tlhelp32.h). [online], máj 2021, [cit. 22. novembra 2021]. Dostupné z: `https://docs.microsoft.com/en-us/windows/win32/api/tlhelp32/ns-tlhelp32-processentry32`

Dodatok A

Zoznam použitých skratiek

API Application Programming Interface

ASCII American Standard Code for Information Interchange

ASLR Address space layout randomization

ATT&CK Adversarial Tactics, Techniques and Common Knowledge

CRC Cyclic redundancy check

FUD Fully Undetectable

ID Identificator

IT Information Technology

JSON JavaScript Object Notation

MSIL Microsoft Intermediate Language

PCRE Perl Compatible Regular Expressions

PE Portable Executable

PEB Process Environment Block

PID Process identifier

PoC Proof of Concept

POSIX The Portable Operating System Interface

RAT Remote Access Trojan

RGB Red, Green, Blue

RGBA Red, Green, Blue, Alpha

SHA Secure Hash Algorithm

TCP Transmission Control Protocol

TIB Thread Information Block

UTF-8 8-bit Unicode Transformation Format

UTF-16 16-bit Unicode Transformation Format

WMI Windows Management Instrumentation

WQL WMI Query Language

XML Extensible Markup Language

YARA Yet Another Recursive Acronym

Dodatok B

Obsah priloženého pamäťového média

```
 readme.md ..........stručný popis obsahu priloženého pamäťového média
 attachments ............................... adresár s prílohami práce
 |__ ...
 src
 |__ Diplomová práca.tex........zdrojová forma práce vo formáte LaTeX
 |__ bibliografia.bib......zdrojová forma bibliografie vo formáte LaTeX
 |__ images ....................... adresár s obrázkami použitými v práci
 text .......................................... adresár s textom práce
 |__ thesis.pdf ............................ text práce vo formáte PDF
```

- ...
- `attachments` ... adresár s prílohami práce
  - `cuckoo_reports` ... adresár s reportami z analýz v Cuckoo Sandboxe
  - `cuckoo_signatures` ... adresár s vytvorenými Cuckoo signatúrami
  - `other_code` ... adresár s vytvorenými zdrojovými kódmi programov
    - `HelloWorld_dotnet` ... testovací program napísaný v jazyku C#
    - `HelloWorld2_dotnet` ... testovací program napísaný v jazyku C#
    - `HelloWorld.cpp` ... testovací program napísaný v jazyku C++
    - `InjectionToProcess.cpp` ... zdrojový kód PoC programu techniky process hollowing
    - `morpheus_gen_yara.py` . skript na generovanie šifrovaných reťazcov Morpheus Crypter-u
  - `samples` ... adresár s analyzovanými a testovanými vzorkami
    - `Aegis Crypter` ... adresár so vzorkami Aegis Crypter-u
    - `Cassandra Crypter` ... adresár so vzorkami Cassandra Crypter-u
    - `ConfuserEx 2` ... adresár pre ConfuserEx 2
      - `protected_samples` adresár so vzorkami chránenými nástrojom ConfuserEx 2
      - `ConfuresEx-GUI.zip` ... archív s nástrojom ConfuserEx 2
    - `Kazy Crypter` ... adresár so vzorkami Kazy Crypter-u
    - `Morpheus Crypter` ... adresár so vzorkami Morpheus Crypter-u
    - `Onion Crypter` ... adresár so vzorkami Onion Crypter-u
    - `process_hollowing` ... adresár pre process hollowing
      - `test` ... adresár so vzorkami používajúcimi process hollowing
      - `demo.py` ... skript s PoC na testovanie process hollowing-u v Cuckoo Sandboxe
      - `executables.zip` ... archív so spustiteľným PoC programom a súbormi potrebnými pre PoC program
    - `signature_test` .. adresár obsahujúci vzorky na testovanie Cuckoo signatúr
  - `yara_modifications` ... adresár obsahujúci zdrojový kód pre úpravu nástroja YARA
  - `yara_rules` ... adresár s vytvorenými detekciami v nástroji YARA