Facultatea de Matematică și Informatică

APLICAȚIE PENTRU DETECȚIA VIRUȘILOR

Coordonator științific:
Conf.univ.dr. Boriga Radu

Student:
Macovei Cosmin-Dumitru

București
2018

Cuprins

Introducere

Ce este un malware ?

Un malware cunoscut şi sub numele de virus este o bucată de cod maliţios sau un program (potenţial) dăunator unui computer, unui utilizator sau unei reţele de internet. Un malware poate executa o varietate de funcţii, incluzând furtul, criptarea sau ştergerea de informatii sau monitorizarea utilizatorilor şi a activitaţii acestora pe computer fără permisiunea lor.

De ce sunt creaţi viruşii ?

Principalele trei motive pentru care sunt creaţi viruşii sunt:

1. obţinerea unor sume de bani
2. obţinerea de informaţii
3. pentru simplul fapt de a cauza probleme altora

O altă întrebare uzuală legată de viruşi este: *Dacă fac un virus foarte complicat voi ajunge faimos şi voi fi angajat de o companie mare/guvern?*. Răspunsul este unul logic şi scurt: NU. Acest răspuns este logic deoarece, dacă creatorii viruşilor ar fi angajaţi atunci s-ar incuraja foarte mult fenomenul de creare a viruşilor, acest lucru fiind totuşi o infracţiune.

Cum au apărut viruşii ?

Prima lucrare academică despre programele care se auto-replică a fost susţinută de John von Neumann în 1949. Structura folosită de acesta a fost considerată a fi primul virus din lume la nivel teoretic. Bob Thomas în 1971 a creat la nivel experimental un virus numit Creeper. Acesta este considerat a fi primul virus de tip worm.

Dacă la început viruşii au fost creaţi doar la nivel experimental, în prezent aceştia urmăresc furtul de informaţii sau a unei sume de bani.

Ce este analiza malware ?

Analiza de malware este arta de a diseca un malware pentru a înțelege cum funcţioneaza acesta, cum poate fi identificat şi cum poate fi eliminat. Cu numărul mare de programe malițioase existente şi cu cele care apar, analiza de malware este critică pentru o persoană care lucrează în domeniul securitaţii.

Capitolul 1
Analiza malware

Informaţii generale

Scopul analizei malware este acela de a determina exact ce poate face un executabil (potenţial dăunător), cum poate fi detectat şi care sunt daunele cauzate de acesta.

Un malware este identificat cu ajutorul unei semnături, aceasta fiind aleasă în urma analizei malware. O semnătură este de obicei un şir unic de biţi din conţinutul virusului sau chiar o parte din codul assembly al acestuia.

Tehnici de analiză malware

De cele mai multe ori când o persoană analizează un malware, aceasta are doar executabilul fără codul din spate ceea ce face ca acest proces de analiză să fie unul destul de dificil.

Analiza malware se împarte în analiză statică şi analiză dinamică. Analiza statică constă în studierea executabilului fără executarea virusului în timp ce analiza dinamică include şi execuţia acestuia. Ambele metode se clasifică în ‘de bază’ şi ‘avansată’.

Analiza statică de bază constă în examinarea executabilului fără studierea instrucţiunilor acestuia. Această metodă de analiză poate confirma dacă un fişier este maliţios şi poate oferi informaţii despre funcţionalitatea acestuia. Analiza statică de bază este o metodă rapidă însă ineficientă împotriva viruşilor mai sofisticaţi şi poate omite paşi importanţi din comportamentul virusului. Un site folosit pentru acest tip de analiză este virustotal.com.

Analiza dinamică de bază constă in principal în executarea virusului şi observarea comportamentului acestuia cu scopul de a găsi o metodă de a elimina infecţia sau de a produce o semnătură. Înainte de a executa virusul trebuie făcut un mediu sigur pentru aceasta care să nu permită afectarea propriului sistem sau a reţelei de internet.

Analiza statică avansată constă în procesul de reverse-engineering al virusului şi în analiza intrucţiunilor rezultate în urma acestuia. Această metodă necesită cunoştinţe avansate de limbaj de asamblare şi de sisteme de operare.

Analiza dinamică avansată constă în utilizarea unui debugger pentru a studia starea în care se află virusul în timpul rularii acestuia. Această metodă permite extragerea unor informaţii detaliate dintr-un executabil, informaţii greu de obţinut prin alte metode.

Tipuri de viruşi

Analiza malware devine mai rapidă în momentul în care se ştie despre ce tip de virus este vorba. Majoritatea viruşilor se încadrează în următoarele categorii:

Backdoor Este un program malițios care odată ce a fost instalat pe un computer oferă acces atacatorului. De obicei aceşti viruşi îi oferă atacatorului posibilitatea de a se conecta la calculatorul infectat şi de a executa comenzi fără a fi nevoie de autentificare.

Worm Este un program care se autoreplică pentru a infecta şi alte calculatoare.

Keylogger Acest tip de virus înregistrează tot ce scrie utilizatorul şi trimite aceste informaţii atacatorului.

Ransomware Este un program care blochează sistemul până când este plătită o răscumpărare. O metodă des folosită este aceea de a cripta fişierele şi de a oferi cheia de decriptare în schimbul unei sume de bani.

Exemplu de ransomware (Wannacry) [4]

Trojan Este un program malițios care inițial pare folositor, încercând astfel să convingă victima să îl instaleze. Majoritatea virusurilor de acest tip sunt o formă de backdoor care oferă atacatorului acces la informaţii sensibile precum conturi, parole şi date bancare.

Spyware Este un virus care are rolul de a descoperi informaţii, de obicei despre o organizaţie fără consimţământul acesteia.

Timebomb Acest tip de virus se execută după o anumită dată.

```
#include <ctime>
#include <iostream>
#pragma comment(linker, "/SUBSYSTEM:windows /ENTRY:mainCRTStartup")

using namespace std;
void addToStartUp(char* szBuf) {

 HKEY hkey;

RegOpenKeyEx(HKEY_CURRENT_USER, "SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\Run", 0, KEY_SET_VALUE, &hkey);

RegSetValueEx (hkey, "test", 0, REG_SZ, (LPBYTE) szBuf, strlen(szBuf) + 1);

RegCloseKey(hkey);
}

int main(int argc, char *argv[])
{
    FreeConsole();
    HWND hWnd = GetConsoleWindow();
    ShowWindow( hWnd, SW_HIDE );
    std::ofstream of( "C:\\Windows\\outfile.txt" );
    addToStartUp("C:\\Windows\\System32\\drivers\\csrss.exe");

    time_t t = time(0);   // get time now
    struct tm * now = localtime( & t );
    cout << (now->tm_year + 1900) << '-'
         << (now->tm_mon + 1) << '-'
         <<  now->tm_mday
         << endl;
    if(now->tm_mon >= 7 && now->tm_mday >= 15)
        addToStartUp("C:\\Windows\\System32\\drivers\\virus.exe");

```

Exemplu de cod pentru timebomb care se foloseşte de regiştrii din Windows

Un virus de obicei face parte din mai multe categorii simultan. De exemplu un executabil poate cripta fişierele şi poate cere o sumă de bani în schimbul decriptării lor, dar în acelaşi timp poate conţine şi cod care să ajute la răspândirea lui încadrându-se astfel atât în categoria de ransomware, cât şi în categoria de worm.

Obţinerea de informaţii din viruşi

Un prim pas important în analiza unui posibil virus care poate salva timp şi muncă este acela de a-l scana cu mai mulţi antiviruşi care poate l-au indentificat deja ca fiind un virus. Din păcate antiviruşii nu sunt perfecţi. Aceştia se bazează pe o bază de date cu semnături pentru identificarea viruşilor. Din această cauză un virus nou creat sau un virus care nu a fost semnat nu va fi identificat de antivirus.

Programele antivirus folosesc semnături diferite ceea ce face ca scanarea unui posibil virus cu mai mulţi antiviruşi să fie un început bun în analiza malware.

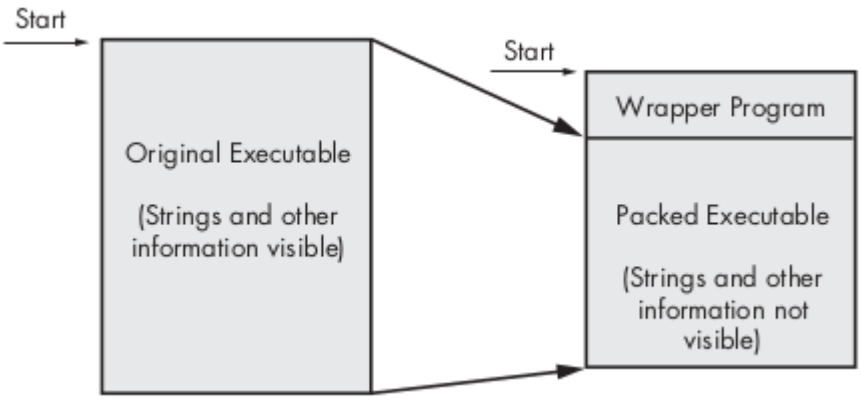
O altă metodă prin care se pot obține informații despre un program este aceea de a analiza secțiunea de stringuri a acestuia. De exemplu dacă un program accesează o pagină web sunt șanse să vedem URL-ul stocat ca string în program. Un exemplu de program care poate fi folosit pentru găsirea string-urilor într-un executabil este programul *strings.* Totuși, string-urile detectate de acest program pot să nu fie string-uri. De exemplu, dacă programul *strings* găsește secvența de biți 0x46, 0x46, 0x33, 0x00 acesta o va interpreta ca fiind string-ul FF3 în timp ce aceasta poate fi doar o simplă adresă de memorie. Filtrarea acestora este datoria persoanei care analizează programul.

```
root@Kali:~/Desktop/learning/security/c++/working_av/viruses# strings wanncry.exe | grep wnry
c.wnry
t.wnry
b.wnryP8
c.wnry%
msg/m_bulgarian.wnry
msg/m_chinese (simplified).wnryR9
msg/m_chinese (traditional).wnry
msg/m_croatian.wnry
msg/m_czech.wnryn
msg/m_danish.wnry
msg/m_dutch.wnry9
msg/m_english.wnryF
msg/m_filipino.wnry
msg/m_finnish.wnry~
msg/m_french.wnry
msg/m_german.wnry
msg/m_greek.wnry4n
msg/m_indonesian.wnry
msg/m_italian.wnry
msg/m_japanese.wnry
msg/m_korean.wnry
msg/m_latvian.wnry`N
msg/m_norwegian.wnry
msg/m_polish.wnry'}7
msg/m_portuguese.wnry
msg/m_romanian.wnry
msg/m_russian.wnry
msg/m_slovak.wnry1
msg/m_spanish.wnry
msg/m_swedish.wnry
msg/m_turkish.wnryO
msg/m_vietnamese.wnry
r.wnry
s.wnry
t.wnry
u.wnryy
b.wnry
c.wnry
msg/m_bulgarian.wnry
msg/m_chinese (simplified).wnry
msg/m_chinese (traditional).wnry
```

Exemplu *strings* (.wnry este extensia fișierelor generate de virusul Wannacry)

Creatorii virușilor folosesc uneori programe pentru comprimarea sau obfuscarea virușilor făcându-i astfel mult mai greu de detectat și de analizat. Aceste programe limitează foarte mult procesul analiza statică.

Executabilele conțin aproape întotdeauna multe string-uri pe când cele comprimate/obfuscate conțin foarte puține. Dacă atunci când analizăm un program observăm că acesta conține foarte puține string-uri cel mai probabil acesta este comprimat sau obfuscat.

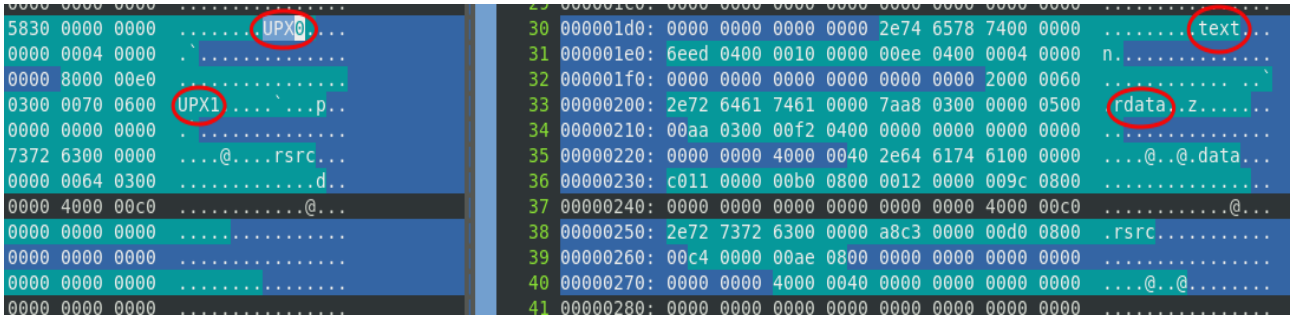


Diagramă executabil comprimat [1]

Un exemplu de program folosit pentru comprimarea executabilelor este *UPX*.

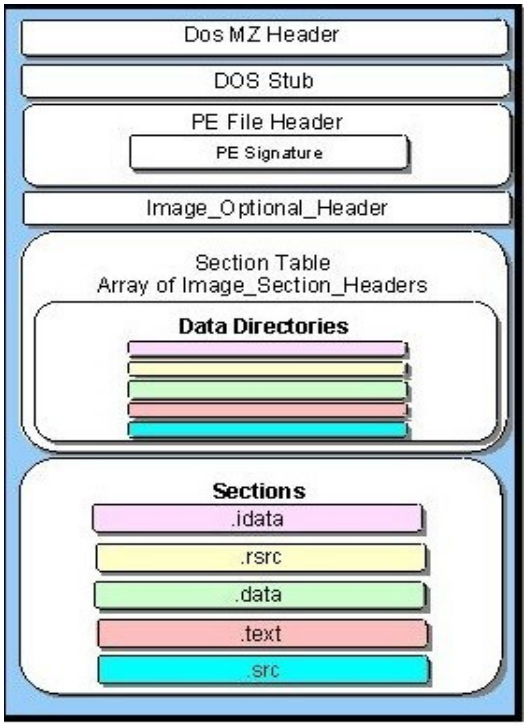
```
root@Kali:~/Desktop/learning/security/c++/working_av/viruses# ls -lth cerber*
-rw-r--r-- 1 root root 605K Nov 20 16:01 cerber.exe
-rw-r--r-- 1 root root 265K Nov 20 16:01 cerber_upxed.exe
```

Exemplu de fişier comprimat cu UPX

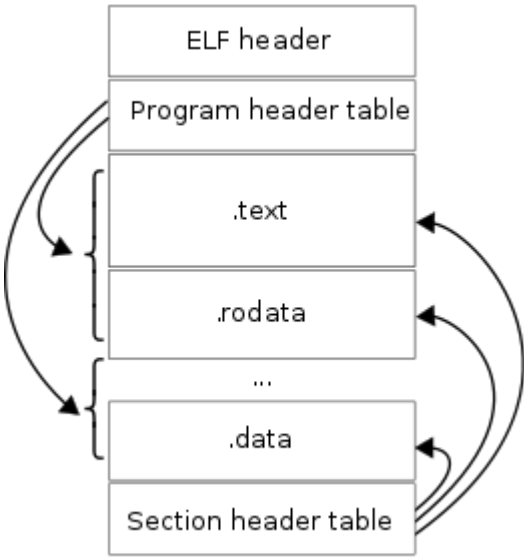


Exemplu diferenţe dintre executabil comprimat şi cel original
(stânga – executabil comprimat, dreapta – executabil original)

Fişierele de tip executabil au un anumit format. Pe Windows formatul fişierelor este Portable Executable (PE) pe când în Linux este Executable and Linkable Format (ELF). Ambele conţin un header care oferă informaţii despre fişierul respectiv.



Structura fişierelor PE [5]



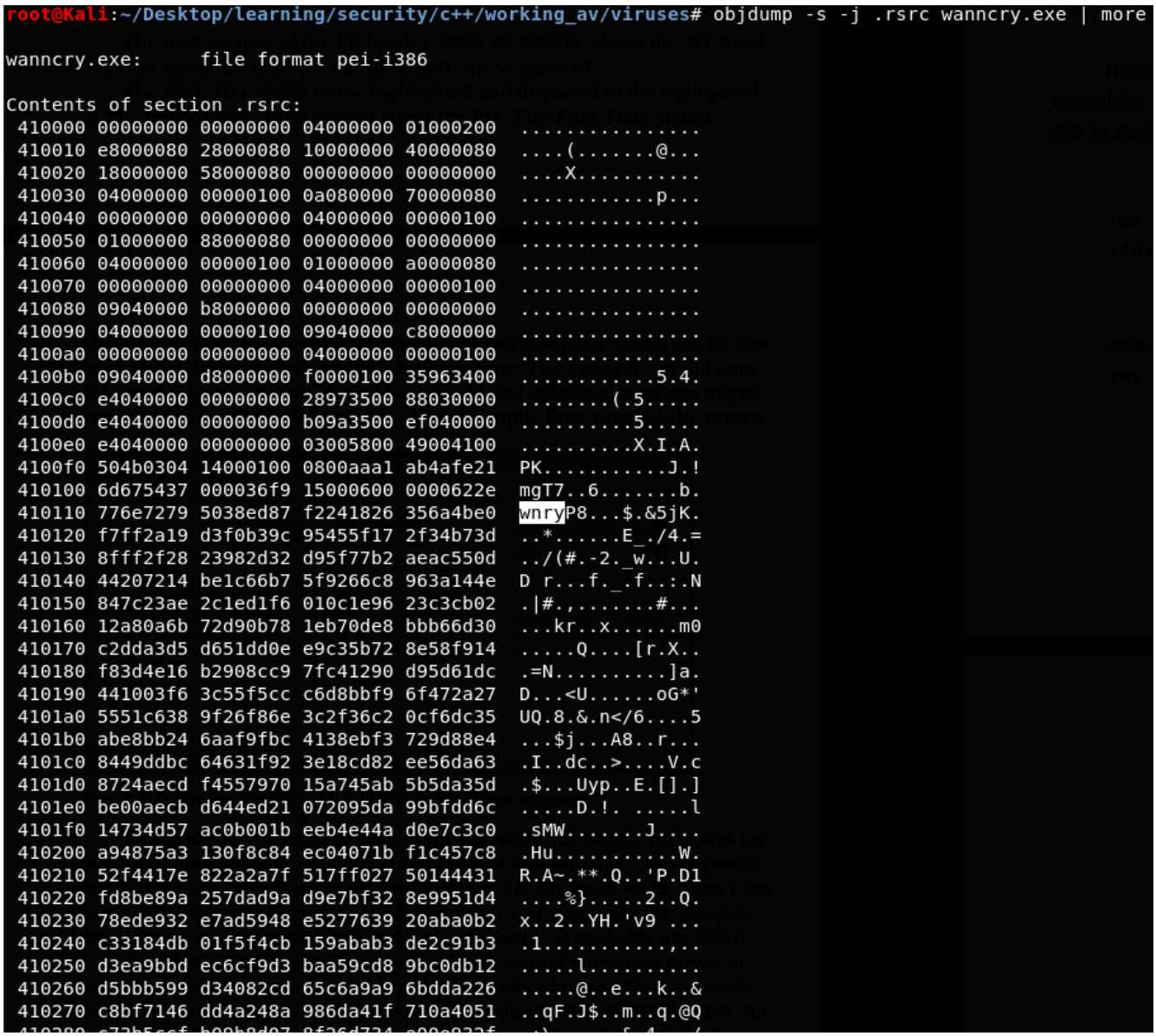
Structura fişierelor ELF [6]

Formatul unui executabil poate oferi de asemenea informaţii importante despre acesta. De exemplu un executabil de tip PE conţine un antet urmat de mai multe secţiuni.

Cele mai comune secţiuni dintr-un executabil de tip PE sunt:

.text	-	Secţiunea .text conţine instrucţiunile executate de către procesor.
.rdata	-	Secţiunea .rdata conţine informaţii despre import-uri şi export-uri. Această secţiune poate conţine şi date de tip read-only folosite de executabil.
.data	-	Secţiunea .data conţine datele globale ale programului
.rsrc	-	Secţiunea .rsrc include resursele folosite de executabil care nu sunt considerate părţi ale acestora cum ar fi imaginile, string-urile.

Putem analiza aceste secţiuni cu ajutorul programului objdump.



Exemplu objdump folosit pentru virusul Wannacry

Observăm că în imaginea anterioară am găsit din nou șirul de caractere *wnry* care reprezintă extensia fișierelor generate de virusul Wannacry.

Capitolul 2
Descrierea proiectului

Informaţii generale

Aplicaţia a fost realizată folosind limbajul C++, Python pentru crearea unor scripturi ajutătoare analizei malware şi introducerii de semnături în baza de date şi MySQL pentru baza de date. Motivul alegerii libajului C++ este libertatea oferită de acesta în ceea ce priveşte accesarea memoriei şi rapiditatea execuţiei, acesta fiind un limbaj compilat spre deosebire de Java care este mai întai compilat într-un byte-code iar mai apoi interpretat de către JVM (Java Virtual Machine).

Aplicaţia a fost structurată pe mai multe clase, fiecare având utilitatea sa in detecţia de viruşi sau a tipului de fişier, sau în criptarea şi stocarea semnăturilor în baza de date.

Parser - clasa în care sunt implementate metodele de detecţie a tipului de fişier (PE sau ELF), citirea antetului acestuia şi semnare a unui fişier, scanarea fişierului şi parcurgerea de directoare în cazul scanării unui folder.

Database - clasa în care sunt implementate conexiunea asupra bazei de date şi operaţiile asupra acesteia, ca de exemplu, returnarea semnăturilor existente sau inserarea uneia noi.

Crypter - clasa în care sunt implementate metodele de criptare ale semnăturilor înainte de a fi inserate în baza de date cu ajutorul funcţiei de hashing MD5.

UPX - clasa în care este implementat algoritmul Rabin-Karp cu hash dublu folosit pentru căutarea semnăturilor care sunt stocate în baza de date în alt executabil. Pe lângă acest algoritm această clasă are implementată şi o metodă pentru detecţia executabilelor comprimate cu UPX.

Algoritmul Rabin-Karp cunoscut şi sub numele de Rolling Hash este un algoritm folosit pentru detecţia de string-uri într-un text. Acesta foloseşte hash-uri pentru detecţie având o complexitate best-case O(n+m) şi worst-case O(nm), unde n este lungimea textului şi m este lungimea pattern-ului căutat.

Funcţia de hash folosită de acest algoritm calculează un număr întreg care reprezintă valoarea numerică a string-ului. De exemplu putem folosi baza 26 în cazul în care avem doar litere mici şi putem folosi codul ASCII pentru a converti fiecare literă în numărul corespunzător acesteia.

De exemplu dacă avem şirul *abc* acesta va fi calculat ca şi:

$$97 * 26^2 + 98 * 26 + 99 = 68219.$$

Pentru a nu depăşi limita superioară a tipului de dată putem efectua acest calcul modulo un număr prim foarte mare pentru a nu avea coliziuni.

De asemenea, pentru a evita şi mai mult coliziunile putem calcula două hash-uri ale string-ului folosind două baze şi două numere prime foarte mari diferite.

O altă alternativă pentru detecţia de string-uri este algoritmul KMP (Knuth-Morris-Pratt). Pentru implementare s-a folosit Rabin Karp cu dublu hash, cu bazele PRIME_BASE = 257, PRIME_BASE_2 = 283 şi numerele prime PRIME_MOD = 1000000007 şi PRIME_MOD_2 = 179426549.

```cpp
int UPX::rollingHash(const string& string_to_search, const string& destination_string)
{
    long long hash1_1 = hashStr1(string_to_search);
    long long hash2_1 = 0;

    long long hash1_2 = hashStr2(string_to_search);
    long long hash2_2 = 0;

    long long power = 1;
    long long power2 = 1;
    for (int i = 0; i < string_to_search.size(); i++)
        power = (power * PRIME_BASE) % PRIME_MOD;
    for (int i = 0; i < string_to_search.size(); i++)
        power2 = (power2 * PRIME_BASE_2) % PRIME_MOD_2;

    for (int i = 0; i < destination_string.size(); i++)
    {
        hash2_1 = hash2_1*PRIME_BASE + destination_string[i];
        hash2_1 %= PRIME_MOD;

        if (i >= string_to_search.size())
        {
            hash2_1 -= power * destination_string[i-string_to_search.size()] % PRIME_MOD;
            if (hash2_1 < 0)
                hash2_1 += PRIME_MOD;
        }

        hash2_2 = hash2_2*PRIME_BASE_2 + destination_string[i];
        hash2_2 %= PRIME_MOD_2;

        if (i >= string_to_search.size())
        {
            hash2_2 -= power2 * destination_string[i-string_to_search.size()] % PRIME_MOD_2;
            if (hash2_2 < 0)
                hash2_2 += PRIME_MOD_2;
        }

        if (i >= string_to_search.size()-1 && hash1_1 == hash2_1 && hash1_2 == hash2_2)
            return i - (string_to_search.size()-1);
    }

    return -1;
}
```

Implementarea algoritmului Rabin-Karp cu dublu hash

Pentru testarea aplicației au fost folosiți viruși deja existenți care au fost rulați și analizați într-un mediu sigur pentru a nu provoca daune nedorite. Acest mediu sigur constă într-o mașină virtuală care nu avea acces la internet pentru a putea evita transmiterea virușilor prin acesta. Ținând cont și de faptul că majoritatea virușilor au fost creați pentru Windows, această mașină virtuală a fost făcută pe Linux pentru o mai bună securitate.

Prima metodă de detecţie

În aplicaţie au fost implementate două metode de detecţie a viruşilor.

Prima metodă constă în obţinerea entry point-ului din antetul fişierului şi citirea primilor 1024 de biţi relativ la acesta. În cazul scanării unui fişier cei 1024 de biţi sunt apoi criptaţi folosind funcţia de hash MD5, iar hash-ul este verificat cu cele existente în baza de date. În cazul semnării unui fişier ca fiind maliţios nu se mai face verificarea cu hash-urile din baza de date, acesta fiind doar stocat în ea.

Mai exact, atunci când un executabil va fi scanat se vor citi primii 1024 de biţi începând de la entry point, se va calcula hash-ul MD5 al acestora şi se va căuta acest hash în baza de date.

MD5 este un algoritm care foloseşte o funcţie de hash pentru a produce un hash de 128 de biţi. Algoritmul a fost inventat de către Ronald Rivest în anul 1991 ca înlocuitor pentru MD4 care s-a dovedit a fi vulnerabil. Una din utilizările acestui algoritm este aceea de a verifica integritatea datelor.

Descrierea algoritmului MD5:

Se adaugă un bit de 1 la finalul mesajului, iar mai apoi se adaugă biţi de 0 până când lungimea acestuia este cu 64 mai mică decât un multiplu al numărului 512. Se adaugă apoi 64 de biţi, aceştia reprezentând lungimea mesajului iniţial modulo 2^{64}.

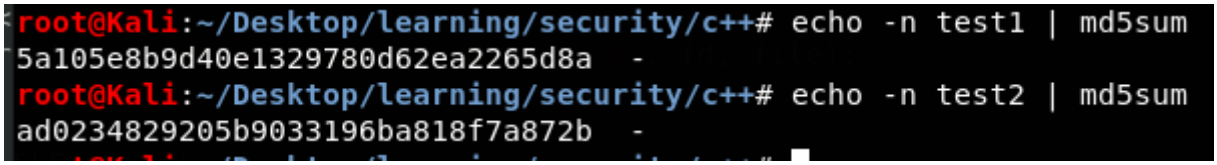
Algoritmul operează pe un bloc de 128 de biţi, aceştia fiind împărţiţi în 4 grupuri de câte 32 de biţi. Vom nota aceste 4 grupuri cu A,B,C şi D. Algoritmul constă în 4 runde, fiecare având 16 operaţii.

F(B,C,D) = (B & C) | (~B & D)
G(B,C,D) = (B & D) | (~D & C)
H(B,C,D) = (B ⊕ C ⊕ D)
I (B,C,D) = C ⊕ (B | ~D)

& - AND , | - OR , ~ - NOT , ⊕ - XOR

Ținând cont de faptul că în cele mai multe cazuri un virus deja creat nu își mai schimbă codul, această metodă este una foarte rapidă și eficientă. Totuși această metodă are dezavantajul că nu poate detecta mai multe executabile cu aceeași semnătură.

Dacă se modifică un singur bit din cei 1024 de biți citiți relativ la entry point atunci hash-ul MD5 se v-a schimba semnificativ.



Exemplu hash md5

Observăm că pentru șirul *test1* obținem hash-ul *5a105e8b9d40e1329780d62ea2265d8a* Dacă schimbăm șirul *test1* în șirul *test2* observăm că hash-ul se schimbă în *ad0234829205b9033196ba818f7a872b* .

Tocmai din acest motiv, această metodă de detecție este una eficientă. Algoritmul MD5 folosește doar operații pe biți, acest lucru făcându-l să fie relativ rapid. Pentru a putea trece de această metodă un creator de virus ar trebui să modifice codul virusului astfel încât atunci când va fi scanat cu un antivirus acesta să nu mai găsească semnătura. Acest lucru este destul de complicat din moment ce creatorul virusului nu va ști exact care octeți din executabil au fost aleși pentru a creea semnătura și pe lângă acest lucru va fi nevoit să transmită din nou virusul. Luând aceste lucruri în considerare, efortul depus de un creator de viruși pentru a modifica virusul ar fi aproape același cu efortul depus pentru a crea un virus nou.

```
11 void Crypter::md5(uint8_t *initial_msg, size_t initial_len) {
12
13     uint8_t *msg = NULL;
14
15
16     uint32_t r[] = {7, 12, 17, 22, 7, 12, 17, 22, 7, 12, 17, 22, 7, 12, 17, 22,
17                     5,  9, 14, 20, 5,  9, 14, 20, 5,  9, 14, 20, 5,  9, 14, 20,
18                     4, 11, 16, 23, 4, 11, 16, 23, 4, 11, 16, 23, 4, 11, 16, 23,
19                     6, 10, 15, 21, 6, 10, 15, 21, 6, 10, 15, 21, 6, 10, 15, 21};
20
21     uint32_t k[] = {
22         0xd76aa478, 0xe8c7b756, 0x242070db, 0xc1bdceee,
23         0xf57c0faf, 0x4787c62a, 0xa8304613, 0xfd469501,
24         0x698098d8, 0x8b44f7af, 0xffff5bb1, 0x895cd7be,
25         0x6b901122, 0xfd987193, 0xa679438e, 0x49b40821,
26         0xf61e2562, 0xc040b340, 0x265e5a51, 0xe9b6c7aa,
27         0xd62f105d, 0x02441453, 0xd8a1e681, 0xe7d3fbc8,
28         0x21e1cde6, 0xc33707d6, 0xf4d50d87, 0x455a14ed,
29         0xa9e3e905, 0xfcefa3f8, 0x676f02d9, 0x8d2a4c8a,
30         0xfffa3942, 0x8771f681, 0x6d9d6122, 0xfde5380c,
31         0xa4beea44, 0x4bdecfa9, 0xf6bb4b60, 0xbebfbc70,
32         0x289b7ec6, 0xeaa127fa, 0xd4ef3085, 0x04881d05,
33         0xd9d4d039, 0xe6db99e5, 0x1fa27cf8, 0xc4ac5665,
34         0xf4292244, 0x432aff97, 0xab9423a7, 0xfc93a039,
35         0x655b59c3, 0x8f0ccc92, 0xffeff47d, 0x85845dd1,
36         0x6fa87e4f, 0xfe2ce6e0, 0xa3014314, 0x4e0811a1,
37         0xf7537e82, 0xbd3af235, 0x2ad7d2bb, 0xeb86d391};
38
39
40     h0 = 0x67452301;
41     h1 = 0xefcdab89;
42     h2 = 0x98badcfe;
43     h3 = 0x10325476;
44
45     int new_len;
46     for(new_len = initial_len*8 + 1; new_len % 512 != 448; new_len++);
47     new_len /= 8;
48
49     msg = (uint8_t*)calloc(new_len + 64, 1);
50
51     memcpy(msg, initial_msg, initial_len);
52     msg[initial_len] = 128;
53
54     uint32_t bits_len = 8*initial_len;
55     memcpy(msg + new_len, &bits_len, 4);
56
57     int offset;
58     for(offset=0; offset<new_len; offset += (512/8)) {
59
60         uint32_t *w = (uint32_t *) (msg + offset);
61
```

Funcția care realizează criptarea cu algoritmul MD5

Metoda checkSignature este apelată atunci când se scanează un executabil cu prima metodă de detecție.

```
57 bool Parser::checkSignature(const char* file) {
58     int32_t fd;
59     Elf32_Ehdr eh;
60     Elf32_Shdr* sh_tbl;
61 
62     fd = open(file, O_RDONLY | O_SYNC);
63     char* pe_file = (char*) malloc(sizeof(char) * strlen(file));
64     strcpy(pe_file,file);
65     if(fd<0) {
66         printf("Error %d Unable to open %s\n", fd, file);
67         return false;
68     }
69 
70     this->read_elf_header(fd, &eh);
71     if(is_ELF(eh)) {
72         Elf64_Addr entry_point  =   this->get_entry_point(eh);
73 
74         lseek(fd, entry_point, SEEK_SET);
75         char buffer[1025];
76         bool isVirus = false;
77         int bytes_number    =   read(fd, buffer, 1024);
78 
79         if(bytes_number > 0) {
80             string signature    =   this->crypt->print_md5(buffer,1024);
81             isVirus         =   this->db->checkSignature(signature);
82             if(isVirus == true) {
83 
84                 cout << "[*] Signature hash : " << signature << "\n";
85 
86             }
87         }
88         return isVirus;
89     }
```

Variabila fd reprezintă descriptorul fișierului pe care dorim să îl scanăm.

Variabila *eh* de tip Elf32_Ehdr reprezintă o structură în care vom stoca antetul unui fișier de tip ELF . Această variabilă are următoarea structură:

Public Attributes

unsigned char	e_ident [EI_NIDENT]
Elf32_Half	e_type
Elf32_Half	e_machine
Elf32_Word	e_version
Elf32_Addr	e_entry
Elf32_Off	e_phoff
Elf32_Off	e_shoff
Elf32_Word	e_flags
Elf32_Half	e_ehsize
Elf32_Half	e_phentsize
Elf32_Half	e_phnum
Elf32_Half	e_shentsize
Elf32_Half	e_shnum
Elf32_Half	e_shstrndx

Structura Elf32_Ehdr (sursa 7)

Verificăm mai apoi tipul fişierului (ELF sau PE) folosind metodele is_ELF şi is_PE.

```
23
24 bool Parser::is_ELF(Elf32_Ehdr eh)
25 {
26     if(!strncmp((char*)eh.e_ident, "\177ELF", 4)) {
27         //printf("ELFMAGIC \t= ELF\n");
28
29         return 1;
30     }
31     else {
32         //printf("ELFMAGIC mismatch!\n");
33
34         return 0;
35     }
36 }
37
38 bool Parser::is_PE(char* fd) {
39     char* pe_magic = (char*) malloc(3 * sizeof(char));
40     FILE* f = fopen(fd,"rb");
41     fread(pe_magic,1,2,f);
42     pe_magic[2] = '\0';
43     if(!strcmp(pe_magic,"MZ")) {
44         free(pe_magic);
45         return true;
46     }
47     else {
48         free(pe_magic);
49         return false;
50     }
51 }
52
```

Pentru a vedea dacă executabilul este de tip ELF comparăm primii 4 octeţi reprezentaţi ca string cu şirul *.ELF* reprezentat în cod ca "\177ELF".

Pentru executabile de tip PE comparăm primii 2 octeţi reprezentaţi ca string cu şirul *MZ*.

După ce am verificat tipul fişierului trebuie să vedem la ce adresă se află entry point-ul, acesta reprezentând adresa de început a codului executabil.

```
Elf64_Addr entry_point  =   this->get_entry_point(eh);

lseek(fd, entry_point, SEEK_SET);
```

În cazul fişierelor ELF ne vom folosi de câmpul *e_entry* din variabila *eh* apelând metoda get_entry_point din clasa Parser. După ce am obţinut adresa entry point-ului ne vom muta la offset-ul respectiv folosind funcţia lseek.

```
52
53 Elf64_Addr Parser::get_entry_point(Elf32_Ehdr elf_header) {
54     //cout << "Entry point : " << elf_header.e_entry << "\n";
55     //printf("Entry point\t= 0x%08x\n", elf_header.e_entry);
56     //cout << hex << elf_header.e_entry << "\n";
57     return elf_header.e_entry;
58 }
```

În cazul fişierelor PE am creat un script în python care parsează header-ul executabilului primit ca argument şi returnează adresa entry point-ului. Antivirusul apelează acest script şi salvează output-ul acestuia într-o variabilă.

După ce am aflat adresa entry point-ului și am mutat cursorul de citire a fișierului citim primii 1024 de biți și apoi calculăm hash-ul MD5 al acestora și îl salvăm ca string în variabila *signature* pentru a-l căuta în baza de date.

```
    Elf64_Addr entry_point  =   this->get_entry_point(eh);

    lseek(fd, entry_point, SEEK_SET);
    char buffer[1025];
    bool isVirus = false;
    int bytes_number    =   read(fd, buffer, 1024);

    if(bytes_number > 0) {
        string signature    =   this->crypt->print_md5(buffer,1024);
        isVirus         =   this->db->checkSignature(signature);
        if(isVirus == true) {

            cout << "[*] Signature hash : " << signature << "\n";

        }
    }
    return isVirus;
```

Pentru a căuta semnătura în baza de date apelăm metoda checkSignature din clasa Database cu hash-ul MD5 calculat anterior. Această metodă folosește un prepare statement pentru a căuta semnătura.

```
62 bool Database::checkSignature(string signature) {
63     int success         =   0;
64     sql::PreparedStatement* pstmt;
65     sql::ResultSet* res;
66     try {
67         pstmt     =   this->con->prepareStatement("SELECT * FROM Signs WHERE signature = ?");
68         pstmt->setString(1,signature);
69         res       =   pstmt->executeQuery();
70         cout << "First method detection : \n";
71         if(res->rowsCount() > 0) {
72             res->next();
73             cout << "\033[1;31mINFECTED FILE\033[0m: ( " << res->getString("name") << " )" << "\n";
74             pstmt->close();
75             delete res;
76             delete pstmt;
77             return true;
78         }
79         else {
80             pstmt->close();
81         }
82         delete res;
83         delete pstmt;
84         cout << "\033[1;32mClean file!\033[0m\n";
85         return false;
86     }
87     catch (sql::SQLException &e) {
88         cout << e.what();
89         cout << " (MySQL error code: " << e.getErrorCode();
90         cout << ", SQLState: " << e.getSQLState() <<
91                 " )" << endl;
92     }
93
94
95 }
```

Dacă interogarea asupra bazei de date a returnat cel puțin o linie, adică în cazul în care găsim semnătura în baza de date afișăm mesajul *INFECTED FILE* împreună cu numele virusului. În caz contrar considerăm că executabilul nu este malițios și afișăm mesajul *Clean File!*

De asemenea, înainte ca un executabil să fie scanat se va face verificarea ca nu cumva acesta să fie comprimat cu UPX.

```
147 bool Parser::scanFile(char* file) {
148     if(this->upx->isUPXFile(file)) {
149         cout << "UPX found\n";
150         bool virus = this->checkSignature((string(file)+".unpacked").c_str());
151         remove((string(file)+".unpacked").c_str());
152         return virus;
153     }
154     else {
155         return this->checkSignature(file);
156     }
157 }
```

În cazul în care executabilul a fost detectat ca fiind comprimat cu UPX vom decomprima executabilul într-un alt fişier cu extensia *.unpacked,* apoi vom scana acest fişier în locul executabilului original şi în final vom şterge acest fişier.

Acest lucru se întâmplă şi atunci când vrem să semnăm un executabil ca fiind maliţios pentru a putea fi detectat mai apoi cu ajutorul acestei prime metode de detecţie. Dacă nu am face această verificare atunci o să alegem ca semnătură primii 1024 de biţi din executabilul comprimat existând astfel şanse ca aceşti biţi să nu fie relevanţi pentru detecţie şi mai mult decât atât, executabilul necomprimat nu va fi identificat ca virus.

```
First method detection :
INFECTED FILE: ( WannaCry )
```

Output-ul primei metode de detecţie

A doua metodă de detecţie

Din cauza faptului că prima metodă de detecţie a viruşilor este limitată la detecţia unui singur executabil am implementat o a doua metodă care se bazează pe analiza mai multor viruşi din aceeaşi categorie şi găsirea unei bucăţi comune de cod assembly. Semnătura constă în opcode-ul instrucţiunilor acestui cod comun.

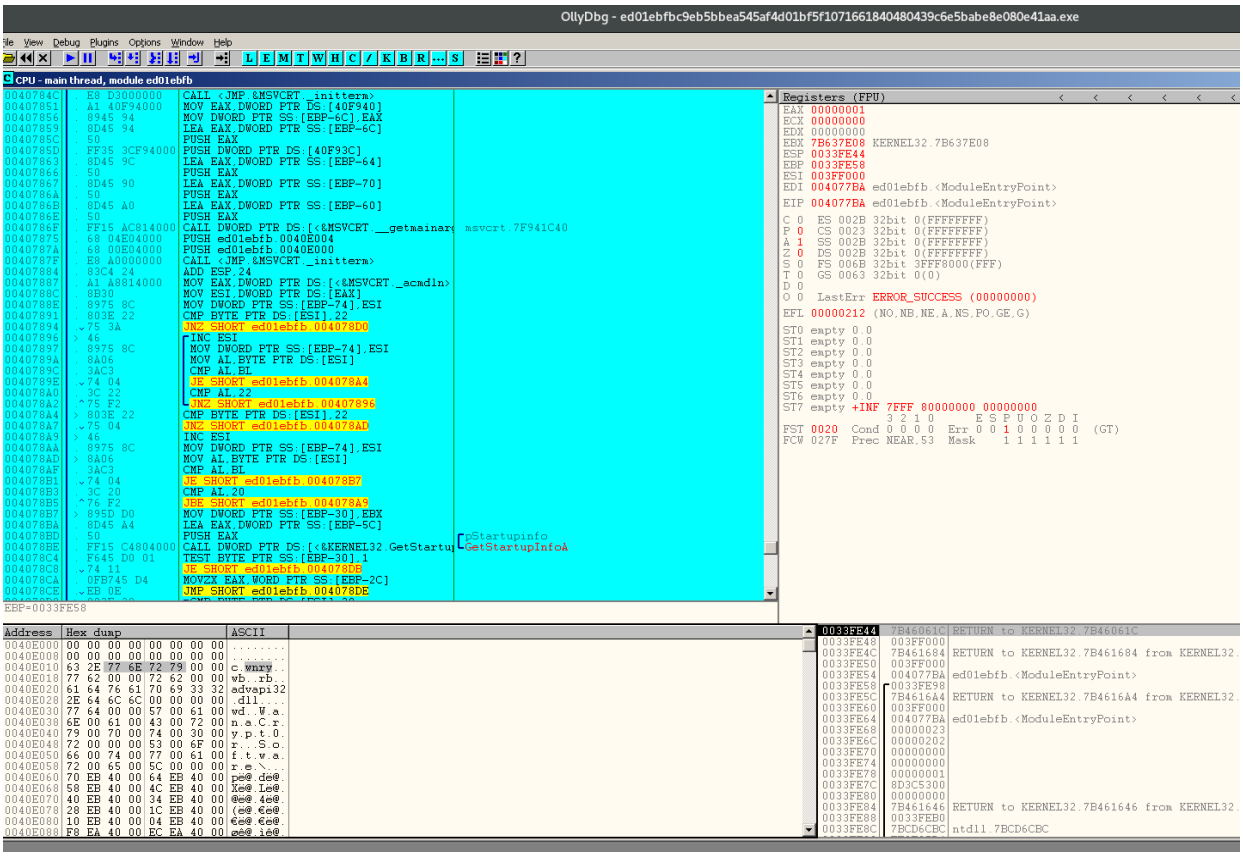
Limbajul assembly este un limbaj de programare low-level care face corespondenţa dintre limbajul de programare si codul maşină. Fiecare limbaj assembly este specific unui tip de arhitectură. Codul assembly este convertit în cod maşină de către un program numit asamblor.

Limbajul assembly foloseşte prescurtări precum MOV, JMP pentru a reprezenta fiecare instrucţiune a codului maşină, instrucţiune numită şi opcode (operation code). Mai exact un opcode reprezintă operaţia care trebuie să fie rulată de procesor.

De exemplu opcode-ul instrucțiunii CMP AL,20 care compară ultimii 8 biți din registrul EAX cu valoarea 20 este *3C 20*.

```
004078B3 | . 3C 20 | CMP AL,20
```

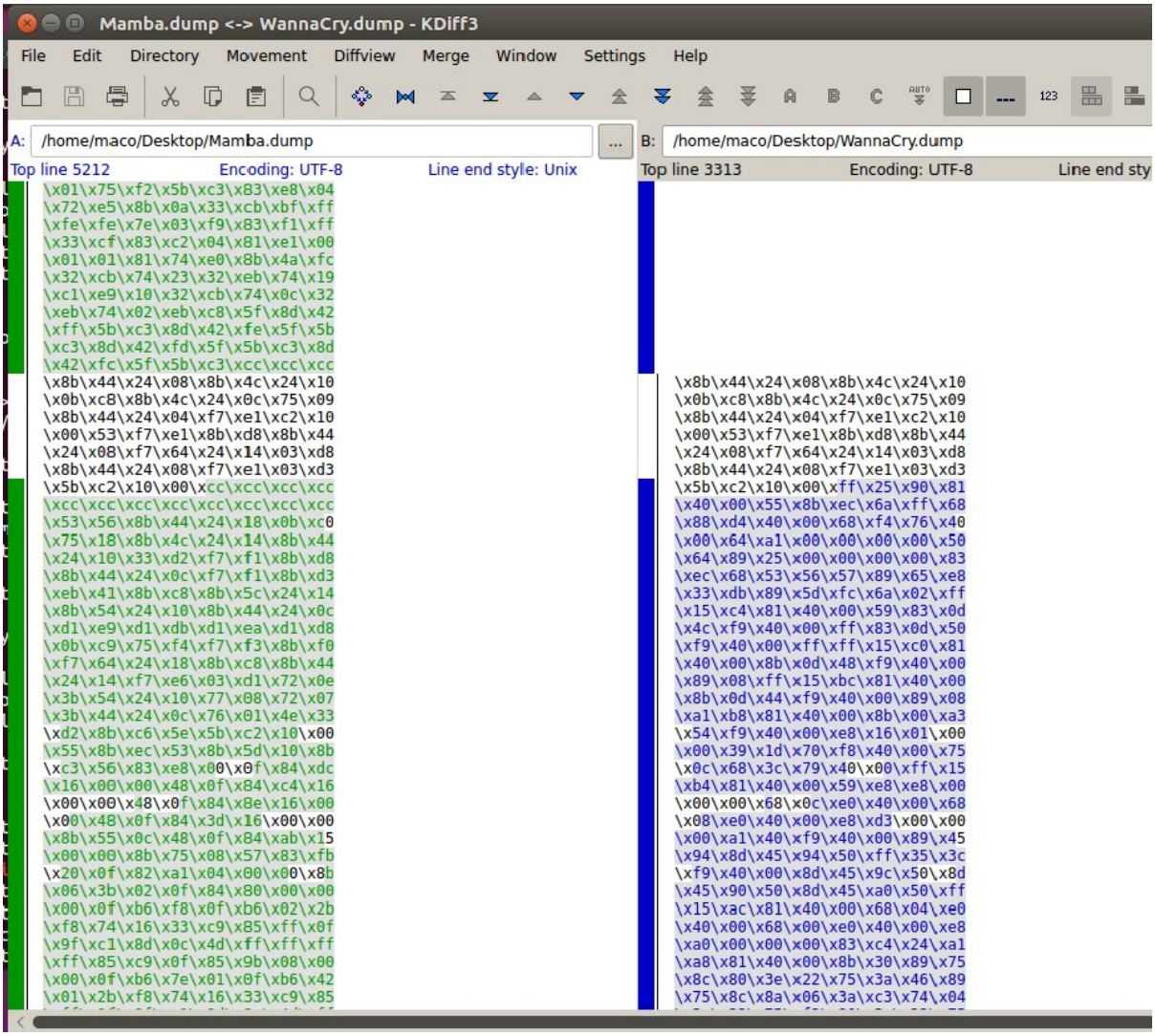
Un exemplu de program cu care putem vizualiza codul assembly al unui executabil folosit și în analiza de malware este ollydbg.



Codul assemby al virusului Wannacry vizualizat în ollydbg

După cum am menționat mai sus a doua metodă de detecție se bazează pe găsirea unei bucăți de cod comun în doi sau mai mulți viruși din aceeași categorie și alegerea acesteia drept semnătură pentru detecție.

Pentru a demonstra această metodă au fost folosiți virușii Wannacry și Mamba, ambii fiind viruși de tip ransomware. Diferența dintre acești doi viruși este aceea că virusul Wannacry criptează fișierele de pe hard-disk în timp ce virusul Mamba criptează MFT-ul (Master File Table). MFT-ul conține informații despre toate fișierele, folderele și sistemul de operare. În cazul infectării cu virusul WannaCry calculatorul va putea porni, în timp ce în cazul infectării cu virusul Mamba sistemul de operare nu va mai putea porni din cauză că nu se pot citi anumite informații despre acesta.



Diferenţe şi asemănări în codul assembly (afişat în hexa)
între viruşii Wannacry şi Mamba

Observăm în poza de mai sus o secvenţă comună, continuă şi destul de lungă pentru a putea fi folosită ca şi semnătură. Astfel, cu această semnătură vom putea detecta ambii viruşi.

```
193 bool Parser::searchHexSignature(char* file) {
194     if(this->upx->isUPXFile(file)) {
195         cout << "UPX found\n";
196         bool virus = this->db->checkHexSignature((char*)(string(file)+".unpacked").c_str());
197         remove((string(file)+".unpacked").c_str());
198         return virus;
199     }
200     else {
201         return this->db->checkHexSignature(file);
202     }
203 }
```

Metoda searchHexSignature din clasa Parser este apelată pentru a scana un executabil cu a doua metodă de detecţie. Această metodă apelează metoda checkHexSignature din clasa Database.

```
Parser.cpp  Crypter.cpp  Database.cpp  UPX.cpp
 97 bool Database::checkHexSignature(char* file) {
 98     int success         =   0;
 99     sql::ResultSet* res;
100     try {
101         //this->pstmt     =   this->con->prepareStatement("SELECT * FROM hexSigns");
102         sql::Statement* stmt = this->con->createStatement();
103         res       =   stmt->executeQuery("SELECT * FROM hexSigns");
104         string file_name(file);
105         size_t found    =   file_name.find_last_of("/");
106         system((string("python3 ") + string(this->cwd) + string("/bin2op.py -f ") + strin
    g(file) + string(" -s > ") + file_name.substr(found+1) + string(".opcodes") ).c_str());
107         FILE* fp    =   fopen((file_name.substr(found+1) + ".opcodes").c_str(),"r");
108
```

Metoda checkHexSignature din clasa Database are ca prim pas generarea opcode-urilor dintr-un executabil folosind un script de python şi salvarea acestora într-un alt fişier.
În acest fişier vom căuta toate semnăturile de acest tip din baza de date.

Pentru căutarea semnăturii am folosit din nou algoritmul Rabin-Karp cunoscut şi sub numele de Rolling Hash, descris mai sus.

Aşadar, după ce am generat fişierul care conţine opcode-urile din executabil vom lua toate semnăturile de acest tip din baza de date şi le vom căuta în acesta. Pentru a face acest lucru vom parcurge pe rând fiecare semnătură şi apoi vom citi conţinutul fişierului care conţine opcode-urile în două variabile diferite *file_content* şi *file_content2*. Ambele variabile vor avea lungimea semnăturii ca să putem verifica toate cazurile. Mai exact dacă de exemplu semnătura ar avea lungimea 20 şi ar începe la offset-ul 10 în fişier atunci prima jumătate din aceasta se va afla în variabila *file_content* iar cealaltă jumătate se va afla în variabila *file_content2*. Deci, în acest caz, dacă vom căuta semnătura separat în cele două variabile nu o vom găsi şi vom presupune că executabilul nu este unul maliţios, cu toate că semnătura există. Aşadar după ce căutăm semnătura separat în cele două variabile o vom căuta inclusiv în concatenarea acestora în cazul în care nu a fost găsită.

Dacă nu am găsit semnătura nici în concatenarea lor atunci vom citi din nou din fişier şi vom relua procesul până când vom găsi o semnătură sau până când nu mai avem semnături în baza de date.

```
115         while(res->next()) {
116             rewind(fp);
117             string signature    =   res->getString("signature");
118             while(!feof(fp)) {
119                 char*   file_content    = new char[signature.size() + 1];
120                 char*   file_content2   = new char[signature.size() + 1];
121                 int nr_bytes = fread(file_content,1,signature.size(),fp);
122                 int nr_bytes2 = fread(file_content2,1,signature.size(),fp);
123                 UPX u;
124                 if(nr_bytes > 0) {
125 
126                     if(u.rollingHash(signature,file_content) != -1 || u.rollingHash(signature,file_content2) !=
    -1) {
127                         cout << "\n\n\033[1;31mINFECTED FILE\033[0m : ( " << res->getString("category") << " )
    \nSignature : " << signature.c_str() << "\n";
128                         remove((file_name.substr(found+1) + ".opcodes").c_str());
129                         //cout << "Signature in file : " << signature << "\n\n";
130                         delete [] file_content;
131                         delete [] file_content2;
132                         delete stmt;
133                         delete res;
134                         fclose(fp);
135                         return true;
136 
137                     }
138                     else {
139                         char* middle    =   new char[2*signature.size() + 2];
140                         strncpy(middle,file_content, signature.size() + 1);
141                         strncat(middle,file_content2, signature.size() + 1);
142                         if(u.rollingHash(signature,middle) != -1) {
143                             cout << "\033[1;31mINFECTED FILE\033[0m : ( " << res->getString("category") << " )
    \nSignature : " << signature.c_str() << "\n";
144                             remove((file_name.substr(found+1) + ".opcodes").c_str());
145 
146                             //cout << "Signature in file : " << signature << "\n\n";
147                             delete [] file_content;
148                             delete [] file_content2;
149                             delete [] middle;
150                             delete stmt;
151                             delete res;
152                             fclose(fp);
153                             return true;
154 
155                         }
156                         else {
157                             delete [] middle;
158                             fseek(fp,-nr_bytes2,SEEK_CUR);
159                         }
160                     }
161                 }
162             }
```

În momentul în care găsim o semnătură în executabil ne oprim şi afisăm mesajul *INFECTED FILE* împreună cu posibila categorie a virusului şi cu semnătura. În cazul în care am parcurs toate semnăturile din baza de date şi nu am găsit niciuna vom considera că executabilul nu este dăunător şi vom afişa mesajul *Clean File!*

```
Second method detection :
INFECTED FILE : ( Ransomware )
Signature : \x8b\x44\x24\x08\x8b\x4c\x24\x10
\x0b\xc8\x8b\x4c\x24\x0c\x75\x09
\x8b\x44\x24\x04\xf7\xe1\xc2\x10
\x00\x53\xf7\xe1\x8b\xd8\x8b\x44
\x24\x08\xf7\x64\x24\x14\x03\xd8
\x8b\x44\x24\x08\xf7\xe1\x03\xd3
\x5b\xc2\x10\x00
```

```
Second method detection :
Clean file!
```

Output-ul celei de-a doua metodă de detecţie

Această metodă de detecţie nu este limitată la folosirea a cel puţin doi viruşi, ea putând fi folosită şi pentru un singur virus în cazul în care vrem să alegem ca semnătură o anumită parte din codul assembly al acestuia. Desigur că acest scenariu presupune cunoştinte mult mai avansate de limbaj de asamblare pentru a putea înţelege codul assembly al virusului şi pentru a putea alege o bucată de cod semnificativă pentru detecţia acestuia.

De exemplu, în cazul unui ransomware un analist de viruşi ar putea folosi analiza malware dinamică avansată pentru a identifica bucata de cod care se ocupă cu criptarea fişierelor şi alegerea acesteia ca şi semnătură.

Cu toate că această metodă pare mai bună din cauza faptului că are capacitatea de a detecta mai mult de un singur virus cu o singură semnătură, ea presupune verificarea tuturor semnăturilor existente în baza de date, lucru care o face să fie mai puţin eficientă decât prima metodă, în special în cazul în care în baza de date avem foarte multe semnături.

De asemenea putem folosi această metodă pentru a detecta viruşi care au codul criptat. Aceşti viruşi se decriptează în momentul rulării încercând astfel să evite detecţia acestora de către antiviruşi. Un exemplu de virus de acest tip este W95/Mad.2736. Pentru a detecta acest tip de viruşi se va folosi drept semnătură bucata de cod care realizează decriptarea.

Pentru a demonstra pe un exemplu acest scenariu am creat un executabil al cărui cod a fost criptat cu operaţia XOR.

În cele ce urmează voi explica cum a fost creat acest executabil si cum este detectat de către antivirus.

Pentru simplitate executabilul are doar o funcţie care afişează un mesaj pe ecran.

```
1  #include <iostream>
2  using namespace std;
3
4  void foo() {
5      cout << "Hello from foo()\n";
6  }
7
8  int main() {
9      foo();
10
11     return 0;
12 }
```

După ce am scris codul şi am generat executabilul am creat un alt executabil care îl citeşte în binar şi îl criptează cu operaţia XOR.

```
 1 #include <stdio.h>
 2 #include <fstream>
 3 #include <iostream>
 4 using namespace std;
 5
 6 int main(int argc, char** argv) {
 7     if(argc >= 3) {
 8         char key = 127;
 9         FILE* f = fopen(argv[1], "rb");
10         FILE* g = fopen(argv[2], "wb");
11         char c;
12         while(fread(&c,1,1,f) == 1) {
13             c ^= key;
14             fwrite(&c,1,1,g);
15         }
16         fclose(f);
17         fclose(g);
18     }
19     return 0;
20 }
```

Acum că avem codul criptat al executabilului într-un fişier mai rămâne să scriem codul de decriptare separat şi să adăugăm codul criptat într-o nouă secţiune din executabil. Pentru această operaţie am folosit *objcopy*. De asemenea executabilul care va decripta acest cod trebuie mai întâi să găsească această secţiune, să citească conţinutul acesteia şi apoi să o decripteze.

```
root@Kali:~/Desktop/learning/learnc++/self_enc# objcopy --add-section toexe=out decrypt decrypt2
```

Observăm că după ce a fost executată această comandă s-a creat o nouă secţiune numită *toexe* în cadrul executabilului care conţine conţinutul fişierului *out*.

```
root@Kali:~/Desktop/learning/learnc++/self_enc# objdump -s -j toexe -M intel decrypt2

decrypt2:     file format elf64-x86-64

Contents of section toexe:
 0000 003a3339 7d7e7e7f 7f7f7f7f 7f7f7f7f  .:39}~~.........
 0010 7c7f417f 7e7f7f7f 6f787f7f 7f7f7f7f  |.A.~...ox......
 0020 3f7f7f7f 7f7f7f7f 0f647f7f 7f7f7f7f  ?........d......
 0030 7f7f7f7f 3f7f477f 767f3f7f 617f627f  ....?.G.v.?.a.b.
 0040 797f7f7f 7a7f7f7f 3f7f7f7f 7f7f7f7f  y...z...?.......
 0050 3f7f7f7f 7f7f7f7f 3f7f7f7f 7f7f7f7f  ?.......?.......
 0060 877e7f7f 7f7f7f7f 877e7f7f 7f7f7f7f  .~.......~......
 0070 777f7f7f 7f7f7f7f 7c7f7f7f 7b7f7f7f  w.......|...{...
 0080 477d7f7f 7f7f7f7f 477d7f7f 7f7f7f7f  G}......G}......
 0090 477d7f7f 7f7f7f7f 637f7f7f 7f7f7f7f  G}......c.......
 00a0 637f7f7f 7f7f7f7f 7e7f7f7f 7f7f7f7f  c.......~.......
```

Mai exact executabilul care va face decriptarea va extrage secțiunea, va decripta conținutul acesteia și îl va scrie într-un alt fișier. Pentru simplitate a fost folosită din nou comanda *objcopy* pentru a extrage conținutul secțiunii adăugate.

```cpp
#include <iostream>
#include <unistd.h>
#include <stdio.h>
#include <string>
using namespace std;

std::string execute(const char* cmd) {
    char buffer[128];
    std::string result = "";
    FILE* pipe = popen(cmd, "r");
    if (!pipe) throw std::runtime_error("popen() failed!");
    try {
        while (!feof(pipe)) {
            if (fgets(buffer, 128, pipe) != NULL)
                result += buffer;
        }
    } catch (...) {
        pclose(pipe);
        throw;
    }
    pclose(pipe);
    return result;
}

int main() {
    execute("objcopy decrypt2 /dev/null --dump-section toexe=toexeout");
    FILE* f = fopen("toexeout","r");
    FILE* g = fopen("virus","wb");
    char key = 127;
    char c;
    while(fread(&c,1,1,f) == 1) {
        c ^= key;
        fwrite(&c,1,1,g);
    }
    fclose(f);
    fclose(g);
    system("./virus");
    return 0;
}
```

Codul care realizează decriptarea

Acum dacă rulăm acest executabil vom observa că vom avea același output ca executabilul de la care am plecat.

```
root@Kali:~/Desktop/learning/learnc++/self_enc# ./self_encrypted
Hello from foo()
root@Kali:~/Desktop/learning/learnc++/self_enc# ./decrypt2
Hello from foo()
root@Kali:~/Desktop/learning/learnc++/self_enc#
```

Pentru a indetifica acest executabil vom alege ca semnătură bucata de cod assembly care realizează decriptarea. Vom folosi comanda objdump pentru a genera codul assembly al executabilului împreună cu opcode-urile.

```
    11eb:       74 29                   je     1216 <main+0xaa>
    11ed:       0f b6 45 bf             movzx  eax,BYTE PTR [rbp-0x41]
    11f1:       32 45 ef                xor    al,BYTE PTR [rbp-0x11]
    11f4:       88 45 bf                mov    BYTE PTR [rbp-0x41],al
    11f7:       48 8b 55 f0             mov    rdx,QWORD PTR [rbp-0x10]
    11fb:       48 8d 45 bf             lea    rax,[rbp-0x41]
    11ff:       48 89 d1                mov    rcx,rdx
    1202:       ba 01 00 00 00          mov    edx,0x1
    1207:       be 01 00 00 00          mov    esi,0x1
    120c:       48 89 c7                mov    rdi,rax
    120f:       e8 ac fc ff ff          call   ec0 <fwrite@plt>
    1214:       eb af                   jmp    11c5 <main+0x59>
    1216:       48 8b 45 f8             mov    rax,QWORD PTR [rbp-0x8]
    121a:       48 89 c7                mov    rdi,rax
```

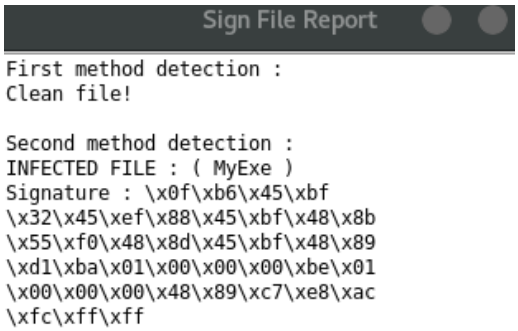
Bucata de cod care realizează decriptarea şi scrierea în fişier

(Pe a doua coloană putem observa opcode-urile fiecărei operaţii)

Aşadar vom alege ca semnătură şirul

\x0f\xb6\x45\xbf\x32\x45\xef\x88\x45\xbf\x48\x8b\x55\xf0\x48\x8d\x45\xbf\x48\x89\xd1\xba\x01\x00\x00\x00\xbe\x01\x00\x00\x00\x48\x89\xc7\xe8\xac\xfc\xff\xff

După ce am adăugat în baza de date această semnătură observăm că fişierul va fi detectat de către antivirus.



Parcurgerea de directoare

Antivirusul are posibilitatea de a scana atât un fişier cât şi conţinutul unui director. În ceea ce priveşte scanarea directoarelor am folosit o metodă recursivă numită scanFolder implementată în clasa Parser.

```
168 void Parser::scanFolder(const char *name) {
169     DIR* dir;
170     struct dirent *entry;
171     cout << "SCANNING " << name << " FOLDER\n\n";
172     if(!(dir = opendir(name))) {
173         cout << "\n" << name << " is NOT a directory\n";
174         return;
175     }
176     chdir(name);
177 
178     while((entry = readdir(dir)) != NULL) {
179         if(strcmp(entry->d_name, ".") == 0 || strcmp(entry->d_name, "..") == 0)
180             continue;
181         else if((isDir(entry->d_name) == true)) {
182             cout << "\n[*] Scaning " << entry->d_name << "...\n\n";
183             this->scanFile(entry->d_name);
184             this->searchHexSignature(entry->d_name);
185         }
186         if(isDir(entry->d_name) == false) {
187             scanFolder(entry->d_name);
188         }
189     }
190     chdir("..");
191     closedir(dir);
192 }
```

Ca prim pas metoda scanFolder verifică dacă valoare de la pointer-ul *name* conţine numele unui director. În cazul în care această valoare nu reprezintă numele unui director afişăm mesajul [*Valoare] is NOT a directory.*

În caz contrar ne mutăm în directorul respectiv şi citim conţinutul acestuia eliminând înregistrările ‘.’ şi ‘..’ care reprezintă directorul curent, respectiv directorul părinte. Dacă înregistrarea actuală reprezintă un fişier atunci îl scanăm cu ajutorul metodei scanFile şi searchHexSignature.

Dacă înregistrarea actuală reprezintă un director apelăm din nou metoda scanFolder cu numele acestuia.

În momentul în care am terminat de scanat toate înregistrările ne mutăm în directorul părinte şi închidem directorul pe care tocmai l-am scanat pentru a putea scana alte posibile subdirectoare.

Capitolul 3
Analiza virusului Wannacry

Informaţii generale

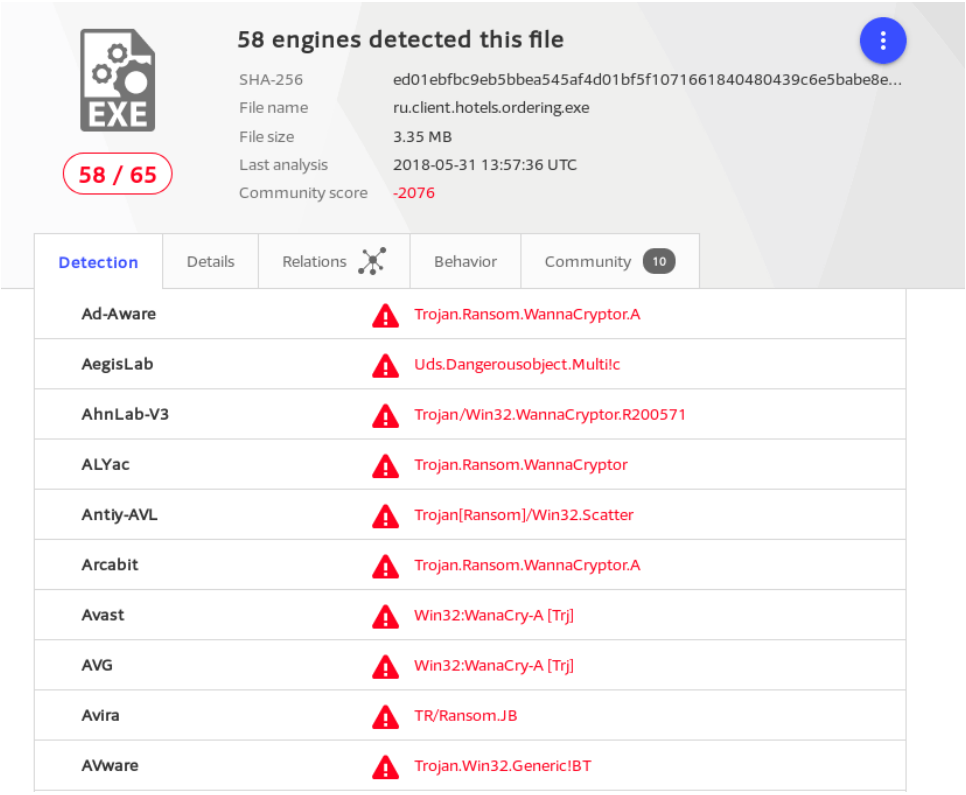
Virusul Wannacry se încadrează în categoria viruşilor de tip ransomware fiind în acelaşi timp şi un virus de tip worm. Infecţia acestui virus a început în data de 12 mai 2017 afectând nenumărate calculatoare care aveau ca sistem de operare Windows. Acest virus a afectat companii şi utilizatori în aproximativ 150 de ţări, incluzând inclusiv agenţii guvernamentale.

Acest virus criptează fişierele punându-le apoi extensia .wncry

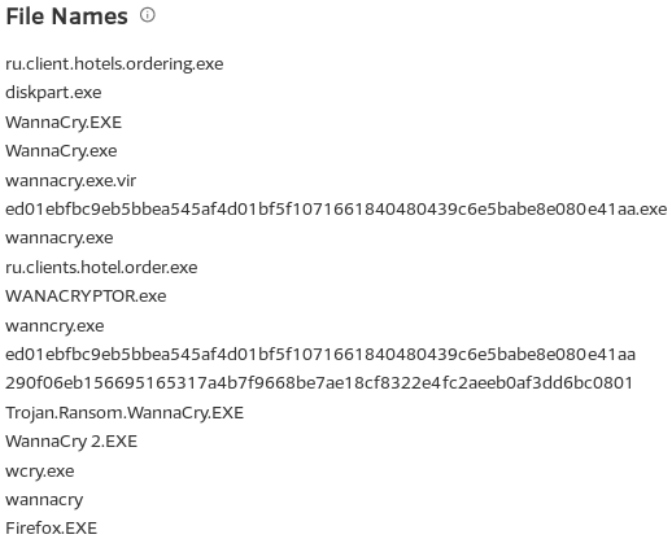
Wannacry foloseşte o vulnerabilitate a serviciului de SMB versiunea 1 numită EternalBlue pentru a se transfera de la un dispozitiv la altul. Această vulnerabilitate oferă posibilitatea de a executa cod malițios pe o altă maşină fără a fi nevoie de autentificare.

Analiza executabilului

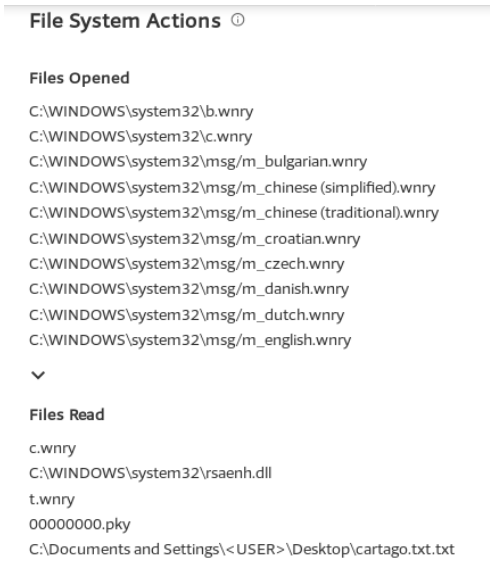
Pentru început putem uploada executabilul pe virustotal.com pentru o analiză statică a acestuia. Observăm că acesta a fost identificat ca virus de către 58 de antiviruşi din 65 având un scor de -2076.



De asemenea putem observa şi alte nume ale acestui executabil.

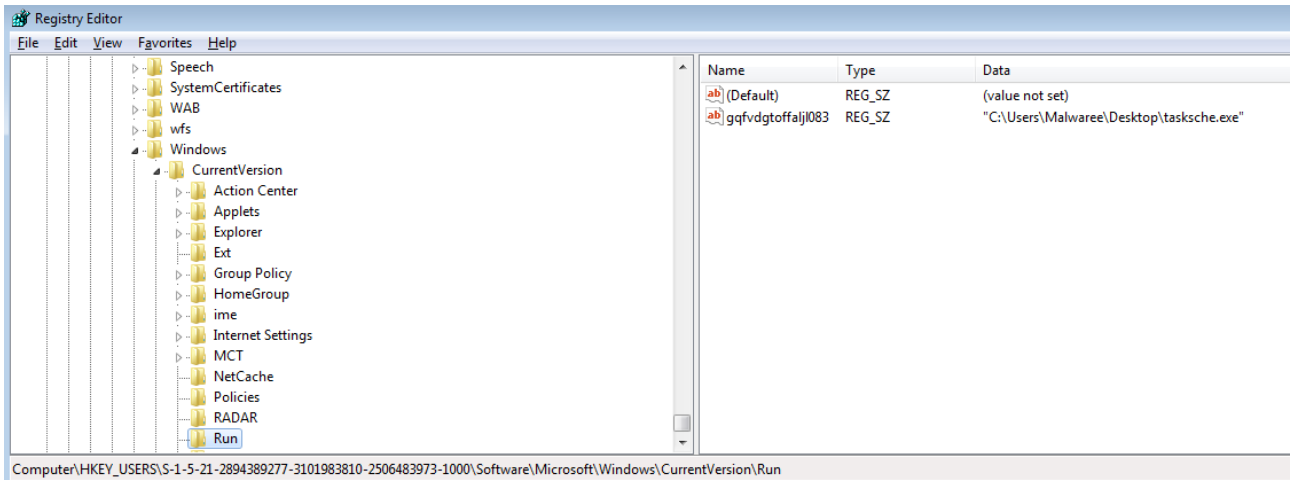


Putem găsi înclusiv acţiunile acestui executabil asupra altor fişiere.



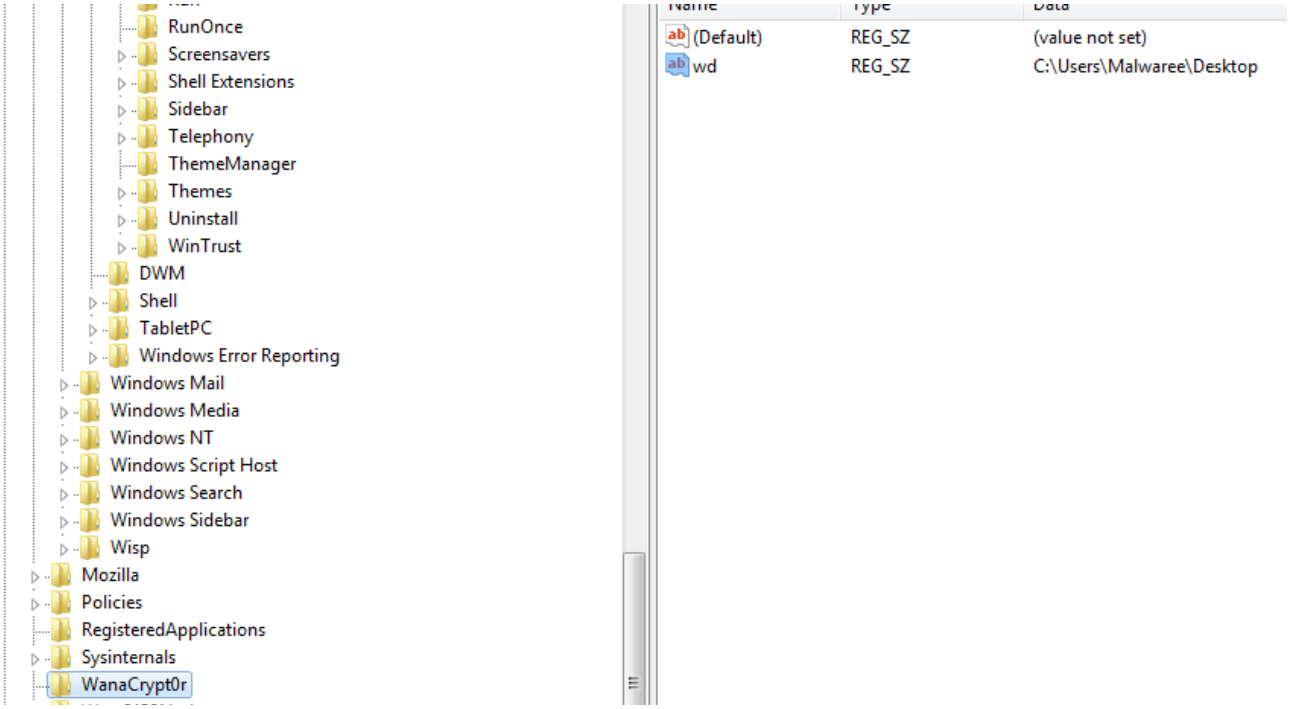
În urma rulării virusului într-o maşină virtuală constatăm că acesta porneşte automat la deschiderea acesteia.

Pentru ca virusul să se execute automat la fiecare pornire a dispozitivului acesta creează un nou serviciu numit tasksche.exe şi apoi adaugă o nouă valoare în registrul *HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run* . Acest registru are rolul de a rula programe atunci când userul se loghează.



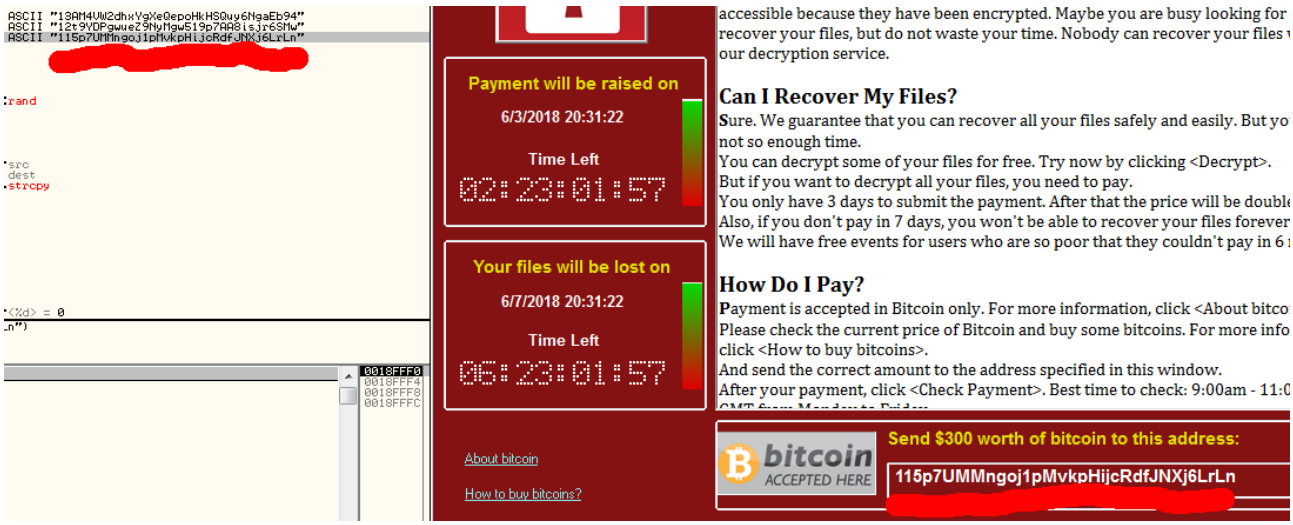
Observăm că s-a creat o înregistrare nouă cu numele *gqfvdgtoffaljl083* care conţine calea către executabilul tasksche.exe creat de virus.

Observăm că virusul salvează în registrul *HKEY_CURRENT_USER\Software\WanaCrypt0r* calea de unde a fost rulat prima dată.



Suma de bani necesară decriptării fişierelor este de 300$, plata făcându-se prin bitcoin.

Dacă vom analiza virusul cu ajutorul programului ollydbg vom găsi adresa la care se va face plata definită ca şir de caractere.



De asemenea, după rularea virusului observăm multe alte fişiere create de către virus.

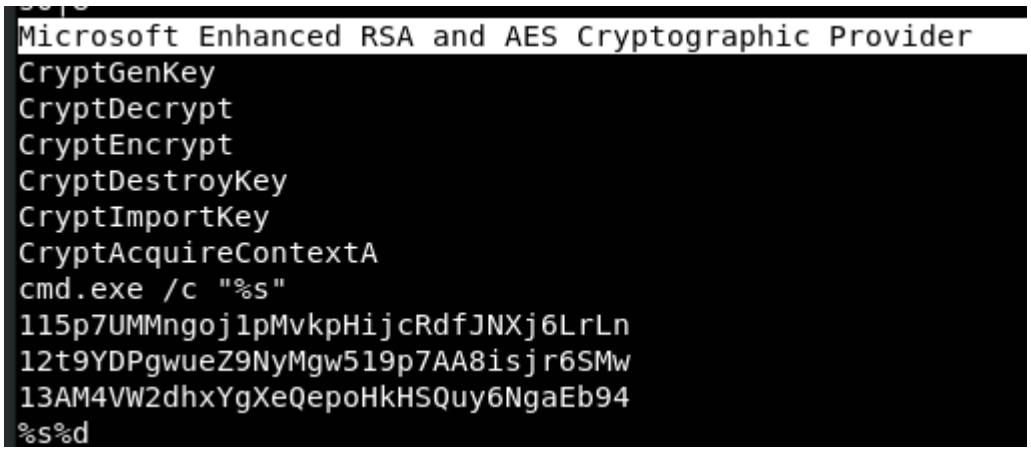
Dacă deschidem fişierul *c* cu un editor de text vom observa că găsim din nou adresa la care se va face plata împreună cu alte url-uri care pot fi accesate doar prin TOR (The Onion Router). Putem presupune deci că virusul comunică cu un server din DarkNet.

Criptarea fişierelor

Din nou vom folosi programul ollydbg pentru a găsi bucata de cod care se ocupă cu criptarea fişierelor.

La offset-ul *401A45* observăm începutul unei subrutine. Această subrutină apelează funcţia *LoadLibraryA* pentru a încărca *advapi32.dll,* librărie în care găsim funcţii precum *CryptDecrypt*, *CryptDeriveKey, CryptDestroyKey, etc.* Mai apoi virusul foloseşte funcţiile de criptare *CryptAcquireContextA, CryptImportKey, CryptDestroyKey, CryptEncrypt, CryptDecrypt* şi *CryptGenKey.*

Dacă afişăm string-urile din executabil putem observa metodele de criptare folosite.

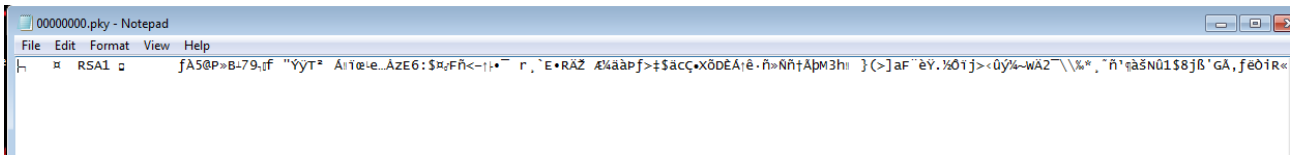


Virusul foloseşte RSA şi AES pentru criptare. Prin urmare virusul WannaCry foloseşte atât criptări simetrice cât şi criptări asimetrice. Mai exact virusul foloseşte criptarea AES-128-CBC pentru criptarea fişierelor şi RSA pentru a se asigura că fişierele nu se pot decripta decât plătind suma de bani cerută.

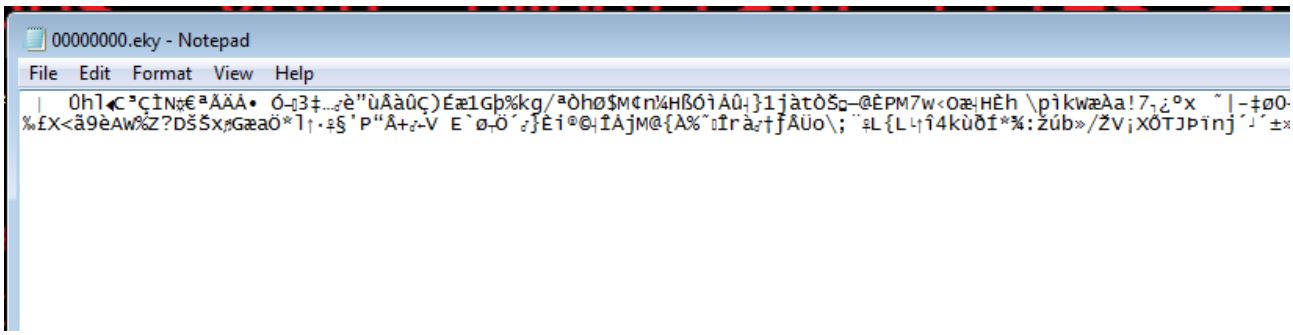
La început viruşii de tip ransomware foloseau doar criptarea asimetrică pentru a cripta fişierele. Mai exact aceştia generau o cheie publică şi una privată de tip RSA, trimiteau cheia privată către un server pentru a o putea returna în momentul în care se făcea plata, iar mai apoi criptau fişierele cu cheia publică. În cazul acestui scenariu criptarea fişierelor dura foarte mult deoarece criptarea asimetrică este destul de înceată, existând în acelaşi timp posibilitatea de a intercepta cheia privată dacă victima ar fi monitorizat traficul de net.

În cazul virusului WannaCry, primul pas este de a genera o cheie publică, o vom nota cu C_{pub} şi una privată, o vom nota cu C_{priv} de tip RSA pe 2048 de biţi. În acelaşi timp în executabil există şi cheia publică de RSA de pe un server, o vom nota cu S_{pub}. După ce s-a generat cheia de tip RSA, virusul criptează cheia privată cu cheia publică de pe server ($S_{pub}(C_{priv})$) , aceasta fiind apoi salvată într-un fişier. Virusul generează apoi o cheie simetrică de tip AES pentru fiecare fişier care va fi criptat de acesta. Vom nota această cheie cu *K*. După ce fişierul a fost criptat, această cheie *K* va fi criptată cu cheia publică C_{pub} şi apoi va fi salvată într-un fişier.

Deci pentru a decripta un fişier trebuie să se facă o *decripţie în lanţ*. Prima dată trebuie să avem cheia privată de pe server S_{priv} pentru a putea decripta cheia privată C_{priv} generată de virus. După ce avem cheia privată C_{priv} putem decripta cheia *K* pentru fişierul respectiv. În final, cu cheia *K* putem decripta fişierul.

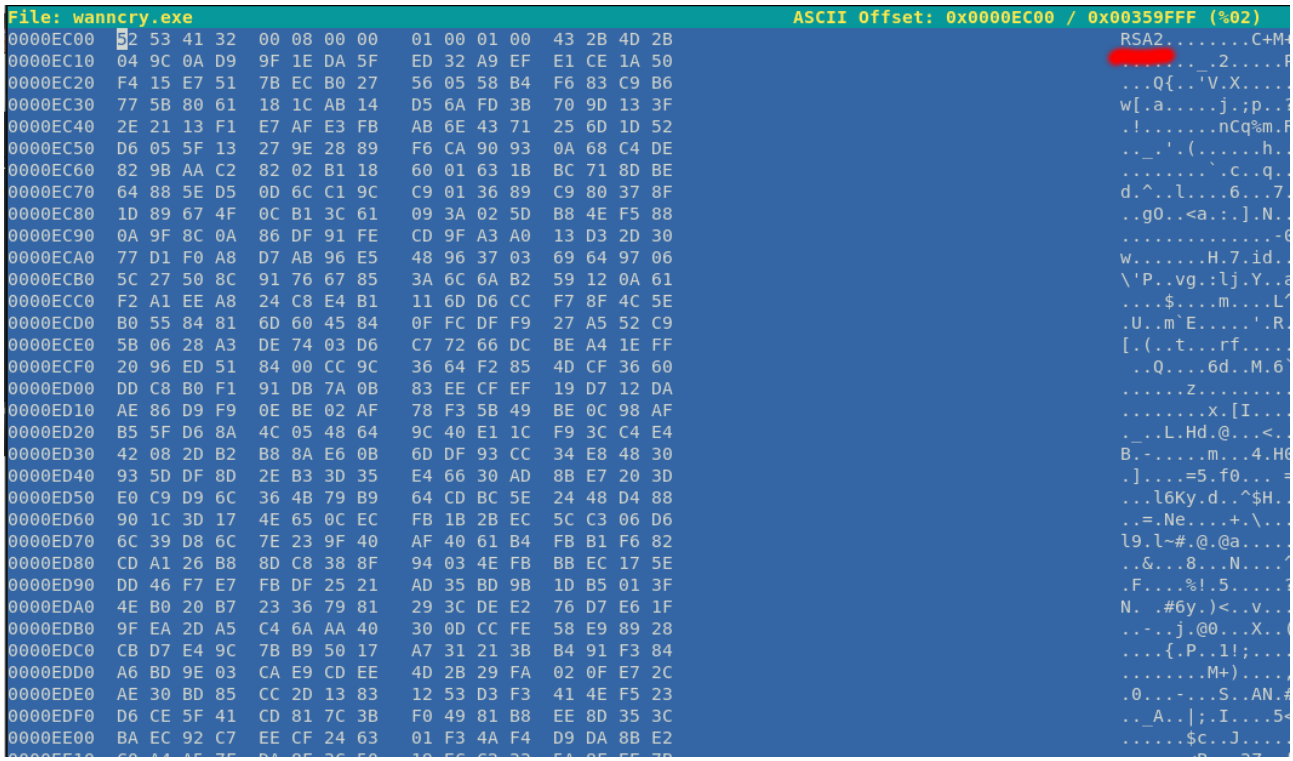


Fişierul 00000000.pky generat de virus care conţine cheia publică C_{pub}



Fişierul 00000000.eky generat de virus care conţine cheia privată C_{priv} criptată

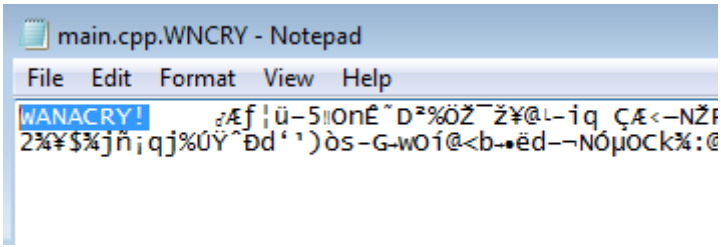
Dacă deschidem virusul cu un editor în hexa (de exemplu *hexeditor*) vom observa o cheie RSA declarată în executabil.



Formatul fişierelor criptate de acest virus este:

```
typedef struct _wc_file_t {
        char    sig[WC_SIG_LEN]
        uint32_t keylen;
        uint8_t  key[WC_ENCKEY_LEN];
        uint32_t unknown;
        uint64_t datalen;
        uint8_t *data;
} wc_file_t;
```

char sig[WC_SIG_LEN] - semnatura fişierelor criptate (WANACRY!)



uint32_t keylen - lungimea cheii de criptare

uint8_t key[WC_ENCKEY_LEN] - cheia AES criptată cu RSA

uint64_t datalen - lungimea fişierului înainte de a fi criptat

uint8_t *data - conţinutul fişierului criptat cu AES

În cazul în care aţi devenit victima unui ransomware vă sfătuiesc să NU plătiţi acea sumă de bani din două motive. Primul motiv este acela că astfel încurajaţi acest fenomen de creare a virusilor de acest tip, iar al doilea motiv fiind acela că NU aveţi nicio garanţie că veţi primi cheia de decriptare putând astfel să faceţi plata şi să rămâneţi cu fişierele criptate.

Cel mai bun scenariu ar fi acela în care cineva găseşte serverul pe care se află cheia privată S_{priv} şi face rost de această cheie, putând apoi decripta orice device ce a fost infectat. Din moment ce serverul pentru virusul WannaCry se află în Dark Web, putând fi accesat doar prin TOR, aflarea cheii de decriptare este aproape imposibilă. Totuşi, acest scenariu a fost întâlnit în cazul virusului *CryptoLocker* când o firmă de securitate a reuşit să găsească baza de date cu cheile private folosite de acest virus, făcând mai apoi un program care decripta fişierele.

Concluzii

Prin această lucrare de licentă am dorit să pun în evidentă importanţa antiviruşilor şi analizei malware în securitate. Am dorit ca metodele implementate în aplicaţie să fie cât mai apropiate de cele native, folosind astfel viruşi deja existenţi pentru testarea detecţiei.

În ceea ce priveşte analiza malware, aceasta rămâne un proces complex tocmai din cauza evoluţiei viruşilor. Atunci când o persoană realizează o analiză malware asupra unui executabil, aceasta trebuie să aibă în vedere că nu va putea înţelege toate detaliile dintr-un virus din moment ce nu are acces la codul acestuia. În cazul în care persoana are de a face cu secţiuni complicate dintr-un virus este mai bine ca acea persoană să încerce să îşi creeze o privire de ansamblu asupra acestor secţiuni în loc să fie prins în detalii.

De asemenea, în cazul analizei unui executabil nu există o *reţetă,* viruşii fiind de mai multe tipuri şi încadrându-se de cele mai multe ori în mai multe categorii simultan.

Antiviruşii şi viruşii sunt ca un joc de-a şoarecele şi pisica. În momentul în care analiştii de viruşi găsesc metode noi de a identifica şi de a analiza viruşii, creatorii de viruşi găsesc metode de a trece de aceste metode şi de a creşte complexitatea şi dificultatea analizei acestor viruşi.

Bibliografie

1. Michael Sikorski and Andrew Honig - no starch press (2012), *Practical Malware Analysis*

2. Michael Hale Ligh,Steven Adair,Blake Hartstein,Matthew Richard - Wiley Publishing (2011), *Malware Analyst's Cookbook*

3. Allen Harper, Shon Harris, Jonathan Ness, Chris Eagle, Gideon Lenkey, and Terron Williams – McGraw-Hill (2011), *Gray Hat Hacking The Ethical Hacker's Handbook Third Edition*

4. WannaCry Analysis - https://logrhythm.com/blog/a-technical-analysis-of-wannacry-ransomware/

5. PE File - https://resources.infosecinstitute.com/2-malware-researchers-handbook-demystifying-pe-file/

6. ELF file type – https://en.wikipedia.org/wiki/Executable_and_Linkable_Format

7. Elf32_Ehdr structure - http://www.openvirtualization.org/documentation/structElf32__Ehdr.html

8. How WannaCrypt Encrypts Your Files - https://www.youtube.com/watch?v=pLluFxHrc30

9. WannaCry Ransomware - https://www.secureworks.com/research/wcry-ransomware-analysis

10. WannaCry Malware Profile - https://www.fireeye.com/blog/threat-research/2017/05/wannacry-malware-profile.html

11. WannaCry -http://trapx.com/wpcontent/uploads/2017/08/Research_Paper_TrapX_WannaCry.pdf