

**SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE  
FAKULTA ELEKTROTECHNIKY A INFORMATIKY**

Evidenčné číslo: FEI-5382-73886

**MODERNÉ TECHNIKY NA UKRÝVANIE  
ČINNOSTI MALVÉRU  
BAKALÁRSKA PRÁCA**

**2020**

**Lukáš Gnip**

**SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE**  
**FAKULTA ELEKTROTECHNIKY A INFORMATIKY**

Evidenčné číslo: FEI-5382-73886

**MODERNÉ TECHNIKY NA UKRÝVANIE**  
**ČINNOSTI MALVÉRU**  
**BAKALÁRSKA PRÁCA**

Študijný program: Aplikovaná informatika  
Číslo študijného odboru: 2511  
Názov študijného odboru: 9.2.9 Aplikovaná informatika  
Školiace pracovisko: Ústav informatiky a matematiky  
Vedúci záverečnej práce: Ing. Peter Švec

**Bratislava 2020**

**Lukáš Gnip**



## ZADANIE BAKALÁRSKEJ PRÁCE

Študent: **Lukáš Gnip**  
ID študenta: **73886**  
Študijný program: **aplikovaná informatika**  
Študijný odbor: **informatika**  
Vedúci práce: **Ing. Peter Švec**  
Miesto vypracovania: **Ústav informatiky a matematiky**

Názov práce: **Moderné techniky na ukrývanie činnosti malvéru**

Jazyk, v ktorom sa práca vypracuje: **slovenský jazyk**

Špecifikácia zadania:

V súčasnosti väčšina malvéru používa sofistikované techniky voči detekcii antivírusovým programom a sťaženiu analýzy škodlivého kódu. Medzi takéto techniky patrí využitie rôznych foriem packerov, ukrývanie škodlivých procesov, detekcia behu vo virtuálnom prostredí a pod. Cieľom práce je analyzovať techniky, ktoré využívajú v súčasnosti najrozšírenejšie druhy malvéru a implementovať ukážkovú aplikáciu na detekciu vybranej metódy.

Úlohy:

1. Naštudujte techniky používané súčasným malvérom.
2. Implementujte aplikáciu na detekciu vybranej techniky.
3. Zhodnoťte implementáciu a porovnajte s existujúcimi riešeniami.

Zoznam odbornéj literatúry:

1. Sikorski, M. – Honig, A. *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*. San Francisco: No Starch Press, 2012. 800 s. ISBN 1-59327-290-1.
2. Scholte, T. – Egele, M. – Kruegel, C. A Survey on Automated Dynamic Malware Analysis Techniques and Tools. [online]. 2011. URL: [https://www.iseclab.org/papers/malware\\_survey.pdf](https://www.iseclab.org/papers/malware_survey.pdf).

Riešenie zadania práce od: **23. 09. 2019**

Dátum odovzdania práce: **01. 06. 2020**

**Lukáš Gnip**  
študent

# SÚHRN

SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE  
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Študijný program:	Aplikovaná informatika
Autor:	Lukáš Gnip
Bakalárska práca:	Moderné techniky na ukrývanie činnosti malvéru
Vedúci záverečnej práce:	Ing. Peter Švec
Miesto a rok predloženia práce:	Bratislava 2020

Práca sa zaoberá súčasnými technikami na ukryvanie činnosti malvéru v napadnutých systémoch. V úvodnej časti práce popisujeme malvér a jednotlivé techniky na ukrývanie, ktoré sú najčastejšie využívané v posledných rokoch. V tejto časti práce sme sa zaoberali najmä vzorkami, ktoré boli detegované počas posledného roka. Cieľom práce bolo navrhnúť detekčný algoritmus vybranej techniky. Metóda, ktorú sme sa rozhodli bližšie preskúmať je tzv. *process hollowing*. Navrhnutý algoritmus je založený na princípe sledovania zvolených Windows API volaní, pomocou ktorých sa dokáže samotný malvér ukryť do bežného procesu. Základom algoritmu je konečný stavový automat, ktorý vyhodnocuje prítomnosť *process hollowing* techniky sledovaním postupnosti typických API volaní, používaných na implementáciu danej metódy. V ďalšej časti popisujeme samotný algoritmus, jednotlivé moduly detekčnej aplikácie a spôsob, akým sme ich implementovali. V záverečnej časti uvádzame výsledky, ktoré sme dosiahli v rámci experimentov so skutočnými vzorkami malvéru.

Kľúčové slová: malvér, process hollowing, detekcia, konečný stavový automat

# ABSTRACT

SLOVAK UNIVERSITY OF TECHNOLOGY IN BRATISLAVA

FACULTY OF ELECTRICAL ENGINEERING AND INFORMATION TECHNOLOGY

Study Programme:	Applied Informatics
Author:	Lukáš Gnip
Bachelor Thesis:	Modern Hiding Techniques in Malware
Supervisor:	Ing. Peter Švec
Place and year of submission:	Bratislava 2020

The bachelor thesis deals with current techniques for hiding the activity of malware in compromised systems. In the introduction of the thesis we describe the malware and individual techniques for hiding, which are most often used in recent years. In this part of the thesis, we dealt mainly with samples that were detected during the last year. The main objective of the bachelor thesis was to design a detection algorithm of a selected technique. The method that we decided to examine in more detail is the so-called *process hollowing*. The proposed algorithm is based on the principle of monitoring selected Windows API calls, which can be used to hide the malware itself in a normal process. The basis of the algorithm is a finite state machine, which evaluates the presence of the *process hollowing* technique by monitoring the sequence of typical API calls used to implement the method. In the next part we describe the algorithm itself, the individual modules of the detection application and the way in which we implemented them. In the final part of thesis, we present the results we achieved in experiments with real samples of malware.

Keywords: malware, process hollowing, detection, finite state machine

# Obsah

<b>Úvod</b>	<b>1</b>
<b>1 Malvér</b>	<b>2</b>
1.1 Techniky ukryvania činnosti . . . . .	2
1.2 Súčasný malvér . . . . .	5
<b>2 Process hollowing</b>	<b>8</b>
2.1 Princíp . . . . .	8
2.2 Využívané API funkcie . . . . .	8
<b>3 Existujúce riešenia na detekciu</b>	<b>11</b>
3.1 PHDetection . . . . .	11
3.2 HollowFind . . . . .	11
<b>4 Algoritmus na detekciu</b>	<b>12</b>
4.1 Konečný stavový automat . . . . .	14
4.2 Matica konečného stavového automatu . . . . .	15
<b>5 Implementácia</b>	<b>17</b>
5.1 Algoritmus . . . . .	18
5.2 Modul na zber dát . . . . .	20
5.3 Detours . . . . .	22
5.4 Konečný automat . . . . .	22
<b>6 Výsledky</b>	<b>23</b>
6.1 Škodlivé vzorky . . . . .	23
6.2 Vyhodnotenie . . . . .	24
<b>Záver</b>	<b>25</b>
<b>Zoznam použitej literatúry</b>	<b>26</b>
<b>Prílohy</b>	<b>I</b>
<b>A Štruktúra elektronického nosiča</b>	<b>II</b>
<b>B Technická dokumentácia</b>	<b>III</b>

## Zoznam obrázkov a tabuliek

Obrázok 1	Ukážka zmien adresného priestoru počas Hollowingu. . . . .	9
Obrázok 2	Konečný stavový automat reprezentujúci postupnosť API volaní vedúcich k ukrytiu malvéru. . . . .	13
Obrázok 3	Zobrazenie jednotlivých modulov riešenia a vzťahov medzi nimi. .	17
Obrázok 4	Stavový diagram priebehu detekcie. . . . .	18
Obrázok 5	Ukážka konfiguračného súboru pre API ResumeThread. . . . .	19
Obrázok 6	Ukážka zapisovaných API volaní. . . . .	19
Tabuľka 1	Techniky ukrývania činnosti využívané súčasným malvérom. . . . .	7
Tabuľka 2	Matica prechodov medzi stavmi pri API volaniach. . . . .	16
Tabuľka 3	Výsledky experimentov so škodlivými vzorkami. . . . .	23

## **Zoznam skratiek a značiek**

**APC** - Asynchronous Procedure Call

**API** - Application Programming Interface

**BOT** - RoBOT

**CnC** - Command and Control Center

**DLL** - Dynamic Link Library

**EWM** - Extra Windows Memory

**FTP** - File Transfer Protocol

**IAT** - Import Address Table

**IP** - Instruction Pointer

**PE** - Portable Executable

**PEB** - Process Environment Block

**VAD** - Virtual Address Descriptor



# Zoznam výpisov

1	Vytvorenie mutexu. . . . .	19
2	Definícia API volania SetThreadContext. . . . .	20
3	Implementácia funkcie writeFunctionToFile. . . . .	21
4	Nahradenie pôvodnej funkcie SetThreadContext. . . . .	21

# Úvod

Malvér môžeme charakterizovať ako škodlivý kód, ktorého primárnym cieľom je uškodiť používateľovi napr. stratou peňazí alebo súkromia, zneužitím osobných údajov a pod. Malvér na svoju infiltráciu najčastejšie využíva bezpečnostné chyby v systéme, prípadne rôzne phishingové kampane. Dôležitým prvkom každého škodlivého kódu je taktiež spôsob, akým sa môže vyhnúť odhaleniu. Jedným z takýchto spôsobov je ukrytie sa v pamäti systému a vydávanie sa za legitímny proces. Kým v minulosti vznikalo približne zopár vzoriek ročne, dnes sú to tisícky jedinečných súborov denne. Z toho dôvodu je nutné sa zaoberať detekciou a odhaľovaním škodlivého kódu v systéme aby sme vedeli zabrániť rôznym bezpečnostným incidentom.

V práci sa zaoberáme technikami, pomocou ktorých dokáže malvér ukryť svoju existenciu pred antivírusovými riešeniami a forenznými inžiniermi. Medzi najznámejšie metódy patria napr. *DLL injection*, *process hollowing* alebo *portable executable injection*. V našej práci sme sa rozhodli bližšie preskúmať techniku *process hollowing*. Princíp spomínanej metódy je podobný ako pri klasickom injektovaní škodlivého kódu do cudzieho procesu. Rozdiel spočíva najmä v tom, že *process hollowing* taktiež odmapuje pamäť pôvodného procesu. V rámci práce sme navrhli detekčný algoritmus, ktorý je založený na konečnom stavovom automate. Aplikácia sleduje v reálnom čase dôležité API volania a na ich základe sa posúva v automate. Jednotlivé stavy konečného automatu, reprezentujú fázy v ktorých sa nachádza malvér, počas injektovania škodlivého kódu do cudzieho procesu pomocou techniky *process hollowing*.

Štruktúra našej práce je nasledovná. V úvodnej kapitole popisujeme najznámejšie techniky na ukrývanie činnosti malvéru. Taktiež uvádzame vzorky malvéru, ktoré boli detegované počas minulého roka spolu s identifikáciou techniky, ktorú používajú.

V druhej kapitole sa zaoberáme detailnou charakteristikou techniky *process hollowing*. V rámci kapitoly taktiež uvádzame zoznam Windows API volaní, ktoré sa používajú na implementáciu techniky *process hollowing*.

Kapitola 3 popisuje súčasné riešenia, ktoré sa zaoberajú detekciou techniky *process hollowing*.

V ďalšej kapitole uvádzame popis nami navrhnutého algoritmu. V rámci kapitoly prezentujeme navrhnutý konečný stavový automat a tabuľku prechodov.

Kapitola 5 sa zaoberá popisom implementácie jednotlivých modulov detekčnej aplikácie, použitých knižníc a technológií.

Záverenčná kapitola sa venuje výsledkom, ktoré sme dosiahli pomocou našej detekčnej aplikácie.

# 1 Malvér

Je to softvér, ktorého cieľom je poškodiť, zablokovať, zmocniť sa alebo odcudziť citlivé informácie uložené v počítači. Cieľom malvéru je získať informácie pre útočníka a následné zneužitie týchto informácií na rôzne druhy nelegálnej činnosti za účelom danej obeti uškodiť alebo sa na nej finančne obohatiť. Malvér sa kedysi členil na rôzne kategórie ako napr. vírusy, *backdoor*, *spyware*, *ransomware* a pod.[5] V súčasnosti už toto delenie nie je veľmi aktuálne, pretože malvér v dnešnej dobe je už viacmenej kombináciou týchto kategórií a využíva rôzne komponenty z jednotlivých druhov škodlivého kódu. V nasledujúcej kapitole sa podrobnejšie zaoberáme rôznymi spôsobmi ukrývania činnosti a prítomnosti malvéru, ktoré môžu byť v súčasnosti využívané.

## 1.1 Techniky ukrývania činnosti

Malvér vo všeobecnosti patrí ku škodlivému kódu. Preto je nutné jeho bežiacie procesy utajiť. Ak chce byť malvér úspešný v získavaní údajov alebo finančných prostriedkov, musí byť jeho beh ukrytý pred potenciálnym antivírusom, prípadne forezným inžinierom ktorý je schopný ho odhaliť. Za týmto účelom sa využívajú rôzne techniky na ukrytie malvéru buď v pamäti počítača, rôznych shell skriptoch alebo dynamických knižniciach, aby boli čo najmenej detegovateľné. [18]

- **DLL Injection / Reflective DLL Injection**

*DLL Injection* je technika, pri ktorej malvér zapíše cestu ku škodlivému DLL súboru do virtuálneho adresného priestoru legitímneho procesu a následne zabezpečí aby daný proces túto dynamickú knižnicu načítal. Postup je nasledovný:

- Škodlivý kód získa prístup ku legitímnemu procesu.
- V rámci legitímneho procesu alokuje priestor dostatočne veľký na zapísanie cesty ku škodlivému DLL súboru.
- Malvér zapíše cestu ku škodlivému DLL súboru do virtuálneho adresného priestoru legitímneho procesu.
- Malvér na záver spustí vlákno v rámci legitímneho procesu, ktorého cieľom je načítať a spustiť DLL súbor.

Hlavným cieľom *DLL Injection* je škodlivý kód ukryť do legitímneho procesu, kde je malvér ukrytý pred antivírusovým softvérom, odkiaľ môže byť následne spustený [10].

- **Proces Hollowing**

Využíva podobné princípy ukrývania škodlivého softvéru ako *DLL Injection*. Cieľom je ukryť škodlivý kód do existujúceho procesu, z ktorého sa následne bude vykonávať [10]. Princíp je podobný ako pri technike *DLL Injection*:

- Malvér spustí nový pozastavený legítimný proces, do ktorého sa plánuje ukryť.
- Škodlivý kód odmapuje pamäť legítimného procesu.
- Do virtuálneho adresného priestoru legítimného procesu nakopíruje svoj škodlivý kód.
- Po dokončení kopírovania nastaví nový vstupný bod procesu a obnoví pozastavený proces.

Ako môžeme vidieť, hlavný rozdiel spočíva najmä v odmapovaní existujúceho kódu. V prípade *Process Hollowing* techniky bude v rámci neškodného procesu bežať výhradne iba škodlivý kód, na rozdiel od *DLL injection*, kde v rámci jedného procesu sa nachádza škodlivý kód spolu s pôvodným. Výsledkom môže byť napr. spustený bežný proces *svchost.exe*, ktorý ale v skutočnosti vykonáva škodlivú aktivitu.

- **Thread Execution Hijacking**

Pri tejto technike existujú určité podobnosti s metódou *Process Hollowing* a *DLL injection* [10]. Hlavný princíp spočíva v získaní prístupu už k existujúcemu vláknu legítimného procesu, do ktorého chceme vložiť škodlivý kód. Po získaní prístupu k vláknu, malvér dané vlákno pozastaví. Samotné injektovanie môže uskutočniť podobne ako pri *DLL injection*, zapísaním cesty k DLL súboru a následnom načítaní. Nevýhodou takéhoto spustenia pozastaveného programu je, že môže spôsobiť pád systému v rámci systémového volania. Výhodou ostáva, že technika nepotrebuje vytvoriť nové vlákno alebo proces a použije už existujúce.

- **Portable Executable Injection**

Výhodou tohoto spôsobu ukrytia malvéru je využitie nakopírovania celého PE súboru do existujúceho procesu [10]. Ako výhodu môžeme taktiež považovať, že malvér nepotrebuje uložiť žiadne súbory na disk ale len priamo prekopíruje dáta PE súboru do legítimného procesu. Hlavnou nevýhodou je, že sa zmení bázo-ová adresa procesu a z toho dôvodu je nutné prepočítať nové adresy v legítimnom procese tak, aby bolo zabezpečené korektné správanie novo vloženého PE súboru.

- **Hook injection**

Hookovanie je všeobecne technika používaná na zachytávanie API volaní. *Hook injection* využíva túto techniku na načítanie škodlivého DLL, pri zachytení určitej udalosti v konkrétnom vlákne [10]. Pomocou príslušného API volania vieme nainštalovať hook na špecifickú udalosť v systéme (napr. stlačenie klávesy), pre konkrétne vlákno v systéme. Taktiež vieme špecifikovať smerník na funkciu, ktorá sa má zavolať ak daná udalosť nastane. V tejto funkcii následne vieme určiť že sa má načítať napr. škodlivé DLL. Môžeme si všimnúť že väčšina techník sa samotné načítanie kódu využíva metódu načítania škodlivého DLL do regulérneho procesu a hlavné rozdiely spočívajú najmä v riešení ako donútiť cudzí proces túto funkcionálnosť vyvolať.

- **APC Injection**

Škodlivý softvér môže využívať výhody tzv. *Asynchronous Procedure Calls* (APC), aby prinútil iné vlákno spustiť svoj vlastný kód [10]. Túto funkcionálnosť vie dosiahnuť tak, že vloží dané vlákno do APC fronty. Každé vlákno má vlastnú APC frontu, z ktorej vykonáva volania ak sa nachádza v pozastavenom stave a čaká na konkrétne udalosti. Podobne ako v predchádzajúcich prípadoch môže malvér do APC fronty vložiť smerník na funkciu, v ktorej sa načíta škodlivé DLL. Malvér zvyčajne hľadá ľubovoľné vlákno ktoré sa nachádza v špeciálnom stave, pri ktorom vykonáva postupne položky v APC fronte. Takýchto vláken je väčšinou v systéme veľa.

- **Extra windows memory injection**

Tento spôsob schovávanie softvéru sa spolieha na možnosť špecifikovať dodatočnú pamäť pri registrácii aplikačných okien v systéme. Pri registrácii nového okna aplikácie, softvér špecifikuje ďalšie bajty pamäte, ktoré rozšíria veľkosť alokovanej pamäte pre spustenú aplikáciu [10], nazývané aj *Extra Windows Memory*(EWM). V tejto časti ale nevzniká dostatok miesta na uloženie dát. Aby sa toto obmedzenie obišlo, škodlivý softvér zapíše kód do zdieľanej pamäte a do EWM vloží ukazovateľ na danú časť. Do tejto rozšírenej časti zdieľanej pamäte ďalej softvér zapíše smerník na funkciu, ktorá obsahuje kód na načítanie malvéru. Malvér môže nakoniec vyvolať spustenie tohoto kódu pomocou konkrétnych Windows API volaní.

## 1.2 Súčasný malvér

Táto kapitola obsahuje opis jednotlivých vzoriek škodlivého kódu z roku 2019, ktoré boli detegované spoločnosťami ako Avast a McAfee. Tieto vzorky sú najčastejšie využívané v oblasti Európy. Kapitola obsahuje bližší opis jednotlivých vzoriek, ich využitie, použité spôsoby útokov a ukrytie malvéru v systéme.

- **Sodinokibi**

Tento malvér bol detekovaný v období okolo apríla 2019. Patrí do rodiny ransomvéru, ktorých cieľom je zašifrovať dáta v zariadení a následne za dešifrovanie pýtať peniaze [6] (väčšinou v podobe kryptomeny). Názov bol objavený v heši, ktorý obsahoval názov *Sodinokibi.exe*. Vírus sa šíri sám zneužívaním zraniteľnosti v serveroch, ktoré používajú *Oracle WebLogic*. Kód je navrhnutý tak, aby rýchlo vykonával šifrovanie špecifických súborov, definovaných v konfigurácii ransomvéru. Prvou akciou škodlivého kódu je načítať všetky externé funkcie potrebné počas behu programu. Technika využívaná na ukrytie malvéru je *Portable Executable Injection*. Analýza spoločnosti McAfee ukazuje podobnosť s iným starším malvérom GandCrab.

- **Emotet**

Emotet je malvér, ktorý sa primárne šíri pomocou rôznych spam emailov [14]. Na infikovanie zariadenia používa rôzne skripty, makrá v dokumentoch alebo linky. Emotet sa teda spolieha najmä na techniky sociálneho inžinierstva. Prezентuje sa ako hodnoverný zástupca napr. banky, rôznych internetových obchodov, a pod. Emotet sa prvýkrát objavil v roku 2014 kedy využíval na infikovanie rôzne JavaScript súbory [3]. V roku 2019 sa tento vírus objavil znova tentokrát už v pokročilejšej verzii. V novej verzii je Emotet už polymorfným škodlivým kódom, čo mu umožňuje vyhnúť sa klasickej detekcii. Emotet môže navyše generovať falošné funkcionality, ak je spustený vo virtuálnom prostredí čo zhoršuje jeho detekciu systéme.

- **Zeus**

Prvýkrát odhalený v roku 2007 sa Zeus Trojan, ktorý sa často nazýva Zbot, stal jedným z najúspešnejších botnetov na svete a postihol milióny počítačov [13]. Taktiež bolo vytvorených množstvo variantov, ktoré boli založené na tomto malvéri. Po čase sa znovu objavil v pozmenenej podobe so zameraním na odchyťávanie bankových operácií (odchyťávanie prihlasovacích údajov do internet bankingu). Dosahuje to

prostredníctvom monitorovania webových stránok a zaznamenávania klávesov. Keď malvér zistí, že sa používateľ nachádza na webovej stránke banky, začne zaznamenávať stlačenia klávesov použité na prihlásenie. Infekcia prebieha pomocou spamov. Keď užívateľ klikne na odkaz v správe alebo stiahne obsah súboru, spolu s ním stiahne a spustí aj makro, ktoré po nainštalovaní umožňuje sledovanie zariadenia.

- **Dridex**

Dridex je známy trójsky kôň, ktorý sa špecializuje na krádež kreditných údajov v online bankovníctve. Tento typ škodlivého kódu sa objavil v roku 2014 a stále sa postupne vyvíja. Nový variant Dridex je schopný vyhnúť sa detekcii tradičnými antivírusovými produktami. Tento malvér je v súčasnosti schopný detekovať približne 25 až 30 percent aktuálnych antivírusových softvérov. [16]

- **Mirai**

*Mirai* je samošíriteľný typ škodlivého súboru na vytvorenie botnetu. Zdrojový kód pre *Mirai* bol autormi verejne sprístupnený po úspešnom a dobre propagovanom útoku na webovú stránku Krebs. Kód botnetu Mirai infikuje zariadenia pripojené k internetu, ktoré využívajú telnet protokol (sieťový komunikačný protokol založený na TCP) na nájdenie tých, ktoré stále používajú svoje predvolené užívateľské meno a heslo. Účinnosť málvéru *Mirai* je spôsobená jeho schopnosťou infikovať desiatky tisíc týchto nezabezpečených zariadení a koordinovať ich tak, aby začali útok DDoS proti vybranej obeti. [12]

Mirai má dve hlavné zložky, samotný vírus a C&C server, ktorý ovláda kompromitované zariadenia (BOT) a posiela im pokyny na spustenie jedného z útokov proti jednej alebo viacerým obetiam. Proces skenera prebieha nepretržite na každom infikovanom zariadení pomocou protokolu telnet (na porte TCP 23 alebo 2323)

C&C predstavuje jednoduché rozhranie príkazového riadku, ktoré umožňuje útočníkovi určiť algoritmus, IP adresu obete a trvanie útoku. C&C tiež čaká na to, aby jej existujúce BOT-y vrátili novoobjavené adresy zariadení, ktoré používa na ďalšie rozširovanie botnetu. Algoritmy sú konfigurovateľné z C&C, ale v predvolenom nastavení má *Mirai* tendenciu náhodne rozdeľovať rôzne polia (ako sú čísla portov, poradové čísla, identifikátory atď.).

- **Osiris**

Osiris je odvodený od malvéru Kronos, ktorý sa zameriaval na bankovníctvo. Podobne ako Kronos, je Osiris modernejšou verziou bankového trójskeho koňa [15].

Táto verzia malvéru využíva na skrývanie metódu *process hollowing*. Umožňuje mu vydávať sa za legitímne procesy. Malvér sa šíri vydávaním sa za legitímny spustiteľný súbor (útoky zaznamenné s malvérom Osiris boli dokumenty Microsoft Word). Vydávanie sa za iný oficiálny softvér značne sťažuje identifikáciu malvéru a obmedzuje možnosti na zastavenie útoku [19]. Malvér v dokumentoch Word obsahoval aj makrá, ktoré po spustení stiahli ďalší škodlivý malvér, ktorý umožňuje zahľtiť zariadenia alebo sťažiť detekciu

- **Loki**

Loki je ďalšou variáciou staršieho malvéru Kronos. Rovnako ako Osiris, aj Loki využíva na svoje ukrytie metódu *process hollowing*. Loki sa zameriava na krádeže osobných údajov ako napr. prihlasovacie údaje a heslá. Od augusta 2018 až do súčasnosti sa Loki zameriava na firemné poštové schránky prostredníctvom phishingových a spamových e-mailov. Phishingové e-maily zahŕňajú prílohu súboru s príponou .iso, ktorá sťahuje a spúšťa škodlivý softvér.

Celkový prehľad použitých techník v súčasných vzorkách škodlivého kódu môžeme vidieť v tabuľke č.1.

Názov malvéru	Technika ukrývania						
	Sodinokibi	Emotet	Zeus	Dridex	Mirai	Osiris	Loki
<i>DLL Injection</i>		✗	✗		✗		
<i>Process hollowing</i>						✗	✗
<i>Thread Execution Hijacking</i>							
<i>Portable Executable Injection</i>	✗			✗			
<i>Hook injection</i>							
<i>APC Injection</i>							
<i>Extra windows memory injection</i>							

Tabuľka 1: Techniky ukrývania činnosti využívané súčasným malvérom.



## 2 Process hollowing

Zvolený spôsob ukrytia malvéru, ktorým sme sa v tejto práci zaoberali je *process hollowing*. Nasledujúca kapitola sa venuje spôsobu akým sa malvér môže ukryť pomocou spomínanej techniky. Taktiež obsahuje potenciálne API funkcie pomocou ktorých môže byť technika *process hollowing* implementovaná.

### 2.1 Princíp

Princíp ukrytia malvéru, ktorý využíva *process hollowing* je do istej miery podobný technike *DLL Injection*. Hlavný cieľ metódy spočíva v ukrytí škodlivého kódu do bežného procesu, ktorý v systéme Windows spôsobuje čo najmenšie podozrenie [17]. Takýmto procesom môže byť napríklad *svchost.exe*, ktorý je v systéme bežne spustený aj vo viacerých inštanciách. *Process hollowing* vytvorí pozastavený proces *svchost.exe* (prípadne získa prístup už k bežiacemu procesu) a odmapuje jeho pamäť. Po odmapovaní alokuje dostatok miesta vo virtuálnom adresnom priestore procesu. Následne, po alokácii nakopíruje škodlivý kód a nastaví nový vstupný bod programu. Po ukončení týchto krokov obnoví pozastavený proces. Výsledkom je navonok bežiaci štandardný proces *svchost.exe*, ktorý ale vo vnútri vykonáva škodlivú činnosť. Schematické znázornenie priebehu tejto techniky môžeme vidieť na obrázku č.1.

### 2.2 Využívané API funkcie

*Process hollowing* využíva na svoje fungovanie rôzne štandardné API volania. Nasledujúce nami vybrané API funkcie[9], môžu byť využité pri technike *process hollowing*:

- **CreateThread**

Vytvorí nové vlákno vo virtuálnom adresnom priestore procesu, ktorý danú funkciu zavolať.

- **CreateRemoteThread**

Vytvorí vlákno, ktoré beží vo virtuálnom adresovom priestore iného procesu.

- **CreateRemoteThreadEx**

Funkcia vytvára vlákno, ktoré sa spúšťa vo virtuálnom adresovom priestore iného procesu a prípadne špecifikujte rozšírené atribúty, ako napr. nastavenie na ktorom procesore bude dané vlákno bežať.



Obrázok 1: Ukážka zmien adresného priestoru počas Hollowingu.

- **ResumeThread**

Funkcia dekrementuje hodnotu, ktorá špecifikuje, koľkrát bolo vlákno pozastavané. V prípade ak sa hodnota dostane na nulu, vlákno je obnovené. V opačnom prípade ostáva pozastavené.

- **SuspendThread**

Pozastavenie vykonávanie činnosti špecifikovaného vlákna.

- **SwitchToThread**

Spôsobuje prepnutie aktuálneho vlákna, na vlákno, ktoré je pripravené bežať na procesore. Tento výber vykonáva operačný systém.

- **CreateProcessA** Vytvorí nový proces a jeho hlavné vlákno. Nový proces beží s rovnakými oprávneniami ako proces, ktorý danú funkciu zavolať.

- **VirtualAlloc**

Alokuje alebo mení oprávnenia stránok vo virtuálnom adresom priestore procesu, ktorý danú funkciu zavolať. Alokovaná pamäť je automaticky inicializovaná na nulu.

- **VirtualAllocEx**

Funguje rovnako ako *VirtualAlloc*, s tým rozdielom, že alokuje pamäť v rámci virtuálneho adresného priestoru iného procesu.

- **WriteProcessMemory**

Zapisuje údaje do pamäti v zadanom procese. Celá oblasť, do ktorej sa zapisuje, musí mať potrebné oprávnenia na zápis.

- **ReadProcessMemory**

Číta údaje z virtuálneho adresného priestoru špecifikovaného procesu.

- **SetThreadContext**

Nastavuje kontext (t.j. obsah registrov) pre špecifikované vlákno.

- **NtUnmapViewOfSection**

Odmapuje pamäť vybraného procesu.

## 3 Existujúce riešenia na detekciu

Doposiaľ známe existujúce riešenia na detekciu techniky *process hollowing* využívanej niektorými vzorkami škodlivého kódu, sú určené na forenznú analýzu. Táto analýza prebieha až po infikovaní zariadenia malvérom a zistením, že škodlivý kód sa už v zariadení nachádza. Riešenia spomínané v tejto kapitole sa teda nezameriavajú na detekciu techniky v reálnom čase.

### 3.1 PHDetection

*PHDetection* hľadá moduly, od ktorých závisí pôvodný spustiteľný program [7]. *PHDetection* kontroluje či sú dané moduly načítané do pamäte programu. Ak nástroj nájde moduly, na ktorých závisí dotýčny program (t.j. sú zapísané v IAT) ale nenájde ich v pamäti procesu, *PHDetection* deteguje že sa jedná o *process hollowing* a pôvodný proces bol nahradený iným. Existuje mnoho spustiteľných súborov, ktoré nezávisia od veľkého počtu modulov a kvôli tomu nástroj analyzuje aj tabuľku importov, v ktorej sa nachádzajú moduly, ktoré sa majú načítať až počas prvého použitia. To znamená že DLL súbor sa načíta do pamäte procesu až v momente, kedy sa zavolá prvá funkcia z tohoto modulu. *PHDetection* teda postupne prechádza všetky bežiacie procesy a analyzuje načítané moduly. V prípade ak daný modul nastavený na neskoršie načítanie a ešte sa nenachádza v pamäti, nástroj porovnáva časové značky spustiteľného súboru na disku a v pamäti. Program bol implementovaný v jazyku C++.

### 3.2 HollowFind

*Hollowfind* je plugin pre nástroj *Volatility* na detekciu rôznych typov techniky *process hollowing* používaných škodlivým kódom [4]. Plugin sa zameriava taktiež na rôzne formy obfuskácie danej techniky s cieľom sťaženia forenznej analýzy. Plugin deteguje metódu na základe porovnávania VAD a PEB tabuliek. VAD tabuľka je stromová štruktúra reprezentujúca jednotlivé stránky vo virtuálnom adresnom priestore a PEB tabuľka obsahuje rôzne informácie o konkrétnom procese. Príkladom môže byť chýbajúca cesta k spustiteľnému súboru vo VAD tabuľke alebo rôzne bázové adresy procesu v PEB a VAD tabuľke.

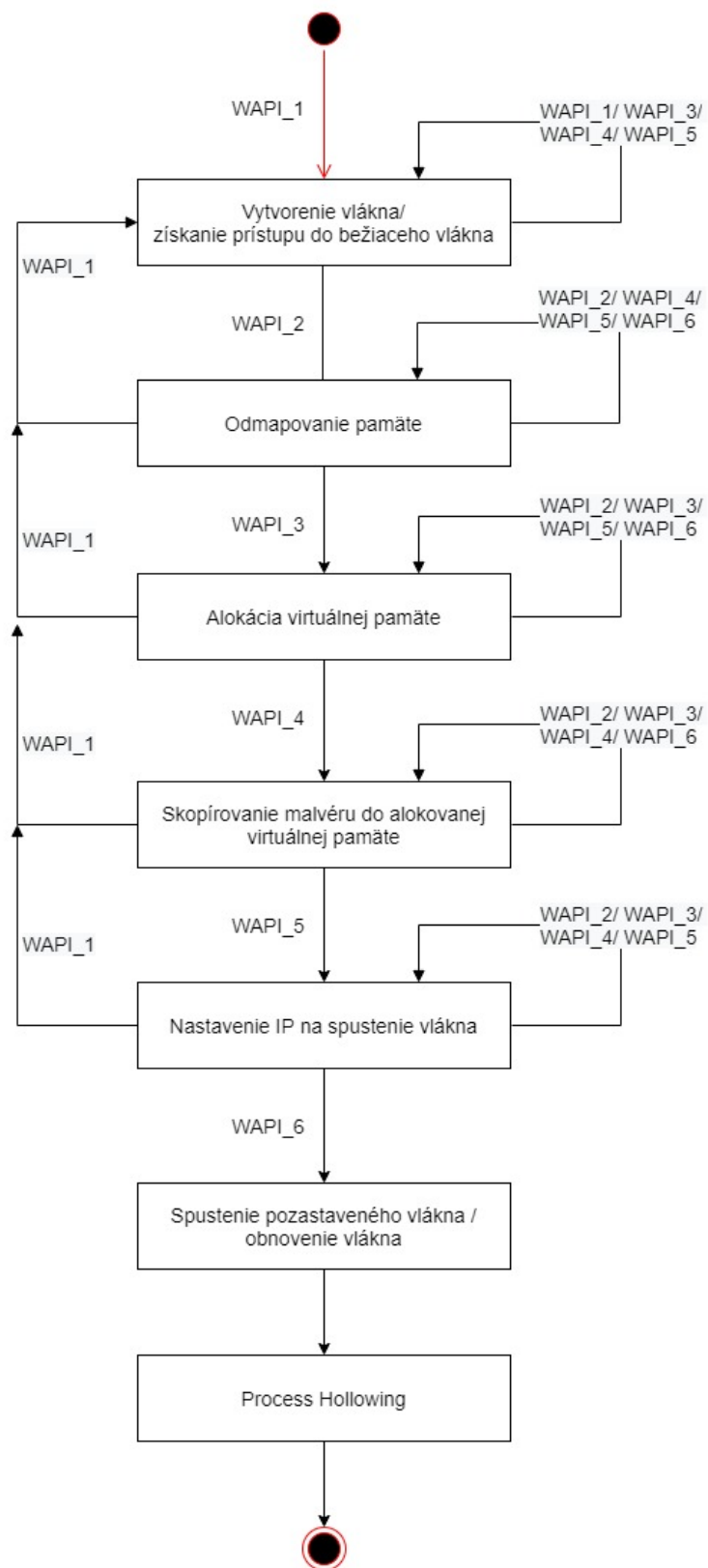
## 4 Algoritmus na detekciu

Nami navrhnutý algoritmus na detekciu techniky *process hollowing* je založený na konečnom stavovom automate. Automat tvorí celkovo šesť prechodových stavov (a jeden počiatočný a jeden konečný stav). Cez tieto stavy postupne prechádzame pri zachytávaní niektorých API volaní. Automat obsahuje nasledujúce stavy:

- **Vytvorenie pozastaveného procesu**
- **Odmapovanie pamäte**
- **Alokácia pamäte**
- **Skopírovanie škodlivého kódu**
- **Nastavenie IP**
- **Spustenie procesu**

Vstupné symboly, ktoré spôsobujú prechody v konečnom automate sú v našom prípade API volania. Nami zvolené API volania môžeme rozdeliť do nasledujúcich skupín:

- **WAPI1** - sú API funkcie, ktoré umožňujú vytvárať nové vlákna/procesy alebo získať prístupy do už existujúcich vlákien/procesov, t.j. *CreateThread*, *SuspendThread*, *CreateProcessA*.
- **WAPI2** - predstavujú API volania primárne určené na odmapovanie pamäte existujúceho procesu, t.j. *NtUnmapViewOfSection*.
- **WAPI3** - volania slúžiace na alokáciu pamäte vo virtuálnom adresnom priestore procesu, t.j. *VirtualAlloc*, *VirtualAllocEx*.
- **WAPI4** - tieto funkcie umožňujú manipulovať s pamäťou; kopírovanie pamäte, zapisovanie a čítanie, t.j. *WriteProcessMemory*, *CopyMemory*, *ReadProcessMemory*.
- **WAPI5** - API volanie, ktoré nastavuje kontext daného vlákna a umožňujú nastaviť IP - *SetThreadContext*.
- **WAPI6** - poslednou skupinou sú API volania, ktoré spúšťajú pozastavené vlákno, t.j. *ResumeThread*.



Obrázok 2: Konečný stavový automat reprezentujúci postupnosť API volaní vedúcich k ukrytiu malvéru.

Algoritmus na detegovanie funguje na jednoduchom princípe. Podozrivú vzorku spustíme a zachytávame vyššie spomínané API volania. Na základe týchto volaní postupne prechádzame stavmi konečného automatu. Ak sa postupne dostaneme do posledného stavu, predpokladáme že nastal *process hollowing*. Celkové znázornenie automatu môžeme vidieť na obrázku č.2.

## 4.1 Konečný stavový automat

Konečný automat je teoretický výpočtový model používaný v informatike na štúdium rôznych formálnych jazykoch. Popisuje veľmi jednoduchý počítač, ktorý môže byť v jednom z niekoľkých stavov, medzi ktorými prechádza na základe symbolov, ktoré číta zo vstupu. Množina stavov je konečná. Automat nemá žiadnu ďalšiu pamäť, okrem informácie o aktuálnom stave. V informatike sa rozlišuje okrem základného deterministického či nedeterministického automatu tiež Mealyho a Moorov automat.

Konečný automat je definovaný ako usporiadaná päťica  $(\Sigma, K, q_0, \delta, F)$  kde:

- $\Sigma$  je konečná neprázdna množina vstupných symbolov, nazývaných aj abeceda. V našom prípade sa jedná o jednotlivé API volania.
- $K$  je konečná množina stavov ako napr. **alokácia pamäte** alebo **process hollowing**.
- $q_0$  je počiatočný stav a platí  $q_0 \in K$ . V prípade nášho automatu sa jedná o stav **štart**, v ktorom sa čaká na prvé vstupné symboly, ktoré reprezentujú napr. vytvorenie pozastaveného vlákna.
- $\delta$  je prechodová funkcia, popisujúca prechod medzi jednotlivými stavmi. Platí:  $\delta : K \times \Sigma \rightarrow K$ .
- $F$  je množina finálnych akceptačných stavov. V našom automate predstavuje stav, v ktorom detegujeme *process hollowing*.

Na začiatku sa automat nachádza v definovanom počiatočnom stave. Ďalej, v každom kroku prečíta jeden symbol zo vstupu a prejde do stavu, ktorý je daný hodnotou, ktorá v prechodovej tabuľke zodpovedá aktuálnemu stavu a prečítanému symbolu. Potom pokračuje čítaním ďalšieho symbolu zo vstupu a ďalším prechodom podľa prechodovej tabuľky.

Podľa toho, či automat skončí po prečítaní vstupe v stave, ktorý patrí do množiny akceptačných stavov, platí, že automat buď daný vstup prijal, alebo neprijal. Množina všetkých reťazcov, ktoré daný automat prijme, tvorí regulárny jazyk.

## 4.2 Matica konečného stavového automatu

Matica reprezentuje prechody medzi jednotlivými stavmi konečného stavového automatu. Tieto prechody sú definované ako volania vybraných API funkcií, ktoré posúvajú automat cez jednotlivé stavy podľa toho v akom stave je aktuálne automat a do akého nového stavu sa automat dostane. Konečná množina stavov je reprezentovaná stavmi **štart**, **vytvorenie procesu**, **alokácia pamäte**, **kopírovanie malvéru**, **nastavenie IP** a **spustenie vlákna**. Počiatočný stav automatu je **štart** a konečný stav je **process hollowing**, ktorý označuje že nastalo injektovanie kódu. Maticu prechodov možno vidieť v tabuľke č.2.

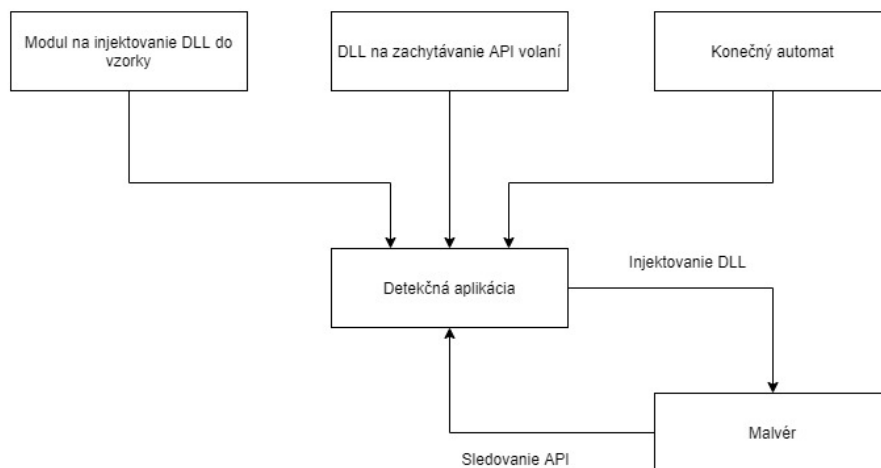


	Prechodové stavy						
	Štart	Vytvorenie procesu / Získanie prístupu do procesu	Odmapovanie pamäte existujúceho procesu	Alokácia virtuálnej pamäte	Kopírovanie malvéru do alokovanej pamäte	Nastavenie IP na spustenie malvéru	Spustenie vlákna / Spustenie pozastaveného procesu
<b>Volané API funkcie</b>							
Skratky stavov	S0	S1	S2	S3	S4	S5	S6
CreateThread	S1	S1	S1	S1	S1	S1	S1
CreateRemoteThread	S1	S1	S1	S1	S1	S1	S1
CreateRemoteThreadEx	S1	S1	S1	S1	S1	S1	S1
CreateProcessA	S1	S1	S1	S1	S1	S1	S1
CreateProcessW	S1	S1	S1	S1	S1	S1	S1
SwitchToThread	S1	S1	S1	S1	S1	S1	S1
OpenThread	S1	S1	S1	S1	S1	S1	S1
SuspendThread	S1	S1	S1	S1	S1	S1	S1
NtUnmapViewOfSection	S0	S2	S2	S3	S4	S5	S6
VirtualAlloc	S0	S1	S3	S3	S4	S5	S6
VirtualAllocEx	S0	S1	S3	S3	S4	S5	S6
CopyMemory	S0	S1	S2	S4	S4	S5	S6
ReadProcessMemory	S0	S1	S2	S4	S4	S5	S6
WriteProcessMemory	S0	S1	S2	S4	S4	S5	S6
ResumeThread	S0	S1	S2	S3	S4	S6	S6
SetThreadPriority	S0	S1	S2	S3	S5	S5	S6

Tabuľka 2: Matica prechodov medzi stavmi pri API volaniach.

## 5 Implementácia

Pre implementáciu riešenia sme si zvolili programovací jazyk C++, pretože našim primárnym testovacím prostredím bol operačný systém Windows. Aplikáciu sme implementovali v prostredí *Microsoft Visual Studio 2019*. Celkovú štruktúru našej aplikácie môžeme vidieť na obrázku č.3.



Obrázok 3: Zobrazenie jednotlivých modulov riešenia a vzťahov medzi nimi.

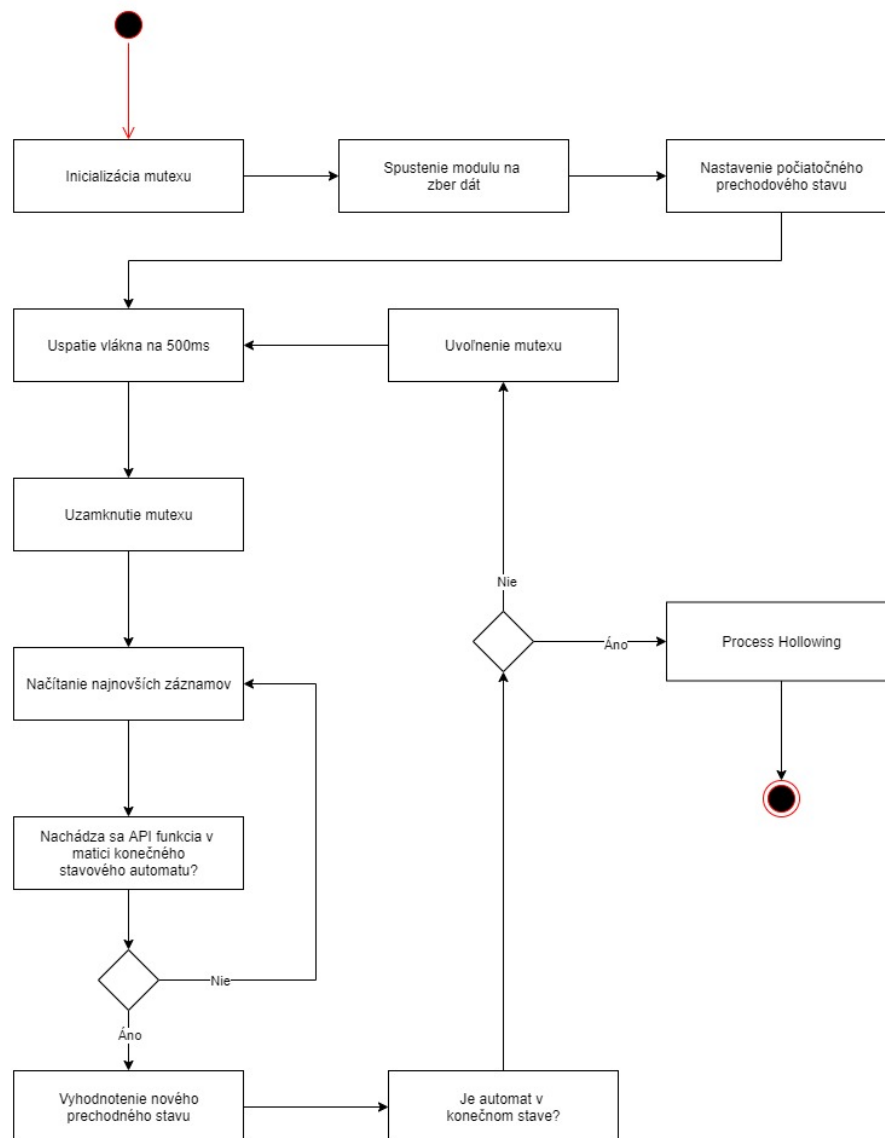
Aplikácia sa skladá z troch základných modulov:

- *DLL na zachytávanie API volaní*; tento modul sa skladá z dynamickej knižnice *Hook.dll*, ktorá slúži na zachytávanie API volaní. Hlavným cieľom nami implementovanej knižnice *Hook.dll* je zachytávať zvolené API volania a zapisovať ich do súboru, spolu s jedinečným identifikátorom. Tento identifikátor slúži na definovanie jednoznačného poradia v akom sa volali jednotlivé API funkcie a taktiež na uľahčenie čítania ďalšími modulmi.
- *Modul na injektovanie DLL do vzorky*; cieľom modulu je injektovať vyššie spomínaný DLL súbor do používateľom vybranej vzorky. Injektovanie prebieha štandardnou formou, ako pri škodlivom kóde: získame prístup k procesu, alokujeme pamäť na cestu k DLL súboru a spustíme vlákno, ktoré danú knižnicu načíta.
- *Konečný automat*; táto časť aplikácie má viacero funkcií. Samotný konečný automat sa načítava z konfiguračného súboru (v ktorom sú definované jednotlivé prechody). Modul v pravidelných intervaloch číta súbor so zapísanými API volaniami a podľa prečítaných záznamov sa posúva v konečnom automate. V momente keď sa

dostaneme do konečného stavu, aplikácia oznámi detekciu techniky *process hollowing*.

## 5.1 Algoritmus

Celkový priebeh detekčného algoritmu môžeme vidieť na obrázku č.4. Aplikácia na začiatku inicializuje pomenovaný mutex, ktorý slúži na synchronizáciu čítania/zapisovania súboru s API volaniami. Keďže k nemu pristupujeme z dvoch procesov, bolo treba vytvoriť pomenovaný mutex v systéme. Tento mutex sa vytvára v hlavnej detekčnej aplikácii a pristupujeme k nemu taktiež z nášho DLL. Inicializáciu mutexu môžeme vidieť vo výpise č.1.



Obrázok 4: Stavový diagram priebehu detekcie.

---

```
HANDLE mutexOnThreadSafe;
mutexOnThreadSafe = CreateMutex(NULL, FALSE, TEXT("MutexOnThreadSafe"));
```

---

Výpis č. 1 Vytvorenie mutexu.

Po inicializácii mutexu sa injektuje DLL do vzorky, čím spustíme zber dát a taktiež sa načíta konečný automat z konfiguračného súboru. Ukážku konfiguračného súboru pre API volanie *ResumeThread* môžeme vidieť na obrázku č.5.

start	ResumeThread	start
vytvorenie_Threadu	ResumeThread	vytvorenie_Threadu
odmapovanie_Pamate	ResumeThread	odmapovanie_Pamate
alokacia_Pamate	ResumeThread	alokacia_Pamate
kopirovanie_Pamate	ResumeThread	kopirovanie_Pamate
nastavenie_IP	ResumeThread	spustenie_Threadu
spustenie_Threadu	ResumeThread	spustenie_Threadu

Obrázok 5: Ukážka konfiguračného súboru pre API ResumeThread.

Konfiguračný súbor sa skladá z troch stĺpcov a obsahuje definíciu prechodov pre každú nami vybranú API funkciu. Každý jeden prechod je definovaný trojicou: aktuálny stav, API volanie, nový stav. Ukážku súboru so zapísanými API volaniami môžeme vidieť na obrázku č.6. Jednotlivé riadky tvoria dvojice: jednoznačný identifikátor a názov API funkcie, ktorá bola volaná škodlivou vzorkou.

```
100000;CreateProcessA
100001;NtUnmapViewOfSection
100002;ReadProcessMemory
100003;ReadProcessMemory
100004;NtUnmapViewOfSection
100005;VirtualAllocEx
100006;WriteProcessMemory
100007;WriteProcessMemory
100008;WriteProcessMemory
100009;WriteProcessMemory
100010;WriteProcessMemory
```

Obrázok 6: Ukážka zapisovaných API volaní.

Po počiatočnej inicializácii sa spustí vlákno, ktoré v pravidelných intervaloch číta súbor so zapísanými API volaniami. Pred samotným čítaním je potrebné uzamknúť zdieľaný mutex aby sme zabezpečili integritu dát. Vo vlákne sa následne načítajú najnovšie záznamy (po dokončení čítania sa vždy uloží identifikátor posledného načítaného záznamu) a aktualizuje sa konečný automat. Ak sa automat nachádza v konečnom stave, aplikácia oznámi detekciu techniky *process hollowing*. V opačnom prípade vlákno uvoľní mutex a znovu sa uspí.

## 5.2 Modul na zber dát

Modul na zber dát predstavuje dynamickú knižnicu *Hook.dll* pomocou ktorej sa odchytávajú API volania. Samotný modul sa skladá z dvoch častí. Prvou časťou je definícia smerníka na pôvodnú API funkciu a definícia novej funkcie pre konkrétne API. Táto funkcia najprv vykoná našu funkcionality a následne zavolá pôvodnú funkciu, viď výpis č.2.

---

```
static BOOL(__stdcall *RealSetThreadContext)(HANDLE, const CONTEXT*) =
    SetThreadContext;

BOOL WINAPI HookSetThreadContext(HANDLE hThread, const CONTEXT* lpContext)
{
    writeFunctionToFile("SetThreadContext");
    return RealSetThreadContext(hThread, lpContext);
}
```

---

Výpis č. 2 Definícia API volania SetThreadContext.

Naša funkcionality, ktorá sa má vykonať pri API volaniach je definovaná vo funkcii *writeFunctionToFile*. Cieľom funkcie je uzamknúť zdieľaný mutex a následne zapísať do súboru jednoznačný identifikátor spolu s názvom volanej API funkcie. Implementáciu funkcie *writeFunctionToFile* môžeme vidieť vo výpise č.3.

---

```

void writeFunctionToFile(std::string originalFunkcion)
{
    DWORD ret = WaitForSingleObject(hMutex, INFINITE);

    if (ret == WAIT_OBJECT_0)
    {
        std::ofstream myFile("api_data.txt", std::ofstream::app |
            std::ofstream::out);

        if (myFile.is_open())
        {
            myFile << Id << ";" + originalFunkcion << endl;
            myFile.close();
        }
        Id++;
        ReleaseMutex(hMutex);
    }
}

```

---

Výpis č. 3 Implementácia funkcie writeFunctionToFile.

---

```

switch (ul_reason_for_call)
{
    case DLL_PROCESS_ATTACH:
    {
        DetourAttach(&(PVOID&)RealSetThreadContext, HookSetThreadContext);
        DetourTransactionCommit();
        break;
    }
    case DLL_PROCESS_DETACH:
    {
        DetourDetach(&(PVOID&)RealSetThreadContext, HookSetThreadContext);
        DetourTransactionCommit();
        break;
    }
}

```

---

Výpis č. 4 Nahradenie pôvodnej funkcie SetThreadContext.

Druhá časť aplikácie nahrádza volanie pôvodnej API funkcie modifikovanou funkciou. Ako môžeme vidieť vo výpise č.4, pôvodnú funkciu *SetThreadContext* nahradíme funkciou *HookSetThreadContext*. Takýmto spôsobom je nutné špecifikovať všetky API volania, ktoré chceme zachytávať.

### 5.3 Detours

Naša aplikácia využíva knižnicu *Detours* na sledovanie API volaní v module na zber dát. *Detours* je knižnica určená na monitorovanie API volaní v systéme Windows [2]. Táto knižnica podporuje rovnako 32 tak aj 64 bitovú verziu operačného systému Windows. Knižnica uľahčuje prácu vývojárom, ktorý pracujú s rozhraním API volaní. *Detours* nahrádza volanie cieľovej funkcie skokom na používateľom zadanú funkciu. Okrem základných funkcií obsahuje *Detours* aj funkcie na úpravu tabuľky importov, pripojenie ľubovoľných segmentov údajov k existujúcim binárnym súborom a na načítanie DLL do nového procesu. Po načítaní do procesu môže DLL knižnica nahradiť akúkoľvek funkciu v procese, ako sú napríklad rozhrania Windows API.

### 5.4 Konečný automat

Konečný automat je definované ako dvojdimenzionálne pole reprezentované pomocou STL kontajnera *std::vector*. Prvý riadok a prvý stĺpec tohto dvojdimensionálneho poľa je určený na orientáciu pre aplikáciu. Kde riadok určuje aktuálny stav automatu a stĺpec predstavuje API funkcie. Pri načítaní nového záznamu tak v tomto poli aplikácia podľa aktuálne stavu a volanej API funkcie vyhladá nový stav automatu. Toto dvojdimenzionálne pole nieje konečné a umožňuje tak pridávanie nových API funkcií (pridaním do súboru *configuration.txt*) bez nutnosti meniť kód aplikácie.

## 6 Výsledky

Našu aplikáciu sme testovali v prostredí *Virtual Box*, na systéme Windows 10. Hlavným cieľom testov bolo vyskúšať naše riešenie na reálnych vzorkách škodlivého kódu, ktorý využíva *process hollowing*. Pred samotnými testami s reálnymi vzorkami sme počas vývoja a experimentovania pracovali s ukážkovou implementáciou metódy *process hollowing*, dostupnej na [8]. Táto aplikácia dokáže injektovať ľubovoľný spustiteľný kód do procesu zadaného prostredníctvom argumentov príkazového riadku. Po dokončení prvotných experimentov, kedy sme úspešne detegovali *process hollowing* na testovacej aplikácii, sme sa posunuli na škodlivé vzorky. V nasledujúcej kapitole uvádzame popis experimentov so škodlivými vzorkami a výsledky, ktoré sme dosiahli.

### 6.1 Škodlivé vzorky

Primárnym cieľom experimentov bolo otestovať, či naše riešenie dokáže detegovať injektovanie kódu pomocou techniky *process hollowing* aj na reálnych vzorkách. Prvým krokom bolo získať nejaké vzorky, ktoré danú techniku využívajú. Tieto informácie sme získavali z rôznych blogov, resp. služieb ktoré sa venujú dynamickej analýze malvéru v sandboxe ako napr. *any.run*[1]. Samotné vzorky sme následne získavali z [11, 1]. V tabuľke č.3 môžeme vidieť výsledky, ktoré sme dosiahli.

Názov vzorky	MD5	Detekcia	Čas
MSIL/Injector.DXQ	706630a77f06ef8fb90eb312fa2cbfe6	✓	2,203 s
MSIL/Kryptik.FQF	3f385d11f6b438ea963cd49e818b9d90	✓	1,655 s
MSIL/Injector.DXQ	9e298807729dff89b56f68f4d42ffd93	✓	1,101 s
Win32/Spy.Zbot.JF	3cfc97f88e7b24d3ceecd4ba7054e138	✗	-
Win32/Kryptik.AVWC	0d42179ff6c448697b67056aecc91c67	✗	-
Win32/Spy.Zbot.YW	ec2dacdbcf194c1e8c8db2dbec605b83	✗	-
Win32/Kryptik.GLKL	6c129b7ed58900286d6cd3a4e85ca15b	✗	-
Win32/Filecoder.Natas.A	f592e7faba96a23ee25ff25f3779f44f	✗	-

Tabuľka 3: Výsledky experimentov so škodlivými vzorkami.

Samotné experimenty prebiehali následovne. Do predpripraveného adresára sme si pripravili našu finálnu aplikáciu spolu so vzorkami. Našu aplikáciu sme spúšťali vždy s jednou konkrétnou vzorkou. Po spustení, detekčná aplikácia injektovala nami vytvorený DLL súbor, určený na sledovanie API volaní, do konkrétnej vzorky. Časový limit na jednu vzorku sme si stanovili na 5 minút. Ak za tento čas aplikácia nehlásila prítomnosť techniky *process hollowing*, výsledok detekcie sme označili za negatívny a proces ukončili.



Každá vzorka, ktorá sa dostala postupne do posledného stavu v konečnom automate bola označená ako správne detegovaná.

## 6.2 Vyhodnotenie

Z nami vybraných vzoriek aplikácia správne detegovala tri vzorky. Zvyšné vzorky však neboli detegované. Ako môžeme vidieť v tabuľke č.3, čas detekcie v prípade spomínaných troch vzoriek bol krátky. Otázkou však ostávajú zvyšné vzorky, ktoré sa nám nepodarilo detegovať. Príčin môže byť viacero. Prvým dôvodom môže byť, že vzorka používala určitú variáciu techniky *process hollowing*, s ktorou náš konečný automat nepočítal. Otázne môžu byť aj informácie uvedené v blogoch a skutočnosť či dané vzorky skutočne obsahovali injektovanie pomocou *process hollowing*.

Jednou z príčin môže byť taktiež fakt, že niektoré vzorky nemusia vykonávať *process hollowing* priamo. To znamená že daná vzorka môže napr. ukladať spustiteľný súbor na disk a ten následne spúšťať. Injektovanie môže byť následne až v tom druhom procese, v ktorom už nemáme vložený náš DLL súbor a teda nedokážeme sledovať API volania. Niektoré vzorky taktiež zvyknú sťahovať škodlivé súbory z URL adresy. Detailnejšou analýzou sme sa však v práci nezaoberali.

# Záver

Cieľom práce bolo naštudovať a zmapovať aktuálne techniky na ukrytie využívané súčasným malvérom, oboznámiť sa s existujúcimi detekčnými algoritmami a navrhnúť a implementovať vlastný algoritmus na detekciu vybranej techniky.

V súčasnosti škodlivý kód využíva rôzne sofistikované techniky na ukrytie svojej činnosti v systéme. Z týchto techník sa najčastejšie využívajú *DLL Injection*, *Process Hollowing*, *Portable Executable Injection* a *Thread Execution Hijacking*. Nami vybranou technikou ktorou sme sa v práci zaoberali bol *Process Hollowing*. V práci sme sa bližšie oboznámili s touto technikou a aj existujúcimi riešeniami. Nami navrhnutý algoritmus je založený na konečnom stavovom automate a sledovaní vybraných API funkcií, pomocou ktorých býva technika *Process Hollowing* implementovaná. Algoritmus využíva DLL súbor, slúžiaci na sledovanie volaných API funkcií vo zvolenom procese. Tieto volania sú následne vyhodnocované v automate

Výsledky na reálnych škodlivých vzorkách poukazujú na čiastočný úspech v možnosti detegovania spomínanej techniky. Neúspešné pokusy na vybraných vzoriek mohli byť spôsobené viacerými faktormi. Vzorka mohla byť len prechodné úložisko pre skutočný škodlivý kód vykonávajúci *Process Hollowing*, prípadne už nefunkčné pripojenie na servery (odkiaľ vzorka sťahuje ďalšie spustiteľné súbory). Výsledky testov ukazujú, že sledovanie API funkcií môže viesť k odhaleniu *Process Hollowing* techniky. Porovnanie s existujúcimi riešeniami však nebolo možné, keďže cieľom väčšiny súčasných nástrojov je odhaliť injektovaný kód už v napadnutom systéme a teda sa nesústreďujú na detekciu v reálnom čase.

Čo sa týka možných vylepšení do budúcnosti, práca by mohla byť rozšírená o sledovanie ďalších API volaní, pomocou ktorých by teoreticky mohla byť technika *Process Hollowing* implementovaná. Týmto spôsobom by sme vedeli zabezpečiť aj odhalenie rôznych variácií implementácie.

# Zoznam použitej literatúry

- [1] ANY.RUN. <https://app.any.run/submissions/>. [Online; cit. 2020-06-02].
- [2] Detours. <https://github.com/microsoft/Detours>. [Online; cit. 2020-05-15].
- [3] Emotet. <https://www.malwarebytes.com/emotet/>. [Online; cit. 2020-04-01].
- [4] HollowFind. <https://github.com/monnappa22/HollowFind>. [Online; cit. 2020-04-20].
- [5] Malvér. <https://www.eset.com/sk/malver/>. [Online; cit. 2020-04-01].
- [6] McAfee ATR Analyzes Sodinokibi aka REvil Ransomware-as-a-Service - What The Code Tells Us. <https://www.mcafee.com/blogs/other-blogs/mcafee-labs/mcafee-atr-analyzes-sodinokibi-aka-revil-ransomware-as-a-service-what-the-code-tells-us/>. [Online; cit. 2020-04-01].
- [7] PHDetection. <https://github.com/idan1288/PHDetection>. [Online; cit. 2020-04-20].
- [8] Process-Hollowing. <https://github.com/m0n0ph1/Process-Hollowing/>. [Online; cit. 2020-06-02].
- [9] Processthreadsapi.h header - Win32 apps. <https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/>. [Online; cit. 2020-05-15].
- [10] Ten Process Injection Techniques: A Technical Survey Of Common And Trending Process Injection Techniques. <https://www.endgame.com/blog/technical-blog/ten-process-injection-techniques-technical-survey-common-and-trending-process>. [Online; cit. 2020-04-05].
- [11] VirusTotal. <https://www.virustotal.com/gui/home/search/>. [Online; cit. 2020-06-02].
- [12] What is the Mirai Botnet? <https://www.cloudflare.com/learning/ddos/glossary/mirai-botnet/>. [Online; cit. 2020-04-05].
- [13] Zeus Virus. <https://usa.kaspersky.com/resource-center/threats/zeus-virus>. [Online; cit. 2020-04-01].

- [14] CIMPANU, C. Emotet, today's most dangerous botnet, comes back to life. <https://www.zdnet.com/article/emotet-todays-most-dangerous-botnet-comes-back-to-life/>. [Online; cit. 2020-04-05].
- [15] LEWIS, N. What new technique does the Osiris banking Trojan use? <https://searchsecurity.techtarget.com/answer/What-new-technique-does-the-Osiris-banking-Trojan-use>. [Online; cit. 2020-04-01].
- [16] OSBORNE, C. New Dridex malware strain avoids antivirus software. <https://www.zdnet.com/article/new-dridex-malware-strain-avoids-antivirus-software-detection/>. [Online; cit. 2020-04-01].
- [17] SALEM, E. Astaroth malware uses legitimate os and antivirus processes to steal passwords and personal data. <https://www.cybereason.com/blog/information-stealing-malware-targeting-brazil-full-research/>. [Online; cit. 2020-05-05].
- [18] Sikorski, Michael and Honig, Andrew. *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*. USA. 1st. No Starch Press.
- [19] YAROSLAV HAKHAKHIAV, N. F. Osiris: An Enhanced Banking Trojan. <https://research.checkpoint.com/2018/osiris-enhanced-banking-trojan/>. [Online; cit. 2020-04-01].

# Prílohy

A	Štruktúra elektronického nosiča . . . . .	II
B	Technická dokumentácia . . . . .	III

# A Štruktúra elektronického nosiča

Príloha práce obsahuje nasledujúce súbory a adresáre:

- **BP.pdf** - elektronickú formu bakalárskej práce.
- **Source\** - adresár so zdrojovými kódmi k aplikácii.
- **Release\** - adresár s *DetectionApp* aplikáciou.
- **Detours\** - adresár s *Detours* knižnicou.

## B Technická dokumentácia

Technická dokumentácia k jednotlivým adresárom:

- **Source\** - Pre používanie obsahu tohto priečinka a prístupom k zdrojovým kódom aplikácie je nutné mať nainštalované Visual Studio 2019.

Priečinok obsahuje dva podadresáre:

- **DetectionApp\** - priečinok obsahujúci zdrojové kódy k aplikácii *DetectionApp*.
  - **Hook\** - priečinok obsahujúci zdrojové kódy ku skompilovaniu knižnice *Hook.dll* slúžiacej na sledovanie API.
- **Release\** - adresár s plne funkčnou aplikáciou *DetectionApp* a knižnicou *Hook.dll*. Adresár ďalej obsahuje nasledujúce súbory:
    - **configuration.txt** - obsahuje definíciu matice konečného stavového automatu so stavmi a jednotlivými prechodmi.
    - **configFile.txt** - súbor na určenie cesty ku testovacej vzorke pre aplikáciu na testovanie. Je potrebné zadať celú cestu aj s názvom vzorky.
    - **archive.txt** - súbor určený na archiváciu záznamov.
    - **hookApplicationResult.txt** - súbor určený na uchovávanie záznamov z modulu na zber dát pred ich ďalším spracovaním.
  - **Detours\** - adresár s *Detours* knižnicou. Na používanie tejto knižnice je nutné ju extrahovať do priečinku *C:\*.

Cestu ku knižnici je potrebné zadať aj v nastaveniach projektu. V sekcii **Properties\Configuration Properties\VC++Directories\Include Directories** pridáme cestu k priečinku *include* knižnice *Detours*. A cestu k priečinku *library* do **Properties\Configuration Properties\VC++Directories\Library Directories**.

V sekcii **Properties\Configuration Properties\Linker\Input\Additional Dependencies** pridáme názov knižnice *Detours; detours.lib*.