Politechnika Warszawska Rok akademicki 2013/2014

Wydział Elektroniki i Technik Informacyjnych

Instytut Informatyki



Praca dyplomowa magisterska

Jakub Sejdak

**Metodologia zaburzania pracy silników antywirusowych**

Opiekun pracy:

dr inż. Krzysztof Cabaj

*Kierunek:* Informatyka

*Specjalność:* Inżynieria Systemów Informacyjnych

*Data urodzenia:* 8 sierpnia 1990 r.

*Data rozpoczęcia studiów:* 1 października 2009

**Życiorys**

Nazywam się Jakub Sejdak. Urodziłem się i mieszkam w Warszawie. Ukończyłem szkołę podstawową nr 209 im. Hanki Ordonówny. W trakcie nauki w gimnazjum nr 72 im. A.B. Dobrowolskiego uzyskałem tytuł laureata w olimpiadzie fizycznej, co pozwoliło mi na uczęszczanie do XIV LO im. Stanisława Staszica. W 2009 roku rozpocząłem studia na Wydziale EiTI Politechniki Warszawskiej. Zgodnie ze swoimi zainteresowaniami wybrałem specjalność Inżynieria Systemów Informacyjnych. W czasie studiów odbyłem praktyki   
w firmie Samsung Electronics R&D Center, a następnie podjąłem tam regularną pracę.

…………………………………………………….

podpis studenta

**Egzamin dyplomowy**

Złożył egzamin dyplomowy w dn.……………………………………………………….

Z wynikiem ……………………………...……..…..……………………………………………

Ogólny wynik studiów …...……………….…………………………………………………

Dodatkowe wnioski i uwagi Komisji ……………………………………………………

………………………………………………………………………………………………………………………………

**Streszczenie**

*Praca poświęcona jest opisowi metodologii zaburzania silników antywirusów   
z wykorzystaniem systemu malware-scrambler, rozwijanym w ramach pracowni dyplomowej magisterskiej. Celem było opracowanie sposobów automatycznego modyfikowania znanych i sklasyfikowanych wirusów komputerowych w sposób, aby ponowna analiza nie oznaczała ich jako niebezpieczne, przy jednoczesnym zachowaniu ich pierwotnych funkcjonalności. Realizowane jest to poprzez szereg zaprojektowanych modyfikacji plików wykonywalnych, określanych mianem silników mutujących. Zmodyfikowane wirusy były następnie ponownie skanowane tym samym zestawem skanerów antywirusowych, w celu potwierdzenia skuteczności mutacji.*

**Słowa kluczowe**: *Wirus komputerowy, Antywirus, Mutacja, Wykrywalność*

**Abstract**

**Title**: *Methodology of interfering with antivirus engines activity*

*This thesis describes methodology of interfering with antivirus engines activity, with help of malware-scrambler system, designed and implemented during graduation laboratory. Main goal of this work was to design methods of automatic modification of known and classified computer viruses, so they will not be marked as malicious anymore, without changing their original functionality. This is accomplished by a series of designed modifications of executable files, that are called mutation engines. After modifications, viruses were checked again by the same set of antivirus scanners to prove mutations’ effectiveness.*

**Key words**: *Computer virus, Antivirus, Mutation, Detectability*

Spis treści

[1. Wstęp 1](#_Toc398070060)

[2. Format plików Portable Executable 4](#_Toc398070061)

[2.1 Podstawowe pojęcia 4](#_Toc398070064)

[2.1.1 Przestrzeń adresowa procesu 5](#_Toc398070065)

[2.1.2 Względny adres wirtualny (RVA) 6](#_Toc398070066)

[2.1.3 Konwersja pozycji w pliku na RVA 7](#_Toc398070067)

[2.2 Model pamięci 8](#_Toc398070068)

[2.3 Podstawowe elementy formatu PE 10](#_Toc398070069)

[2.3.1 Nagłówek DOS 11](#_Toc398070070)

[2.3.2 Nagłówek PE 12](#_Toc398070071)

[2.3.2.1 Katalog danych 16](#_Toc398070072)

[2.3.3 Tabela sekcji 17](#_Toc398070073)

[2.3.4 Sekcje 19](#_Toc398070074)

[2.3.4.1 Sekcja kodu 19](#_Toc398070075)

[2.3.4.2 Sekcja danych 20](#_Toc398070076)

[2.3.4.3 Sekcja eksportu 20](#_Toc398070077)

[2.3.4.4 Sekcja importu 23](#_Toc398070078)

[2.3.4.5 Sekcja relokacji 26](#_Toc398070079)

[3. Metody detekcji wirusów 28](#_Toc398070080)

[3.1 Skanery pierwszej generacji 28](#_Toc398070082)

[3.1.1 Skanowanie łańcuchów 29](#_Toc398070083)

[3.1.2 Symbole wieloznaczne 29](#_Toc398070084)

[3.1.3 Metoda zakładek 30](#_Toc398070085)

[3.2 Skanery drugiej generacji 30](#_Toc398070086)

[3.2.1 Skanowanie inteligentne 30](#_Toc398070087)

[3.2.2 Dopasowanie szkieletu 32](#_Toc398070088)

[3.3 Metody statycznego wykrywania deszyfratora 33](#_Toc398070089)

[3.4 Analiza behawioralna 34](#_Toc398070090)

[3.4.1 Dynamiczna detekcja deszyfratorów 35](#_Toc398070091)

[3.5 Analizy heurystyczne 35](#_Toc398070092)

[4. Projekt malware-scrambler 37](#_Toc398070093)

[4.1 Wymagania systemu 37](#_Toc398070094)

[4.2 Cykl życia modyfikowanych plików 38](#_Toc398070095)

[4.3 Potok mutacyjny 39](#_Toc398070096)

[4.4 Silniki mutacyjne 41](#_Toc398070097)

[4.4.1 Silniki pomocnicze 41](#_Toc398070098)

[4.4.1.1 Zmiana rozmiaru i położenia sekcji 41](#_Toc398070099)

[4.4.1.2 Przenoszenie danych w obszarze pliku 43](#_Toc398070100)

[4.4.1.3 Generacja kodu wykonywalnego 44](#_Toc398070101)

[4.4.2 Szyfrowanie 47](#_Toc398070102)

[4.4.2.1 Deszyfrator doklejany 48](#_Toc398070103)

[4.4.2.2 Deszyfrator nadpisujący 50](#_Toc398070104)

[4.4.3 Wstawianie martwego kodu 53](#_Toc398070105)

[4.4.4 Usuwanie tablicy IAT 55](#_Toc398070106)

[4.5 Weryfikacja poprawności działania 59](#_Toc398070107)

[5. Wyniki eksperymentów 61](#_Toc398070108)

[6. Podsumowanie 70](#_Toc398070109)

[6.1.1 Ograniczenia i limitacje programu 70](#_Toc398070110)

[6.1.2 Perspektywy dalszego rozwoju 72](#_Toc398070111)

[7. Bibliografia 74](#_Toc398070112)

[8. Spis obrazków 77](#_Toc398070113)

[9. Spis tabel 78](#_Toc398070114)

[10. Załączniki 79](#_Toc398070115)

# Wstęp

Korzystanie z rozwiązań informatycznych od zawsze wiązało się z ryzykiem naruszenia bezpieczeństwa systemu komputerowego lub danych użytkownika. Wraz z rozwojem technologicznym i powstaniem wielu usług sieciowych, wzrosło także zagrożenie ze strony osób trzecich, które przy pomocy złośliwego oprogramowania (ang. malicious software, malware) próbują przejąć dostęp do systemu użytkownika lub znajdujących się w nim danych. W odpowiedzi na coraz większe zagrożenie, Międzynarodowa Organizacja Normalizacyjna ISO (ang. International Organization for Standarization) wprowadziła standard ISO 7498-2 [1], który definiuje 5 usług bezpieczeństwa, które mogą być celem potencjalnego ataku i wymagają odpowiedniego zabezpieczenia. Są to: uwierzytelnienie, kontrola dostępu, poufność danych, integralność danych oraz niezaprzeczalność. Często nieformalnie definiuje się również usługę dostępu, gdyż stanowi ona ważny fundament działania większości systemów globalnych.

Nadal popularnymi narzędziami ataku są wirusy komputerowe, konie trojańskie oraz robaki internetowe. Większość z nich występuje w formie plików wykonywalnych, które mogą zostać załadowane i uruchomione przez system operacyjny. Z uwagi na to, iż pod względem budowy nie różnią się od zwykłych programów, trudno jest jednoznacznie określić, czy dany plik zawiera   
w rzeczywistości złośliwy kod czy nie.

Kolejnym zagrożeniem jest coraz częstsze wykorzystywanie przez atakujących tzw. socjotechniki (ang. social engineering), czyli zestawu metod służących do uzyskania niejawnych informacji lub sugestii wykonania przez drugiego człowieka pewnej akcji. Fundamentem jest w tym przypadku łatwowierność ludzi, naturalna chęć pomocy obcym oraz strach w obliczu zagrożenia. Kevin Mitnick w swojej książce [2] opisuje w jaki sposób przy pomocy metod socjotechnicznych wyłudzał od pracowników firm hasła i tajne informacje bez fizycznego dostępu do komputerów. Metody te są obecnie bardzo często stosowanie w postaci tzw. phishingu, czyli podszywania się pod pewną usługę lub podmiot. Przykładem mogą być hiperłącza (ang. links) w poczcie elektronicznej, które sugerują odniesienie do innej treści niż w rzeczywistości. Twórcy malware-u stosują tą metodę do rozprzestrzeniania zainfekowanych programów na komputery nieświadomych użytkowników.

Równolegle z rozwojem cyberprzestępczości opracowywane były nowoczesne metody ochrony zasobów komputera. Wiele lat pracy naukowców i inżynierów zaowocowało powstaniem dedykowanych narzędzi jak zapora ogniowa (ang. firewall) oraz skanery antywirusowe (potocznie zwane antywirusami), które monitorują i w pewnym stopniu niwelują zagrożenia. Obecnie jest wiele komercyjnych i bezpłatnych rozwiązań dla użytkowników prywatnych   
i biznesowych mających na celu ochronę zasobów komputerowych. Wiele dużych firm zajmujących się bezpieczeństwem informacji w informatyce (ang. IT security) posiada specjalne centra i laboratoria, gdzie odbywają się ciągłe inspekcje w celu jak najszybszego zidentyfikowania zagrożenia i opublikowaniu pakietów uodparniających ich produkty.

Mimo tak dużego zaangażowania w poprawę bezpieczeństwa systemów informatycznych, istnieje wiele czynników, które sprawiają, że nawet komputery wyposażone w programy antywirusowe i zapory ogniowe są wciąż narażone na atak. Najczęstszym powodem jest brak świadomości użytkowników na temat istniejących zagrożeń lub ich jawne bagatelizowanie. Często użytkownicy twierdzą, iż aspekty bezpieczeństwa ich nie dotyczą. Drugim kluczowym elementem jest trudność w rozpoznawaniu zagrożeń w czasie rzeczywistym. Mimo iż obecne pakiety antywirusowe dysponują bazą danych znanych postaci malware-u   
i odpowiednimi technikami porównywania, to wciąż zazwyczaj zawodzą  
w przypadku analizy nieskatalogowanego jeszcze złośliwego oprogramowania.

Większość wiedzy na temat działania skanerów antywirusowych nie jest podawana do wiadomości publicznej, gdyż jest to tajemnica handlowa. Kwestii badania sposobów wykrywania wirusów komputerowych poświęcono wiele prac, np. [3], [4], [5], [6], [7]. Mimo tego badacze wciąż nie wiedzą jakich rozwiązań używają producenci konkretnego rozwiązania antywirusów w implementacji swoich silników skanujących. Inne pytanie jakie można zadać, to ile ze znanych algorytmów i technik wykrywania malware-u jest faktycznie wykorzystywanych. Kwestią rozstrzygającą może być tu czas lub narzut obciążenia na procesor   
i pamięć operacyjną podczas skanowania plików i procesów. Pytania te są ostatnimi czasy coraz istotniejsze w związku z pojawiającymi się głosami iż programy AV są przeżytkiem, głównie ze względu na niską wykrywalność zagrożeń. Przykładem takiej opinii może być wypowiedź Briana Dye’a, wice dyrektora firmy Symantec, w której określa on programy antywirusowe jako martwe [8].

Celem tej pracy jest próba zweryfikowania twierdzenia dotyczącego skuteczności programów antywirusowych z wykorzystaniem metody eksperymentalnej. Dodatkowo, podjęta została próba udowodnienia, iż względnie nieskomplikowanymi metodami przy ograniczonych funduszach, czasie i zasobach możliwe jest ominięcie komercyjnych form zabezpieczeń systemów komputerowych. Wiedza ta może pozwolić na próbę ocenę skuteczności działania programów antywirusowych i pomóc w decyzji co do wyboru najlepszego rozwiązania. Jest to problem zyskujący coraz większą wagę, w związku   
z nasilaniem się tzw. ataków spersonalizowanych. Atakujący wybiera ofiarę   
i dobiera odpowiedni sposób ataku bazując na wiedzy o używanej przez nią oprogramowaniu. Przykładem może być szpiegostwo korporacyjne lub działania obcych wywiadów. Badanie polegało na opracowaniu metod automatycznej modyfikacji malware-u w postaci plików wykonywanych i zbadanie wpływu tych modyfikacji na wykrywalność. W eksperymencie zostanie wykorzystanych kilkadziesiąt popularnych programów antywirusowych, które będą skanowały te same pliki. Duża liczba branych pod uwagę programów ma zapewnić obiektywny osąd działania skanerów. Ważne jest, aby zwrócić uwagę zarówno na błędy niewykrycia złośliwego kodu (ang. false negative), ale również błędy uznania prawidłowych aplikacji za podejrzane (ang. false positive). Oba rodzaje nieprawidłowego wyniku skanowania wpływają na jakość danego produktu oraz pozwalają wnioskować na temat tego w jaki sposób działa.

Praca została podzielona na trzy główne części. Pierwsza obejmuje rozdział drugi, który opisuje szczegółowo format plików wykonywalnych na platformie Windows. Druga część opisuje stan dostępnej wiedzy o sposobach skanowania używanych przez programy antywirusowe. Tematykę tę obejmuje rozdział trzeci. Na koniec, w rozdziale czwartym, przedstawiony zostanie projekt malware-scrambler oraz omówione zostaną zaproponowane modyfikacje plików wykonywalnych. W rozdziale piątym przedstawione zostaną wyniki uzyskane podczas eksperymentów.

# Format plików Portable Executable

Pliki wykonywalne, jak inne dane komputerowe mają ściśle określoną budowę. W związku z różnicami związanymi z systemem operacyjnym oraz architekturą używanego procesora mają one zawsze na danej platformie ustalony i inny między platformami format. W przypadku systemu Unix jest to aktualnie typ ELF, a na platformie Windows jest to aktualnie typ PE. Z uwagi na to, iż cała praca poświęcona jest wyłącznie wirusom napisanym na platformę Windows, format ELF nie będzie omawiany.

Pliki wykonywalne formatu PE (ang. Portable Executable) są stosowane   
w systemach Microsoft Windows od wersji NT 3.1. Istotą zdefiniowania jednolitego formatu jest ułatwienie implementacji i późniejszej pracy programu, który odpowiada za załadowanie i uruchomienie innych aplikacji. Każdy plik, który jest w jakikolwiek sposób niezgodny z przyjętymi wytycznymi, nie może być uruchomiony na danej platformie. Zazwyczaj za utworzenie poprawnego pliku tego formatu odpowiada linker, uruchamiany w trakcie budowania aplikacji przez dewelopera. Istnieją nieznaczne różnice w plikach wynikowych utworzonych przez linkery różnych producentów, jednak są zbyt małe, aby je szczegółowo opisywać oraz w żaden sposób nie wpływają na skanery plików. Należy jednak mieć na uwadze taką możliwość, aby nie czynić żadnych założeń co do formatu   
w miejscach, gdzie nie jest to zdefiniowane. Przykładem mogą być nazwy sekcji kodu, tworzone przez kompilator firmy Borland i Microsoft. W pierwszym przypadku nadawana nazwa to „CODE”, a w drugim to „.text”. Jednak nazwa sekcji jest dodawana wyłącznie dla ułatwienie człowiekowi późniejszej analizy   
i debugowania.



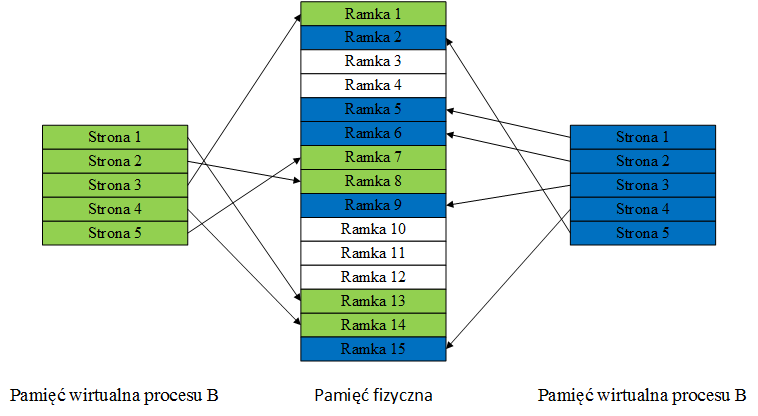
## Podstawowe pojęcia

Zrozumienie budowy i celowości wprowadzenia niektórych pól w formacie PE jest kluczowe dla dalszych rozważań na temat wirusów komputerowych oraz systemach przed nimi zabezpieczających. W poniższych podrozdziałach omówione zostaną poszczególne struktury, zależności między nimi i wpływ na akcje podejmowane przez program ładujący przy uruchamianiu.

### Przestrzeń adresowa procesu

Każdy działający program w systemie Windows nazywany jest procesem.   
W jego skład wchodzi kod i dane znajdujące się w pliku wykonywalnym przed załadowaniem, ale również wszystkie zasoby, które zostaną mu przydzielone podczas uruchamiania i dalszej pracy. Najważniejszym z zasobów jest pamięć operacyjna, do której program jest ładowany.

W celu ochrony dostępu do pamięci procesu przed innymi procesami, wprowadzony został mechanizm pamięci wirtualnej, który opiera się na separacji przestrzeni adresowej każdego z programów. Podczas uruchomienia każdy nowy proces otrzymuje fragment pamięci fizycznej, będącej zazwyczaj wielokrotnością pewnej ustalonej wielkości, tzw. strony pamięci. Jednak nie muszą to być kolejne sekwencyjne ramki fizyczne, gdyż te mogą być już wykorzystywane. W celu ukrycia tego faktu przed wyższymi warstwami, każda uruchamiana aplikacja otrzymuje pewną abstrakcję pamięci fizycznej, w postaci pamięci wirtualnej (Rysunek 1).



Rysunek 1. Separacja przestrzeni adresowych poprzez pamięć wirtualną

Logicznie jest to pełna przestrzeń adresowa (tzn. taka którą da się zaadresować), która w systemach 32-bitowych (architektura x86) ma rozmiar 4 GB. Dzięki temu mechanizmowi procesy nie muszą być świadome ilości faktycznie przydzielonej pamięci RAM, gdyż ta będzie dołączana na żądanie. Możliwe jest też jednolite adresowanie z wykorzystaniem tzw. adresów wirtualnych. W architekturze x86 taka przestrzeń jest odwzorowana adresami wirtualnymi (ang. VA – Virtual Address) z zakresu 0x00000000 – 0x7FFFFFFF. W przypadku odwołania do adresu pamięci, któremu została przez program zaalokowana strona pamięci (odpowiednik ramki pamięci w przestrzeni wirtualnej), a któremu nie odpowiada jeszcze żadna ramka fizyczna (tzw. chybienie strony) następuje automatyczny przydział i sprowadzenie jej do przestrzeni procesu. Strategia przydziału stron pamięci jest zależna od systemu operacyjnego i nie będzie w tej pracy omawiana.

### Względny adres wirtualny (RVA)

Format pliku Portable Executable wymaga w wielu miejscach podania adresu wirtualnego. Przykładem mogą być adresy, pod którymi znajdować się będą sekcje kodu, danych lub inne. Jednak podawanie wprost adresu wirtualnego, czyli takiego do którego faktycznie będzie odwołanie w przestrzeni procesu, powoduje szereg problemów. Najważniejszy z nich to możliwość załadowania programu pod inny adres (tzw. adres bazowy, z ang. base address) niż przewidują to dane   
w pliku. W takim przypadku program ładujący (ang. loader) musiałby przy uruchamianiu wszystkie te dane nadpisać adresami faktycznymi. Jest to zadanie możliwe, jednak znacząco opóźniające czas startu aplikacji.

W celu eliminacji tego problemu wprowadzone zostało pojęcie adresu wirtualnego względem adresu bazowego, czyli takiego pod który został załadowany cały program. Względny adres wirtualny często skraca się do akronimu RVA, od angielskiej nazwy Relative Virtual Address. Dzięki temu mechanizmowi, określenie przesunięcia zamiast adresu zawsze umożliwia obliczenie prawidłowego adresu wirtualnego niezależnie od adresu bazowego. Zależność pomiędzy adresem wirtualnym, a RVA przedstawia poniższe równanie.

Przykładowo, jeżeli w nagłówku sekcji kodu pole odpowiadające RVA tej sekcji ma wartość 0x1000, a plik (w terminologii Microsoftu nazywany modułem [9]) zostanie załadowany pod adresem 0x400000, to adres wirtualny sekcji kodu będzie równy 0x401000.

### Konwersja pozycji w pliku na RVA

Każda sekcja zdefiniowana w danym pliku wykonywalnym jest opisana przez nagłówek sekcji. Znajdują się w nim informacje takie jak nazwa sekcji, RVA sekcji oraz przesunięcie względem początku pliku, pod którym można znaleźć pierwszy bajt tej sekcji. Z uwagi na to, iż każdą sekcję opisuje zarówno parametr dotyczący pamięci wirtualnej, jak i pozycji z w pliku, jest możliwe określenie zależności pomiędzy pozycją w pliku, a RVA.

W przypadku konwersji ze względnego adresu wirtualnego na pozycję   
w pliku, należy najpierw ustalić w skład której sekcji wchodzi dany adres względny. Można tego dokonać poprzez iteracyjne sprawdzanie, czy adres jest w zakresie zdefiniowanym przez RVA początku sekcji oraz koniec równy RVA początku   
z dodanym rozmiarem dla każdej kolejnej sekcji. Gdy znana jest już sekcja, należy obliczyć przesunięcie danego RVA względem RVA jej początku. Następnie należy dodać tą wartość do parametru opisującego pozycję w pliku tej sekcji, a wynikiem będzie szukany odpowiednik RVA jako przesunięcie pliku.

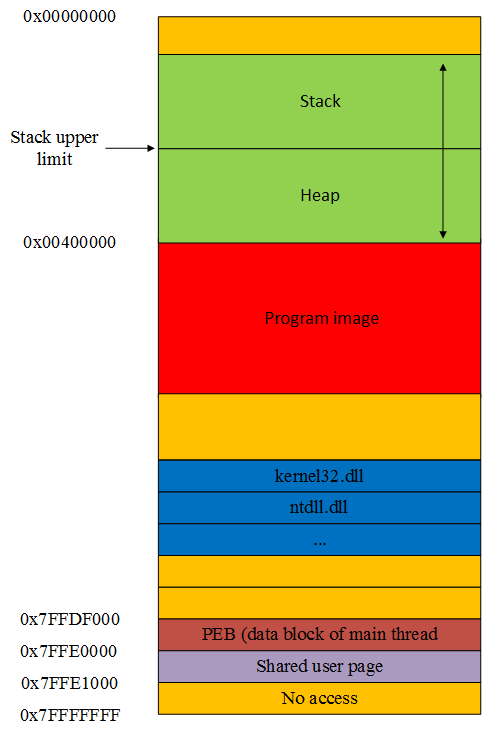
Konwersji odwrotnej dokonuje się analogicznie, szukając najpierw sekcji na podstawie przesunięcia w pliku, a następnie dodając do RVA jej początku różnicę pomiędzy konwertowanym przesunięciem w pliku, a pozycją początku tej sekcji   
w pliku.

Umiejętność dokonywania zamiany RVA na pozycję w pliku i odwrotnie jest bardzo ważna w kontekście rozważania działania i modyfikacji wirusów komputerowych oraz programu ładującego. W dalszej części pracy zostanie poczynione założenie, że jeżeli znany jest względny adres wirtualny znajdujący się wewnątrz jakiejś sekcji, to znana jest również pozycja w pliku i odwrotnie.

Poważnym problemem pozostaje sytuacja, gdy dany adres wirtualny lub przesunięcie w pliku nie odpowiada żadnej ze zdefiniowanych sekcji. Podczas przeprowadzania badań analizowano pliki posiadające takie adresy. Niestety nie udało się wyjaśnić tej sytuacji korzystając z literatury, jednak stwierdzono empirycznie, że jeżeli jakieś dane mają RVA mniejszy niż RVA pierwszej sekcji, to liczbowo jest to równe pozycji w pliku. Przykładowo, jeżeli adres względny pierwszej sekcji jest równy 0x1000, a pewna dana ma RVA równe 0x270, to   
z dużym prawdopodobieństwem w pliku dana ta znajduje się na pozycji 0x270 względem początku pliku. Należy mieć na uwadze, iż ze względu na niemożliwość potwierdzenia tego w oficjalnej dokumentacji firmy Microsoft, założenie to może być nieprawdziwe.

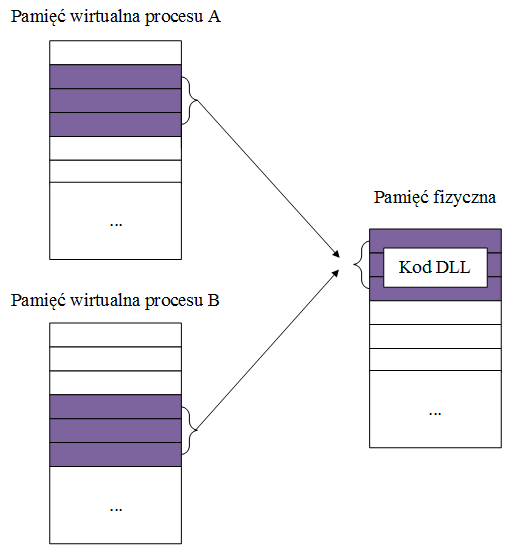
## Model pamięci

System Windows przy uruchamianiu każdej aplikacji tworzy dla niej nowy proces oraz inicjalizuje jego kontekst. Podstawowym zasobem każdego procesu jest jego przestrzeń adresowa. Domyślnie w większości wersji systemu Windows   
z serii NT przestrzeń ta jest podzielona na dwa regiony (ang. chunks) zajmujące po 2 GB: przestrzeń użytkownika mieszcząca się w przedziale adresów 0x00000000 – 0x7FFFFFFF oraz przestrzeń jądra obejmująca adresy z zakresu 0x80000000 – 0xFFFFFFFF [10], [11]. Według oficjalnej dokumentacji systemu Windows [12], podział ten można nieznacznie zmodyfikować ustawiając w opcjach systemowego programu ładującego (ang. bootloader) flagę uaktywniającą tryb 4GT (4-Gigabyte Tuning), który zmienia tą proporcję, rezerwując aż 3 GB na obszar pamięci użytkownika i 1 GB na obszar pamięci jądra.

Program ładujący podczas fazy ładowania programu do pamięci odwzorowuje dane znajdujące się w pliku wykonywalnym na odpowiednie struktury w pamięci wirtualnej. Dodatkowo, każdy proces do poprawnego działania musi mieć dostęp (z poziomu pamięci operacyjnej) do pewnych ustalonych struktur systemowych, jak np. bibliotek dynamicznych lub struktury PEB (ang. Process Environment Block). Typowy podział pamięci procesu na platformie Windows odzwierciedla Rysunek 2.

Rysunek 2. Mapa pamięci procesu w systemie Windows

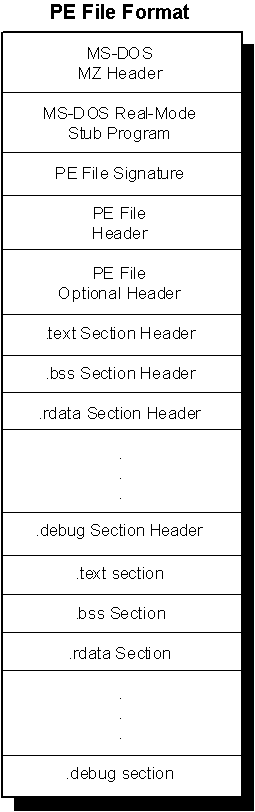
Kolorem czerwonym zaznaczony został obraz modułu, który jest odczytywany i ładowany z pliku przez loader systemowy. Zazwyczaj preferowanym adresem bazowym każdego modułu jest adres o wartości 0x400000. Jest to jednak tylko sugestia ze strony aplikacji i nie musi być wzięta pod uwagę. Wpływ na to może mieć zajętość danego fragmentu przestrzeni adresowej przez inny moduł wykorzystywany przez dany proces. Jeżeli dochodzi do takiej sytuacji, to program ładujący dokonuje tzw. relokacji pamięci, podczas której kod programu ładowany jest w inne miejsce, a wszystkie adresy, które opierały się wewnątrz aplikacji   
o preferowany adres bazowy, zostają przeliczone i nadpisane nowymi, poprawnymi wartościami.

Rysunek 2 wskazuje także, że w pamięci procesu odwzorowane są także wszystkie biblioteki dynamicznie współdzielone (ang. DLL – Dynamically Linked Library), takie jak kernel32.dll czy ntdll.dll. Są one potrzebne do prawidłowego działania programu na danej platformie, gdyż posiadają definicje najbardziej podstawowych funkcji systemowych jak LoadLibrary czy HeapAlloc. Mechanizm zarządzania pamięcią przewiduje, iż ze względu na to, że każdy proces musi mieć do nich dostęp, to optymalniej jest, gdy biblioteki DLL są fizycznie załadowane do pamięci operacyjnej tylko raz, a procesy mają odpowiednio skonfigurowane mapowanie pamięci wirtualnej na fizyczną, aby wskazywały na ten sam fragment pamięci fizycznej. Sytuację tę ilustruje Rysunek 3.

Rysunek 3. Odwzorowanie bibliotek współdzielonych w przestrzeni procesów

## Podstawowe elementy formatu PE

W tym podrozdziale opisane zostaną podstawowe elementy formatu plików wykonywalnych PE, które mają wpływ na to jak i gdzie program zostanie załadowany w pamięci oraz w jaki sposób będzie następowało jego wykonywanie. Ogólny zarys struktury pliku PE przedstawia Rysunek 4.

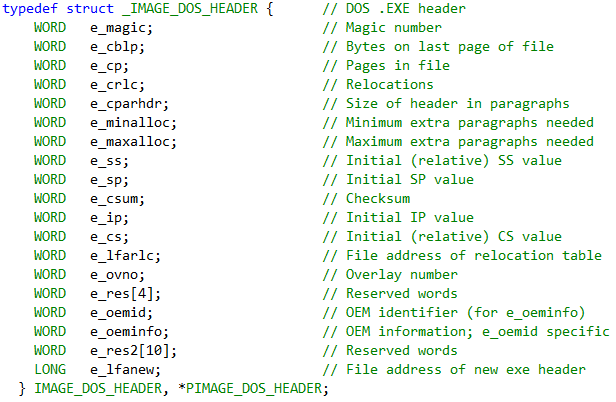
Dla wszystkich opisywanych struktur danych i zależności między nimi, główną referencją powinien być zawsze plik winnt.h dostępny jako standardowy plik nagłówkowy w każdym systemie Windows NT.

Rysunek 4. Struktura pliku Portable Executable ( [23] [24])

### Nagłówek DOS

Każdy plik w formacie Portable Executable rozpoczyna się nagłówkiem DOS, który zajmuje pierwsze 64 bajty pliku. Mimo, iż jego obecność związana jest głównie z kompatybilnością wsteczną z systemem MS-DOS, to jest obowiązkowym elementem każdego pliku wykonywalnego. Jego definicję (zaczerpniętą wprost   
z pliku winnt.h) przestawia Rysunek 5.

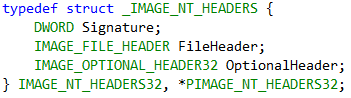
Rysunek 5. Definicja nagłówka DOS



Nagłówek DOS zdefiniowany jest w systemie Windows jako struktura o nazwie IMAGE\_DOS\_HEADER. Pierwsze dwa bajty to tzw. sygnatura nagłówka DOS   
i równa jest znakom „MZ”, pochodzącym od inicjałów jednego z architektów systemu MS-DOS, Marka Zbikowsky’iego. Jej znaczenie logiczne ogranicza się tylko do sprawdzenia, czy dany plik może być uruchomiony w tym systemie. Tuż za IMAGE\_DOS\_HEADER znajduje się niewielki fragment kodu (zwany często „DOS stub”), którego jedynym zadaniem jest powiadomienie użytkownika, iż program nie jest w pełni zgodny z system DOS i nie może być dalej wykonywany. Domyślnie na standardowe wyjście wypisywany jest komunikat „Ten program nie może być uruchomiony w trybie DOS”. Tekst ten jest zdefiniowany w pliku WINSTUB.EXE,   
z którego korzysta linker podczas budowania aplikacji. Możliwe jest zdefiniowanie własnej wersji komunikatu po zastosowaniu opcji –STUB w fazie konsolidacji.

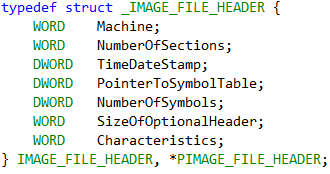
Drugim istotnym elementem nagłówka DOS jest pole e\_lfanew, które określa pozycję w pliku, pod którą znajduje się główny nagłówek PE. Przy analizie plików Portable Executable zawsze powinno się odwoływać do nagłówka PE tylko poprzez wspomnianą pozycję, ponieważ założenie, iż znajduje się on tuż za fragmentem kodu przeznaczonego dla systemu DOS jest w większości przypadków niepoprawne.

### Nagłówek PE

Jedną z największych i zarazem najważniejszych struktur w pliku wykonywalnym zgodnym z formatem PE jest nagłówek PE, który jest zdefiniowany poprzez strukturę IMAGE\_NT\_HEADERS. Jej budowa przedstawiona jest na Rysunku 6.

Rysunek 6. Struktura nagłówka PE

Nagłówek składa się z trzech pól. Pierwsze z nich to sygnatura PE, która złożona jest z 4 znaków: „PE00”. Częstym błędem jest ignorowanie dwóch dodatkowych zer w sygnaturze i sprawdzanie tylko tych pierwszych. Warto mieć na uwadze, iż program ładujący korzysta ze specjalnego makra zdefiniowanego   
w pliku winnt.h, aby porównać sygnaturę z wartością wzorcową, która ma dokładnie 4 podane wyżej znaki. Jeżeli którykolwiek znak jest inny, program uznawany jest za uszkodzony lub niezgodny z formatem PE i nie jest dalej przetwarzany.

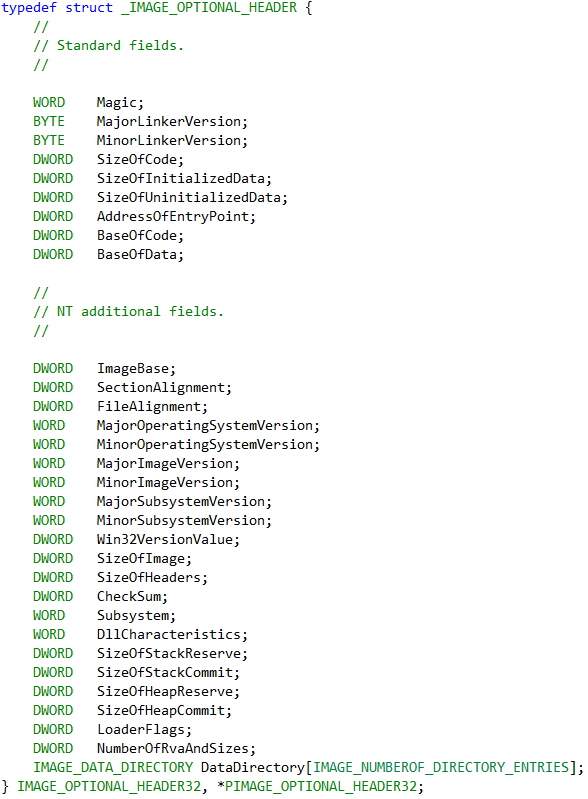
 Drugim polem, na które warto zwrócić uwagę, jest struktura IMAGE\_FILE\_HEADER, która określa ogólne właściwości pliku. Wśród nich znajdują się trzy, które są najbardziej interesujące z punktu twórców wirusów komputerowych: NumberOfSections, Characteristics i SizeOfOptionalHeader. Pełny wygląd nagłówka pliku obrazuje Rysunek 7.

Rysunek 7. Nagłówek pliku

Pole NumberOfSections określa liczbę sekcji, która znajduje się w pliku. Jest to bardzo ważna informacja dla programów analizujących pliki wykonywalne jak i dla samego loadera, gdyż tylko na tej podstawie są w stanie odszukać wszystkie nagłówki sekcji, a w konsekwencji adresy poszczególnych sekcji pliku. Natomiast pole Characteristics jest zbiorczą zmienną, która przechowuje różne właściwości pliku w postaci flag połączonych ze sobą operacją OR. Jedną z dwóch ważnych własności, które może to pole reprezentować, to fakt czy dany plik jest   
w istocie plikiem wykonywalnym aplikacji (najczęściej z rozszerzeniem exe), czy plikiem biblioteki ładowanej dynamicznie (ang. Dynamic-link Library, DLL). Powodem wprowadzenia tej flagi jest umożliwienie rozróżnienie czy dany plik zawiera pełną aplikację która może być bezpośrednio załadowana i uruchomiona czy jest to plik zawierający kod biblioteki dynamicznej. W systemach rodziny Windows zarówno biblioteki współdzielone jak i zwykłe aplikacje korzystają z tego samego formatu plików PE. Druga istotna flaga to opcja wymuszająca na programie ładującym umieszczenie danego modułu w pamięci operacyjnej wyłącznie pod preferowanym adresem bazowym. W przypadku, gdyby dane miejsce nie było dostępne loader powinien zgłosić błąd. Ostatnie z wymienionych pól, SizeOfOptionalHeader, jest niczym innym jak rozmiarem kolejnego nagłówka, który znajduje się tuż za IMAGE\_FILE\_HEADER. Pole to jest potrzebne, gdyż   
w zależności od tego czy plik PE jest 32-bitowy (PE32), czy 64-bitowy (PE32+) to rozmiar następnego nagłówka jest inny.

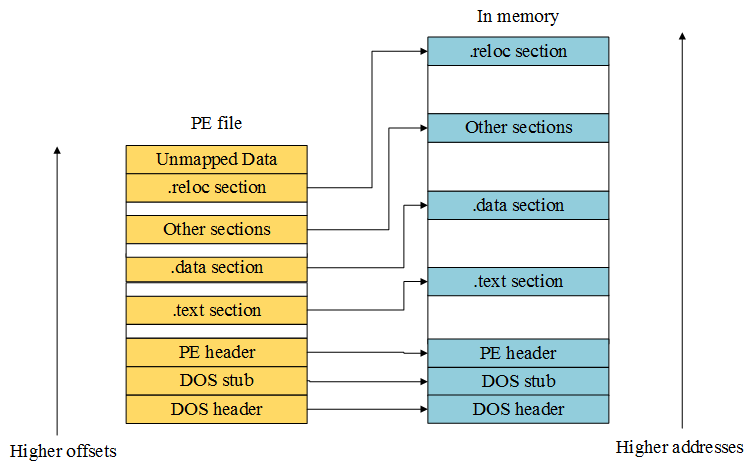
OptionalHeader to struktura, której nazwa sugeruje jakoby była mało istotna, a wręcz opcjonalna. W rzeczywistości przechowuje ona kluczowe informacje, na podstawie których program ładujący systemu Windows rozmieszcza poszczególne elementy pliku w pamięci wirtualnej. Jej elementy składowe prezentuje szczegółowo Rysunek 8.

Analizując strukturę IMAGE\_OPTIONAL\_HEADER można zauważyć, iż przechowuje ona dwa wyróżniające się rodzaje danych: zmienne związane   
z wersjonowaniem oraz zmienne związanie z adresami i rozmiarami innych fragmentów pliku. Spotyka się określenie, iż IMAGE\_OPTIONAL\_HEADER jest swojego rodzaju mapą całego pliku, gdyż przechowuje ona przesunięcia wszystkich innych niestandardowych elementów formatu PE, które nie muszą występować w każdym pliku.

Pierwszym i z punktu widzenia wykonywania kodu najważniejszym polem tej struktury jest AddressOfEntryPoint. Zawiera on RVA miejsca w sekcji kodu, gdzie rozpoczynają się instrukcje użytkownika (często spotykane w programach debugujących w angielskiej postaci „user code”). Jest to podstawowe pole, które może być modyfikowane przez wirusy komputerowe, gdyż odpowiada ono bezpośrednio za sterowanie przepływem wykonania kodu. Program ładujący nie czyni żadnych wstępnych analiz względem miejsca na które wskazuje ta zmienna. Jedynym logicznym wymaganiem jest, aby sekcja do której następuje skok miała prawa do wykonywania kodu. Innymi polami, które trzeba brać pod uwagę przy modyfikacjach sekcji pliku wykonywalnego są BaseOfCode oraz BaseOfData. Oba pola przechowują wirtualny adres względny odpowiednio sekcji kodu oraz sekcji danych.

Rysunek 8. IMAGE\_OPTIONAL\_HEADER

Kolejne trzy zdefiniowane pola odgrywają ważną rolę w procesie rozmieszczania kodu sekcji w pliku i w pamięci wirtualnej. Składowe pole ImageBase przechowuje wspominany w poprzednich podrozdziałach adres bazowy, czyli preferowany adres pod który program ładujący powinien załadować dany moduł. Jest to adres wirtualny, do którego odnoszą się wszystkie adresy względne danego modułu. Zmienna SectionAlignment określa do jakiej wartości powinny być wyrównane adresy początkowe sekcji w pamięci wirtualnej.   
W praktyce sprowadza się to liczby, której wielokrotnością musi być adres początkowy. Bardzo często spotykaną wartością jest w tym przypadku 0x1000.   
Z kolei FileAlignment określa podobną właściwość sekcji, tym razem dotyczącą rozmieszczenia w pliku na dysku. Typową wartością jest 0x200. Warto zauważyć, iż oba te pola zazwyczaj nie są sobie równe. Wynika to z tego, że wyrównanie sekcji w pamięci operacyjnej z powodów wydajnościowych oraz polityki zarządzania pamięcią powinno być określone przez domyślny rozmiar strony pamięci w danym systemie. Zazwyczaj dla platformy x86 jest ona równa 4 KB, co w zapisie heksadecymalnym równe jest 0x1000. Z kolei wyrównanie w pliku, wg oficjalnej dokumentacji firmy Microsoft [13] powinno być potęgą liczby   
2 z przedziału 512 – 65536. Ze względu na oszczędność miejsca wybiera się zazwyczaj wartość 512, równą w systemie szesnastkowym wartości 0x200. Ilustrację różnicy pomiędzy wyrównaniem w pliku na dysku a modułu w pamięci pokazuje Rysunek 9.



Rysunek 9. Porównanie wyrównania sekcji w pliku i w pamięci

Warto zwrócić uwagę na pole Checksum, które wbrew naturalnym oczekiwaniom nie określa sumy kontrolnej w każdym przypadku. Jest ono wykorzystywane tylko w przypadkach, gdy dany plik wykonywalny lub biblioteka DLL ma bezpośredni dostęp do krytycznych miejsc systemu operacyjnego.   
W szczególności pole to sprawdzane jest dla wszystkich sterowników oraz bibliotek współdzielonych, które są ładowane podczas startu systemu. Algorytm obliczający sumę kontrolną jest w zdefiniowany w bibliotece imagelp.dll. We wszystkich innych przypadkach pole to jest wyzerowane.

#### Katalog danych

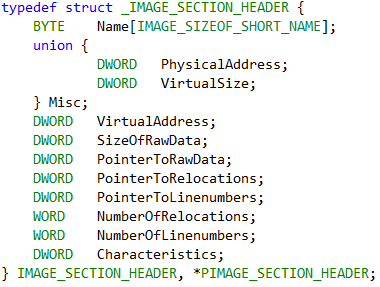
Ostatnim, bardzo ważnym polem struktury IMAGE\_OPTIONAL\_HEADER jest tablica DataDirectory (nazywana czasem katalogiem), która posiada dokładnie 16 elementów typu IMAGE\_DATA\_DIRECTORY. Zgodnie z dokumentacją formatu Portable Executable, każdy taki element zawiera względny adres wirtualny oraz rozmiar tablicy lub łańcucha znakowego, z którego korzysta system Windows. Jest to jednak bardzo ogólna definicja i wymagająca doprecyzowania. Każdy indeks tablicy DataDirectory związany jest z konkretnym fragmentem pliku, którego przeznaczenie jest precyzyjnie zdefiniowane:

1. tablica eksportu
2. tablica importu
3. tablica zasobów
4. tablica wyjątków
5. tablica certyfikatów
6. tablica relokacji bazowych
7. dane debugowe
8. indeks zarezerwowany (musi być wyzerowany)
9. globalny wskaźnik
10. tablica TLS (ang. Thread Local Storage)
11. tablica konfiguracji ładowania
12. tablica bindowania
13. tablica importu adresów (IAT)
14. deskryptor opóźnionego importu
15. nagłówek „CLR runtime”
16. wpis końcowy (musi być wyzerowany).

Należy zwrócić uwagę, aby nie czynić żadnych założeń co do położenia którychkolwiek wymienionych danych. W szczególności istnieje możliwość, iż zdefiniowany adres wirtualny w katalogu danych będzie wskazywał adres wirtualny nienależący do żadnej sekcji. Z wyników uzyskanych podczas części eksperymentalnej wynika, iż taka sytuacja najczęściej dotyczy tablicy bindowania. W tym przypadku adres wirtualny odpowiada w rzeczywistości pozycji w pliku, która znajduje się przed początkiem pierwszej sekcji.

### Tabela sekcji

Format pliku PE przewiduje, iż tuż za strukturą OPTIONAL\_IMAGE\_ HEADER znajdować się będzie tabela sekcji. Powodem jej istnienia jest potrzeba poinformowania programu ładującego o tym jakie w danym module są dostępne sekcje, w jaki sposób należy ich odszukać w pliku oraz gdzie mają się docelowo znajdować w pamięci wirtualnej. Każdy element tej tabeli to w rzeczywistości nagłówek sekcji, który reprezentuje struktura IMAGE\_SECTION\_HEADER. Jej budowa jest przedstawiona na Rysunku 10.

Poniżej opisane zostaną ważniejsze, z punkty widzenia analizy i modyfikacji wirusów komputerowych, pola struktury IMAGE\_SECTION\_HEADER:

Rysunek 10. Nagłówek sekcji

* Name – każdy nagłówek sekcji rozpoczyna się polem nazwy danej sekcji.   
  W rzeczywistości nie jest ona do niczego używana, a dodawana jest tylko po to, aby zapewnić programistom wygodę podczas szukania błędów   
  w ich programach. Warto mieć na uwadze, iż nazwa ta nie jest typowym dla programów łańcuchem znaków ASCII, więc nie jest zakończona znakiem zera. Składa się z 8 znaków i w szczególności może być pozostawiona pusta.
* VirtualSize – jest to faktyczny rozmiar sekcji, gdy zostanie ona załadowana do pamięci operacyjnej. W szczególności wartość ta jest zazwyczaj mniejsza niż rozmiar sekcji w pliku. Jedynym wyjątkiem od tej zasady jest sekcja danych, która poprzez możliwość posiadania danych niezainicjalizowanych nie potrzebuje ich przechowywać w pliku, ale loader systemowy musi wziąć ich rozmiar przy alokacji pamięci.
* VirtualAddress – jest to wirtualny adres względny danej sekcji, często mylony ze względu na nazwę z bezpośrednim adresem wirtualnym. Wartość ta musi być wielokrotnością opisywanego wcześniej wyrównania sekcji (SectionAlignment).
* SizeOfRawData–rozmiar danej sekcji na dysku. Wartość ta, podobnie jak RVA sekcji, musi być zaokrąglona zgodnie z wyrównaniem pliku (FIleAlignment). Kwestią wartą podkreślenia jest fakt, iż wyrównanie sekcji w pamięci sterowane jest poprzez odpowiednią wartość adresu wirtualnej, podczas gdy wyrównanie sekcji w pliku określane jest poprzez jej rozmiar. Ta subtelna różnica wynika z tego, iż sekcje w pliku zawierają dane wypełniające (ang. padding), które zazwyczaj są bajtami o wartości 0.
* PointerToRawData – odległość w bajtach od początku pliku do miejsca, gdzie rozpoczyna się kod danej sekcji. Na podstawie tego wskaźnika program ładujący jest w stanie odczytać dane modułu z pliku   
  i przekopiować je do pamięci. Istnieje możliwość, że wartość ta będzie równa 0. Oznacza to, iż dane sekcji nie znajdują się w pliku i będą wyliczone lub skopiowane z innego miejsca dopiero podczas ładowania danych programu.
* Characteristics – zestaw flag, które określają między innymi czy sekcja przechowuje kod wykonywalny, czy ma prawa odczytu, zapisu   
  i wykonania oraz czy posiada dane zainicjalizowane i niezainicjalizowane.

### Sekcje

W momencie, gdy program ładujący odczyta i przeanalizuje wszystkie wpisy   
w tablicy sekcji, dysponuje wszelkimi niezbędnymi danymi do faktycznego odczytu i załadowania pozostałych danych modułu. Kolejne podrozdziału przedstawiają opisy poszczególnych sekcji, ich znaczenia i schematów budowy.

#### Sekcja kodu

Jest to główna sekcja, która domyślnie posiada kod wykonywalny. Adres pierwszej instrukcji do wykonania określa punkt wejścia (ang. entry point), który zdefiniowany jest w nagłówku PE. Warto zauważyć, iż nie jest to równoznaczne   
z pierwszym bajtem tej części pliku. Typowe nazwy, umieszczane w nagłówku to „.text”, którą generuje kompilator Visual C++ lub „CODE”, którą generuje kompilator Borland C++. Sekcja ta, poza specjalnym znaczeniem przechowywanych w niej danych, wyróżnia się prawem do odczytu   
i wykonywania. Większość wirusów, które dokonują deszyfracji swojego kodu przed rozpoczęciem wykonywania, musi najpierw zmodyfikować zestaw praw dostępu, aby umożliwiały one również zapis.

#### Sekcja danych

W każdym module znajdują się pewne definicje danych, które ze względów praktycznych, jak i samej architektury systemu Windows, musza być przechowywane w osobnej sekcji. W rzeczywistości, kompilatory zazwyczaj dzielą te dane na kilka sekcji, z których każda przechowuje pewny logicznie spójny podzbiór tych danych. Najczęściej spotykanymi sekcjami danych są:

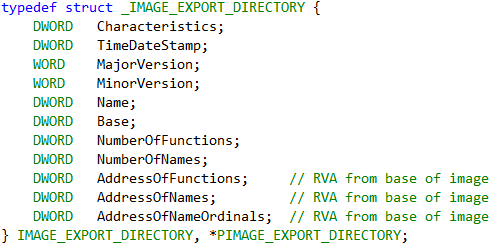
* sekcja „,.bss” – przechowuje dane niezainicjalizowane dla aplikacji, włączając w to wszystkie dane zadeklarowane wewnątrz kodu jako statyczne w funkcji, klasie lub pojedynczej jednostce translacji.
* sekcja „.rdata” – reprezentuje dane z prawami tylko do odczytu, takimi jak łańcuchy znaków, stałe oraz niektóre dane do debugowania.
* sekcja „.data” – przechowuje wszystkie pozostałe zmienne, które są zadeklarowane jako globalne dla danego modułu. Nie dotyczy to zmiennych automatycznych, które wg budowy architektury x86 alokowane są na żądanie w obszarze stosu.

#### Sekcja eksportu

Sekcja eksportu jest to sekcja, która określa zestaw symboli, które są udostępniane na zewnątrz danego modułu do użytku przez zewnętrzne aplikacje. Rzadko spotyka się, aby aplikacje miały zdefiniowane jakiekolwiek udostępniane funkcje. Jednak ze względu na wspólny format stosowany przez aplikacje   
i biblioteki współdzielone, sekcja ta została wprowadzona, aby informować program ładujący o liście wszystkich symboli, które są udostępniane przez daną bibliotekę DLL.

Funkcje mogą być eksportowane przez biblioteki na dwa sposoby: poprzez nazwę I poprzez numer porządkowy (ang. ordinal). Numer porządkowy to   
16-bitowa liczba jednoznacznie identyfikująca funkcję w obrębie konkretnego modułu. Jest unikalna tylko w obrębie pliku, do którego się odwołuje. Nazwa funkcji to ciąg znaków, który może zostać wykorzystany przez inną bibliotekę lub aplikację jako parametr wywołania funkcji GetProcAddress do odnalezienia adresu szukanego symbolu wewnątrz danej biblioteki.

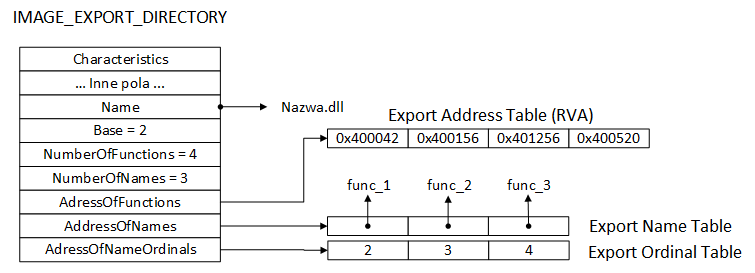
Główną strukturą, która przechowuje informacje potrzebne do identyfikacji udostępnianych na zewnątrz modułu funkcji, jest IMAGE\_EXPORT\_DIRECTORY. Jej schemat ilustruje Rysunek 11.

Najważniejsze pola tej struktury danych to:

Rysunek 11. IMAGE\_EXPORT\_DIRECTORY

* Name – adres wewnętrznej nazwy modułu. Pole to jest wymagane, ponieważ nazwa pliku może ulec zmianie w wyniku modyfikacji przez użytkownika. Jeżeli się tak stanie, program ładujący PE odwołuje się do nazwy wewnętrznej.
* Base – Początkowy numer porządkowy, potrzebny do pobrania indeksów w tablicy adresów funkcji.
* NumberOfFunctions – łączna liczba funkcji (symboli), która jest eksportowana z modułu.
* NumberOfNames – liczba symboli, które są eksportowane przez nazwę. Warto pamiętać, iż liczba ta nie jest liczbą wszystkich funkcji w danym module. Aby poznać ilość wszystkich symboli należy zawsze się odwoływać do pola NumberOfFunctions. W szczególnym przypadku liczba funkcji eksportowanych przez nazwę może być równa 0.
* AddressOfFunctions – jest to RVA, który wskazuje na tablicę wskaźników (również względnych) wewnątrz modułu. Najpopularniejszą nazwą tej tablicy jest EAT (skrót od Export Address Table).
* AddressOfNames – RVA do tablicy nazw funkcji, które są eksportowane. Zazwyczaj nazywana jest ENT (akronim pochodzący od Export Name Table).
* AddressOfNameOrdinals – RVA tablicy, która przechowuje numery porządkowe wszystkich nazwanych funkcji. Jej nazwa to EOT (od Export Ordinal Table).

Fizyczną ilustracją znaczenia poszczególnych pól obrazuje Rysunek 12.



Rysunek 12. Eksportowanie symboli przez moduł PE

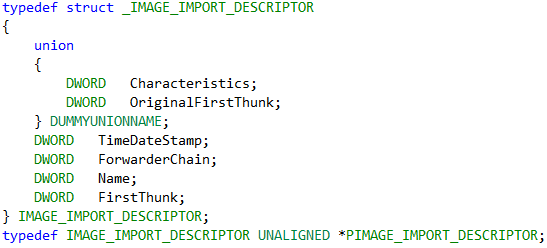
Na powyższym rysunku można zauważyć, iż nazwa biblioteki, której dotyczy przykład, to Nazwa.dll. Eksportuje ona 4 funkcje, z czego 3 używając do tego nazw   
i tylko jednej, do której można odwoływać się wyłącznie przez numer porządkowy. Pole Base informuje, iż pierwsza funkcja eksportowana przez nazwę ma indeks równy 2. Aby dowiedzieć się, która funkcja jest dostępna tylko przez numer porządkowy, należy sprawdzić który adres w EAT nie jest skojarzony z tabelą EOT. Sprowadza się to do wykluczenia wszystkich adresów o indeksach większych lub równych wartości pola Base. W tym przypadku jest to funkcja o indeksie 1.

Firma Microsoft chcąc udostępnić dla zewnętrznych aplikacji tylko wysokopoziomowy interfejs, ukrywając przy tym niskopoziomową implementację, która może się znacząco różnić pomiędzy dwoma wersjami systemu Windows, wprowadziła mechanizm tzw. eksportu przekierowującego (ang. export forwarding). Przykładem użycia tej techniki jest funkcja HeapAlloc, która jest eksportowana przez bibliotekę kernel32.dll, mimo iż jej faktyczna implementacja jest w postaci funkcji RtlAllocHeap w module ntdll.dll. Przekierowywanie implementowane jest poprzez umieszczanie w tablicy EAT adresu łańcucha znakowego w formacie ASCII postaci nazwa biblioteki, znak ‘.’ i nazwa funkcji (np. NTDLL. RtlAllocHeap) zamiast faktycznego adresu funkcji. Aby określić, czy funkcja jest dostępna w innym module należy sprawdzić, czy adres znajdujący się w tablicy eksportu odwołuje się do sekcji eksportu danego modułu (w przypadku funkcji przekierowanej) czy innego miejsca w tej bibliotece (w przypadku funkcji lokalnej).

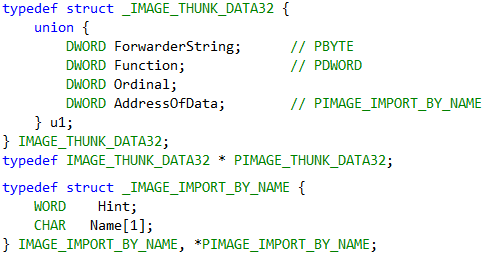
#### Sekcja importu

Celem istnienia sekcji importu jest informowanie programu ładującego   
o wszystkich funkcjach, które są zdefiniowane w innych modułach, a do których są odwołania w danym pliku wykonywalnym. Definicja tych informacji, aby była poprawna, musi być zgodna z formatem jaki jest używany przez biblioteki   
w mechanizmie eksportowania.

Loader systemu Windows jest odpowiedzialny podczas ładowania obrazu pliku do pamięci za odwzorowanie w przestrzeni adresowej wszystkich bibliotek DLL, od których plik jest w jakiś sposób zależny, odnalezienie wewnątrz nich adresów funkcji oraz udostępnienie ich głównej aplikacji. W tym celu zdefiniowana została struktura danych nazywana tablicą importu (ang. Import Address Table – IAT), która przechowuje adresy wirtualne wszystkich importowanych funkcji. Rolą programu ładującego jest uzupełnienie jej odpowiednimi wpisami.

Na początku sekcji importu zdefiniowana jest tablica struktur typu IMAGE\_IMPORT\_DESCRIPTOR, która przedstawiona jest na Rysunku 13.

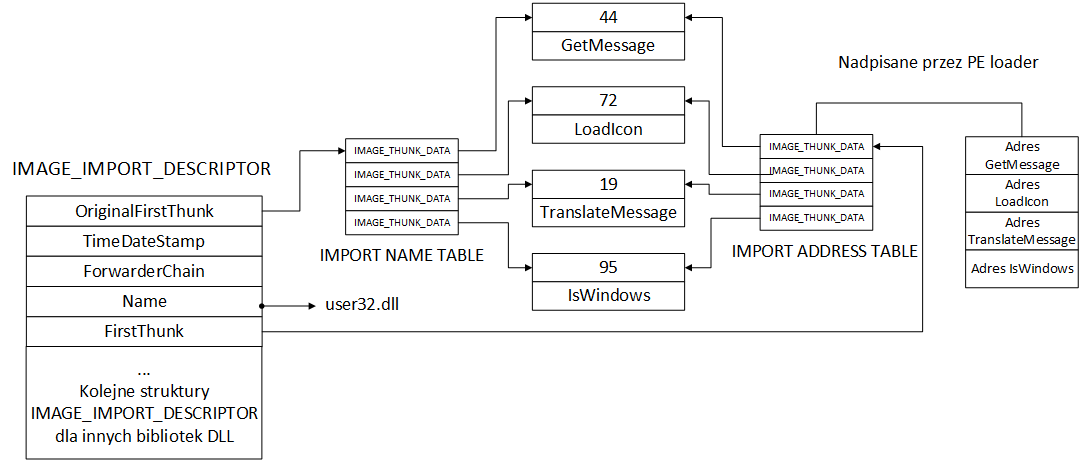
Rysunek 13. Struktura IMAGE\_IMPORT\_DESCRIPTOR

Każda instancja tej struktury danych opisuje jedną importowaną bibliotekę DLL. Przykładowo, jeżeli dany moduł importuje 3 biblioteki, to tablica ta będzie zawierała trzy struktury IMAGE\_IMPORT\_DESCRIPTOR. Najważniejsze pola tej struktury to OriginalFirstThunk, Name oraz FirstThunk. Pierwsze z nich określa RVA tablicy struktur IMAGE\_THUNK\_DATA. Pole Name przechowuje RVA nazwy biblioteki DLL w postaci łańcucha ASCII. Natomiast zmienna FirstThunk również przechowuje RVA tablicy struktur IMAGE\_THUNK\_DATA, która dodatkowo jest identyczna jak ta wskazywana przez OriginalThunkData. Obie tablice zakończone są wpisem, którego wszystkie pola są wyzerowane. Definicja IMAGE\_THUNK\_DATA przedstawiona jest na Rysunku 14.

Rysunek 14. IMAGE\_THUNK\_DATA

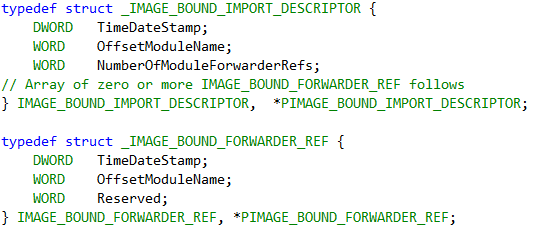
Pierwszym polem tej struktury jest unia, która efektywnie przyjmuje tylko dwie wartości: numer porządkowy importowanej funkcji lub wirtualny adres względny do struktury IMAGE\_IMPORT\_BY\_NAME, widocznej również na Rysunku 14.   
W momencie załadowania obrazu pliku do pamięci, tablice wskazywane przez pola FirstThunk w strukturach IMAGE\_IMPORT\_DESCRIPTOR są nadpisywane rzeczywistymi adresami funkcji, na podstawie danych zawartych w załadowanych do pamięci bibliotekach DLL. Od tego momentu tablica ta nazywana jest Import Address Table (IAT). Zaś druga, niezmieniona, tablica nazywana jest zazwyczaj Import Name Table (INT) lub Import Lookup Table (ILT).

Celem definicji dwóch identycznych tablic struktur IMAGE\_THUNK\_DATA jest umożliwienie odwoływania się do nazw i numerów porządkowych funkcji nawet w momencie, gdy druga z tablic zostanie nadpisana odpowiednimi adresami. Cały proces obrazowo przedstawia Rysunek 15. Przykładowo, gdy istnieje potrzeba znalezienia adresu funkcji IsWindows, program ładujący przeszukuje tablicę INT pod kątem obecności nazwy szukanej funkcji. Gdy ją znajdzie, odszukuje jej adres w IAT posługując się tym samym indeksem. Warto też mieć na uwadze, iż IAT rezyduje w sekcji, która domyślnie ma prawa wyłącznie do odczytu. Aby było możliwe nadpisanie tablicy importów, program ładujący tymczasowo zmienia prawa dostępu do sekcji, dodając jej prawo do zapisu. Po zakończonej operacji ponownie przywracane są poprzednie wartości uprawnień.



Rysunek 15. Schemat importowania symboli

Istotnym problemem podczas tworzenia IAT jest czas potrzebny na wyszukanie odpowiednich adresów funkcji. Aby zoptymalizować tą operację, firma Microsoft udostępnia wraz z konsolidatorem program o nawie bind.exe, który dokonuje predykcji adresów pod którymi zostaną załadowane biblioteki DLL   
i wstępnie podmienia adresy funkcji w tablicy importów podczas etapu budowania aplikacji. Jeżeli rzeczywiste adresy bazowe DLL będą zgodne z tymi założeniami to nastąpi znaczące skrócenie czasu ładowania programu.

 Korzystanie z tego mechanizmu optymalizacji wymusza konieczność sprawdzania przez program ładujący, czy przewidywane adresy importowanych funkcji są poprawne. W tym celu wprowadzona została struktura IMAGE\_BOUND\_IMPORT\_DESCRIPTOR, widoczna na Rysunku 16.

Rysunek 16. Struktura używana w procesie weryfikacji importowanych symboli

Dostęp do niej można uzyskać poprzez Data Directory w nagłówku PE   
w przypadku, gdy plik jest faktycznie bindowanym plikiem wykonywalnym (ang. bounded executable). Pierwszą czynnością jaką wykonuje loader systemu Windows jest porównanie wartości pola TimeDateStamp z tą zapisaną w samej bibliotece. Jeżeli nie są równe, to program zakłada, iż moduł jest zbindowany ze złą wersją biblioteki DLL i konieczne jest przeprowadzenie normalnego procesu wyszukiwania adresów importowanych symboli. Pole OffsetModuleName zawiera przesunięcie względem pierwszej struktury IMAGE\_BOUND\_IMPORT\_ DESCRIPTOR, które określa łańcuch znaków w formacie ASCII będący nazwą biblioteki współdzielonej. Zmienna NumberOfModuleForwarderRefs przechowuje informację o liczbie struktur typu IMAGE\_BOUND\_FORWARDER\_REF (także widocznej na Rysunku 16), które znajdują się tuż za daną strukturą IMAGE\_BOUND\_IMPORT\_DESCRIPTOR. Obie opisane struktury są bardzo podobne. Istota obecności tej drugiej sprowadza się do potrzeby sprawdzania również adresów funkcji przekierowywanych do innych DLL.

#### Sekcja relokacji

W momencie, gdy konsolidator tworzy plik wykonywalny, dokonywane jest pewne założenie nt. miejsca w pamięci, gdzie plik będzie załadowany. Wynik tej operacji zapisywany jest w postaci pola ImageBase w nagłówku PE. Bazując na tej wartości, linker ma możliwość uzupełnienia pliku prawdziwymi adresami odwołującymi się do miejsc w sekcjach kodu i danych w przypadku, gdy instrukcje maszynowe wymagają pełnego adresu. Przykładami takich rozkazów są instrukcje skoku bezwzględnego. Istnieje jednak możliwość, iż z jakiegoś powodu plik zostanie odwzorowany w innym miejscu pamięci wirtualnej. W takim przypadku wszystkie ustalone w fazie konsolidacji adresy są niepoprawne. Celem istnienia sekcji relokacji jest dostarczenie programowi ładującemu wszystkich niezbędnych informacji, które pozwolą mu, znając nowy adres bazowy modułu, poprawić adresy bezwzględne. Gdy jednak moduł jest ładowany zgodnie z przewidywaniami linkera, sekcja ta jest niepotrzebna i wobec tego ignorowana.

Wpisy znajdujące się sekcji relokacji nazywane są relokacjami bazowymi (ang. base relocations), ponieważ zależą od adresu bazowego aplikacji. Tworzą one pewnego rodzaju listę lokacji w ładowanym obrazie, które muszą zostać poddane modyfikacji. Sprowadza się ona do obliczenia różnicy pomiędzy zakładanym adresem bazowym, a jego faktyczną wartością oraz dodanie jej do wszystkich wpisów w tej liście.

Lista relokacji składa się z bloków, z których każdy opisuje jedną stronę pamięci o rozmiarze 4 KB, oraz ciągu 16-bitowych wartości, z których każda opisuje jeden adres wymagający przeliczenia wewnątrz danej strony pamięci. Blok definiuje wirtualny adres względny (RVA) strony, której dotyczy oraz całkowity swój rozmiar. Następnie każda z 16-bitowych wartości definiuje typ relokacji (bardziej znaczące 4 bity) oraz przesunięcie względem adresu strony, gdzie znajduje się konkretny adres do zamiany (mniej znaczące 12-bitów).

Przykładowo, niech linker określa przy tworzeniu pliku wykonywalnego, że adres bazowy będzie równy 0x10000. Dodatkowo niech na pozycji 0x1234 wewnątrz ładowanego obrazu znajduje się wskaźnik do łańcucha znakowego, który znajduje się w pamięci wirtualnej pod adresem 0x14002. Zatem wskaźnik powinien mieć wartość ustaloną na 0x14002. Jednak z pewnych względów plik zostaje załadowany pod adresem bazowym równym 0x60000. Oznacza to, iż cały obraz jest przesunięty w stronę większych adresów względem tego, który został założony przez konsolidator. Zatem łańcuch znakowy znajduje się pod innym adresem wirtualnym niż ten, który określa wskaźnik w kodzie. Różnica w adresach bazowych wynosi 0x50000, zatem fizycznie stała łańcuchowa znajduje się teraz pod adresem 0x64002. Wskaźnik jest więc nieprawidłowy i musi zostać poddany modyfikacji. Aby rozwiązać ten konflikt, program ładujący odczytuje z listy relokacji wszystkie miejsca w pamięci, do których musi dodać wartość 0x50000. Wśród nich znajduje się również adres błędnego wskaźnika. Po zaaplikowaniu wszystkich zmian, adresy bezwzględne są ponownie poprawne.

# Metody detekcji wirusów

Celem tego rozdziału jest zaznajomienie czytelnika ze znanymi sposobami detekcji wirusów komputerowych. Znajomość obecnego stanu wiedzy  
dotyczącego sposobów wykrywania złośliwego oprogramowania jest niezbędna, aby w najefektywniejszy sposób modyfikować istniejący kod malware-u   
w celu ukrycia go przed programami antywirusowymi. Kolejne podrozdziały opisują zebrane z literatury przedmiotu metody algorytmicznego wykrywania złośliwego kodu. Istnienie wielu technik detekcji nie jest jednak równoznaczne   
z ich implementacją o oraz użyciem w dostępnych rozwiązaniach antywirusowych. Przyczyną może być wysoki stopień fałszywych alarmów, szybkość działania lub wprowadzone obniżenie wydajności komputera. Metody zostały pogrupowane biorąc pod uwagę ich stopień zaawansowania oraz typ wykorzystywanych algorytmów.



## Skanery pierwszej generacji

Skanery pierwszej generacji są jedną z bardziej rozpowszechnionych metod wykrywania złośliwego kodu. Zawdzięczają to prostej implementacji i wysokiej skuteczności przeciwko znanym zagrożeniom. W większości opierają się   
o statyczną analizę ciągów binarnych. Podczas skanowania przeprowadzane są pewne obliczenia i przekształcenia (jak obliczanie skrótu lub oznaczanie charakterystycznych ciągów bajtów), a rezultaty są porównywane z bazą danych wyników tych samych operacji wykonanych na zidentyfikowanych już niebezpiecznych plikach. Bazy danych nazywane są często bazami sygnatur   
i opracowywane są w specjalnych laboratoriach danego producenta silnika antywirusowego.   
W przypadku, gdy wykryte zostają nowe zagrożenia o krytycznym znaczeniu, firma publikuje aktualizację bazy sygnatur, która jest pobierana przez wszystkich klientów. Wspomnianą wadą takich skanerów jest bardzo słaba wykrywalność nowych typów wirusów.

### Skanowanie łańcuchów

Jedną z najprostszych metod wykrywania złośliwego kodu, zaliczanych do skanerów pierwszej generacji, jest skanowanie łańcuchów. Idea polega na wyodrębnieniu z danego pliku wirusa ciągów (jednego lub wielu) bajtów, które są dla niego charakterystyczne. Utrudnieniem jest wymaganie, aby podczas selekcji zminimalizować ryzyko wystąpienia tej samej sekwencji w plikach bezpiecznych.   
Analitycy starają się zawsze wybierać fragment wirusa, który nie zawiera danych liczbowych. Może to być klucz deszyfratora lub licznik pętli. Wybrany ciąg danych (nazywany często sygnaturą) jest wtedy odporny na drobne modyfikacje kodu, mogące wynikać z metod adaptacji wirusa do nowego nosiciela podczas replikacji. Przykładowo, darmowy skaner antywirusowy ClamAV (www.clamav.net) umożliwia definiowanie sygnatur opierających się o odległości od początków sekcji, punktu wejścia programu (ang. Entry Point) lub względem całego pliku [14].

### Symbole wieloznaczne

Metoda symboli wieloznacznych jest rozwinięciem analizy łańcuchów bajtów. Polega ona na celowym pominięciu dopasowania kilku ustalonych bajtów   
w danym łańcuchu. Dzięki temu, analitycy mogą wybierać jako sygnaturę fragmenty kodu zawierające adresy, stałe liczbowe lub stałe znakowe, bez konieczności ograniczania danej sygnatury do jednego wariantu wirusa. Innym sposobem wykorzystania wieloznaczności w sekwencji jest wyszukiwanie wskazanej wartości wśród kilku kolejnych bajtów w stosunku do bieżącej pozycji. To podejście uodparnia skaner antywirusowy na drobne mutacje kodu, jak zamiana kolejności instrukcji nie powodująca efektów ubocznych lub wstawianie martwego kodu. Ponadto, Peter Shor w swojej książce [15] opisuje sposób tworzenia swojego rodzaju wyrażeń regularnych z wykorzystaniem obu opisanych podejść.

Dana metoda, poprzez zastosowanie wieloznaczności, odrzuca sztywne ograniczenia wprowadzone przez metodę skanowania łańcuchów. Stanowi to jednocześnie problem, który może być równie katastrofalny jak niewykrycie danego malware-u. Istnieje bowiem możliwość, iż sygnatura wieloznaczna, która kwalifikuje plik jako zainfekowany wirusem A, poprzez zbyt luźne ograniczenia oznaczy plik zainfekowany wirusem B jako A. Istnieją dwie niepożądane konsekwencje takiej pomyłki. W pierwszym przypadku użytkownik może zostać błędnie poinformowany o infekcji wirusem, który nie jest groźny i przez to zlekceważy komunikat programu antywirusowego. W drugim przypadku, program zabezpieczający może podjąć niewłaściwy sposób leczenia wirusa, co może potencjalnie zniszczyć dany plik lub nawet doprowadzić do załamania całego systemu. Z tego powodu wszystkie metody opierające się na obliczaniu sygnatur zazwyczaj są używane tylko podczas wstępnej eliminacji zagrożeń. Następnie pliki oznaczone jako podejrzane (tzw. oflagowywanie) kierowane są na bardziej zaawansowane testy. Dzięki takiemu podejściu można uzyskać sporą wydajność poprzez minimalizację uruchomień długotrwałych skanerów heurystycznych lub behawioralnych.

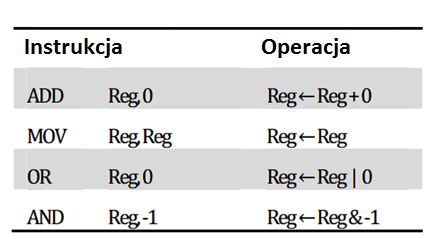
### Metoda zakładek

Kolejną metodą analizy binarnej struktury złośliwego kodu jest metoda zakładek (nazywana czasami metodą bajtów kontrolnych). Wprowadzona została jako środek zwiększenia niezawodności i dokładności wyżej opisanych technik. Polega ona na sprawdzaniu, czy pod odpowiednim przesunięciem względem początku kodu wirusa (tzw. bajtem zerowym) znajdują się bajty o ustalonej wartości. Dla każdego typu wirusa odpowiedni jest inny zestaw zakładek. Przykładowo, jeżeli dany wirus infekując plik podmienia jego oryginalną zawartość to zazwyczaj, w celu umożliwienia późniejszego uruchomienia nosiciela, kopiuje jego oryginalny kod w ustalone miejsce. W takiej sytuacji dobrą zakładką będzie bajt określający pozycję skopiowanych danych. Innym przykładem mogą być wirusy sektora startowego (ang. bootsector). Podczas infekcji złośliwy kod musi odnaleźć dokładną pozycję sektora rozruchowego na dysku. Zazwyczaj jest ona stała dla danego systemu i jest identyfikowana numerem głowicy, ścieżki i sektora (w przypadku dysków talerzowych). Dla takiego przypadku, odpowiednimi zakładkami będą bajty określające trzy wymienione numery.

## Skanery drugiej generacji

### Skanowanie inteligentne

W trakcie rozwoju technik pisania wirusów komputerowych oraz programów ich wykrywających, szybko stało się jasne, iż dane złośliwe oprogramowanie przestaje być zagrożeniem w momencie obliczenia jego sygnatury i aktualizacji bazy danych w silnikach antywirusowych. Twórcy malware-u zaczęli stosować techniki zaciemniania kodu, wprowadzania drobnych poprawek i modyfikacji, aby sygnatura ich kodu była zawsze inna. W ten sposób powstały nowe typy wirusów: metamorficzne i polimorficzne. Ich działanie polega na samodzielnej modyfikacji (często określanej mianem mutacji) własnego kodu. Jedną z najprostszych technik jest dołączenie w dowolnych miejscach tzw. śmieciowego lub martwego kodu. Istnieje bowiem zestaw operacji maszynowych, które nie posiadają efektów ubocznych pojedynczo lub w ustalonej grupie. Przykładami mogą być dodawanie wartości zero do rejestru lub operacja AND z wartością „-1” (postać binarną   
w kodowaniu U2 składa się z samych zapalonych bitów). Szczególnym przypadkiem jest też operacja NOP (ang. no operation), która została przewidziana w architekturze x86 specjalnie do tego celu, czyli nie wykonywania żadnej operacji. Rysunek 17 przedstawia poglądowy zestaw instrukcji, z których może składać się martwy kod.



Rysunek 17. Przykłady instrukcji martwego kodu [25]

Metoda inteligentnego skanowania polega na statycznej analizie kodu wykonywalnego i wykrywanie sekwencji instrukcji nie mających żadnego znaczenia dla logiki wykonywanego kodu. Są one wtedy pomijane podczas innych typów skanowań. Dzięki temu skaner staje się odporny na proste metody mutacji. Technika ta jest często łączona z metodą symboli wieloznacznych, gdzie symbolami wieloznacznymi są w tym przypadku śmieciowe instrukcje. Zadanie jest o tyle trudne, że niekiedy nieskomplikowane operacje zaciemniające kod są rozłożone na wiele instrukcji. Przykładem może być operacja dodania wartości do jakiegoś rejestru z późniejszym odjęciem jej, ale w zupełnie innym miejscu kodu lub z odejmowaniem kilku mniejszych liczb, których suma jest równa dodanej wartości.

### Dopasowanie szkieletu

Opracowanie algorytmów oczyszczających kod z nieznaczących instrukcji   
w trakcie skanowania inteligentnego pozwoliło w znacznym stopniu wyodrębnić logikę wirusów komputerowych z dostępnych próbek. Ewolucja wirusów metamorficznych i polimorficznych sprawiła jednak, iż nawet wtedy dokładna identyfikacja złośliwego kodu nie jest prosta, a stosowanie jednej uniwersalnej metody leczenia do całej rodziny malware-u wywodzącej się z jednego prototypu jest groźne w skutkach dla systemu klienta. Eugene Kaspersky bazując na znanych do tej pory metodach opracował technikę detekcji szkieletowej, która po wstępnym oczyszczeniu kodu opiera się na dopasowaniu nie jednego, a kilku łańcuchów bajtów. Technika ta nazywana jest także identyfikacją niemal dokładną.

Późniejszym rozwinięciem tego sposobu detekcji i kwalifikacji wirusów komputerowych stała się metoda identyfikacji dokładnej, która precyzyjnie określa wariant danego wirusa. W ogólności, technika opiera się na skonstruowaniu mapy stałych punktów w danym typie złośliwego kodu. Proces wymaga określenia, które ciągi bajtów są niezmienne, a następnie obliczenia przesunięć krańców tych ciągów względem początku kodu wirusa (bajtu zerowego). Następnie dla tak powstałych sekwencji stałych liczona jest suma kontrolna, która wchodzi w skład mapy. Dzięki odpowiedniemu doborowi punktów, można rozróżnić nawet bardzo podobne warianty tego samego wirusa. Należy jednak mieć na uwadze, iż nie zawsze aż taka dokładność jest potrzebna.   
W niektórych przypadkach możliwe jest bowiem zastosowanie tej samej metody leczenia. Przykładem mogą być dwie znane wersje wirusa Stoned, który podczas działania wypisywał na ekran napis „Your PC is now Stoned! Legalise marijuana!” w pierwszej wersji oraz napis „Your PC is now stoned! Legalize marijuana!”   
w drugiej. Na pierwszy rzut oka oba napisy są takie same. Różnica polega na wielkości liter w wyrazie „stoned” oraz użyciu angielskiej wersji słowa „Legalize”. Jeżeli mapa stałych obejmowałaby binarną reprezentacje tego napisu, można byłoby dokładnie określić z którym wariantem mamy styczność. Jednak w tym przypadku różnica nie ma znaczenia dla logiki wirusa i może być zastosowana ta sama szczepionka.

## Metody statycznego wykrywania deszyfratora

Niekiedy zdarza się, iż metoda prostego dopasowywania łańcuchów jest nieskuteczna dla pewnych wirusów. Poza możliwymi mutacjami, które sprawiają iż mimo braku zmiany logiki programu kod wygląda zupełnie inaczej, złośliwe oprogramowanie stosuje również proste metody szyfrowania. Zazwyczaj ograniczają się one do operacji typu ADD, SUB, NOT lub XOR na całym kodzie wirusa. Implikuje to jednocześnie potrzebę istnienia pewnego fragmentu kodu   
w samym wirusie, który nie jest zaszyfrowany, jest uruchamiany na samym początku i wykonuje operację deszyfrowania. Sposób w jaki tego dokonuje jest bezpośrednio sprzężony z metodą szyfrowania. W przypadku operacji ADD i SUB wystarczy dokonać operacje przeciwną z takim samym kluczem. W przypadku metody NOT i XOR należy wykonać ponownie tą samą operację.

Statyczne wykrywanie kodu deszyfratora jest operacją względnie łatwą, gdyż na pewnym etapie zawsze sprowadza się do odnalezienia jednej ze wspomnianych instrukcji. Problemem są deszyfratory, które korzystają z bardziej zaawansowanych operacji lub kombinacji kilku poprzednich. Przykładem takiego typu złośliwego kodu są wirusy z rodziny SMEG (skrót od „Simulated Metamorphic Encryption Generator”). Charakteryzują się one deszyfratorem, który zmienia wartość swojego klucza przy każdej iteracji oraz stosuje różne operacje szyfrujące w różnych swoich wariantach. Z tego powodu opracowana została metoda X-RAY, która dokonuje analizy kryptograficznej kodu wirusa, w celu ustalenia metody szyfrowania, klucza oraz metody zmiany jego wartości bez odnajdywania kodu samego deszyfratora. Założenie, iż klucz zmieniany jest algorytmicznie,   
a odczytywany z zapisanej sekwencji ma sens, gdyż w przeciwnym razie kod wirusa miałby dużo stałych bajtów możliwych do wykrycia opisanymi wcześniej metodami. Autorem tej techniki jest Frans Veldman, inspirowany dokumentacją techniczną Eugena Kaspersky-iego. Opiera się ona głównie na znajomości kilku pierwszych bajtów odszyfrowanej wersji wirusa. Implementowane są odpowiednie procedury, które wykonują wszystkie popularne instrukcje używane przy szyfrowaniu. Kod wirusa zapisywany jest do kilku różnych buforów, po jednym na każdy typ przekształcenia. Następnie dla wszystkich przypadków obliczany jest klucz na podstawie pierwszego znanego bajtu wirusa (wykonanie operacji odwrotnej z kluczem równym bajtowi zerowemu malware-u),   
a przesunięcie klucza obliczane jest na podstawie prostych wzorów znanych po ręcznej analizie kilku próbek silnika SMEG. Następnie dla wszystkich buforów następuje odpowiadająca im procedura szyfrowania. W końcowej fazie następuje sprawdzenie, czy w którymś z buforów kilka pierwszych bajtów odpowiada znanej sygnaturze wirusa. Jeżeli tak, następuje wtedy dokładna weryfikacja złośliwego kodu innymi metodami, jednak operująca już na odszyfrowanym kodzie. Wadą tej metody jest długi czas działania i potencjalnie skomplikowana implementacja,   
w przypadku różnych typów silników generujących deszyfratory.

## Analiza behawioralna

Istnieje duża grupa wirusów komputerowych, które są w stanie ukryć swoją obecność przed metodami statycznej analizy kodu wykonywalnego. Wachlarz rozwiązań stosowanych w tym celu jest bardzo szeroki. Może to być spowodowane używaniem zaawansowanych metod szyfrowania, które są odporne na ataki kryptograficzne. Częstym przypadkiem jest też brak informacji   
o danym wirusie w bazie danych silnika skanującego. W takim przypadku jedyną skuteczną metodą obrony komputera klienta jest próba emulacji działania kodu w odizolowanym środowisku oraz monitorowanie jego działania. W tym celu implementowana jest specjalna piaskownica (ang. sandbox), która posiada zaimplementowaną emulację CPU, pamięci oraz obsługi API systemowego danego systemu operacyjnego. Kod jest wczytywany do emulowanej pamięci, a następnie odczytywany bajt po bajcie. Następuje jego analiza oraz wykonanie odpowiadającej mu akcji. Monitorowane są wywołania systemowe i w przypadku wykrycia, uruchamiana jest specjalna funkcja, która z punktu widzenia wykonywanego kodu ma wymagany interfejs i zwraca odpowiednią wartość, bez dokonywania fizycznych zmian w systemie. Wadą tego rozwiązania jest potrzeba dodatkowego uruchomienia aplikacji w izolowanym środowisku, przed faktycznym wykonaniem tej czynności w systemie. Emulacja trwa zazwyczaj, z doświadczenia autora, około 10 - 15 sekund. Każde podejrzane zachowanie odnotowywane jest w pamięci emulatora. Jeżeli nie stwierdzono żadnych alarmów, kod uruchamiany jest normalnie w systemie. W przeciwnym wypadku aplikacja jest blokowana   
i podejmowana jest akcja zgodna z polityką danego antywirusa.

Emulacja ma dodatkową zaletę w postaci wykonania faktycznego kodu. Istnieją bowiem wirusy, które stosują pewne metody obrony przed deasemblacją   
i statyczną analizą kodu. Przykładem może być kod robaka Conficker, którego kod po transformacji kodu assemblera (inżynieria wsteczna) nie miał sensu, gdyż na pewnym etapie wykonywał skok w środek innej instrukcji. Okazuje się jednak, iż operacja ta miała sens, gdyż po rozpoczęciu analizy dokładnie od docelowego miejsca skoku ujawniały się nowe, nieodkryte dotąd instrukcje. Emulacja wykonywania programu jest całkowicie odporna na taką technikę, gdyż analiza każdej kolejnej instrukcji jest wykonywana zawsze z definicji.

### Dynamiczna detekcja deszyfratorów

Metody emulacji kodu są bardzo przydatnym narzędziem w walce   
z nieznanymi typami silników szyfrujących. Zastosowanie prostej kontroli pętli   
i monitorowania wystąpień w nich instrukcji XOR, ADD lub SUB z ustaloną stałą wartością lub wartością w stałym rejestrze może wspomagać proces wyszukiwania kodu deszyfratora. Dodatkowo, jeżeli pętla stosuje inkrementację lub dekrementację pewnej wartości (np. instrukcja *loop* w architekturze x86 dekrementuje wartość rejestru *ecx*, aż do stwierdzenia wartości 0), istnieje duże prawdopodobieństwo iż wartość ta w momencie inicjalizacji była długością szukanego wirusa. Ustalenie wtedy klucza jest zadaniem względnie prostym.

## Analizy heurystyczne

Wprowadzenie metod heurystycznych do analiz i klasyfikacji wirusów komputerowych jest względnie nowoczesną metodą, która została opracowania   
w momencie zaprezentowania przez firmę Microsoft formatu plików wykonywalnych Portable Executable. Spowodowane było to głównie czynnikiem wydajnościowym. Ograniczenie potencjalnie dużego zbioru analizowanych plików możliwie jak najmniejszej. Ma to istotne znaczenie, gdyż można w ten sposób uniknąć wielu długotrwałych analiz sygnaturowych lub emulacji wykonania. Większość złośliwego kodu wprowadza w plikach PE niestandardowe modyfikacje lub wręcz niecałkowicie wspiera ten format, pozostawiają po swojej aktywności pewne niezgodności. Prace [16] i [17] dowodzą, iż w większości przypadków fakt wykrycia niebezpiecznego lub podejrzanego kodu w pliku wykonywalnym wiązał się z obecnością pewnych zmian w strukturze samego pliku. Z tego powodu skanery antywirusowe zaczęły parsować pliki wykonywalne, w celu wyszukiwania ustalonych z góry właściwości. Najpopularniejsze z nich to:

* punkt wejścia programu znajduje się w sekcji innej niż pierwsza (sekcja kodu zazwyczaj jest na samym początku),
* przestrzenie pomiędzy sekcjami są wypełnione bajtami niezerowymi (zazwyczaj wypełniane są zerami przez linker),
* obecność wielokrotnych nagłówków PE (dozwolony jest tylko jeden),
* podejrzane nazwy sekcji (większość kompilatorów jest zgodna   
  z konwencją ustaloną przez Microsoft),
* przekazywanie sterowania pomiędzy sekcjami,
* sekcja kodu ma prawo do zapisu (kod nie może modyfikować sam siebie),
* sekcja danych ma prawo do wykonania (dane nie mogą być wykonywane),
* rozmiar sekcji na dysku jest istotnie różny od rozmiaru sekcji w pamięci (różnice w tych rozmiarach wynikają zazwyczaj z obecności danych niezainicjalizowanych – wtedy sekcja w pamięci ma większy rozmiar lub wynikają z zaokrąglenia do wyrównania naturalnego w pamięci i w pliku)
* podejrzane importy z biblioteki kernel32.dll (zwłaszcza importowanie funkcji GetModuleHandleA, GetProcAddress oraz LoadLibraryA przez numer porządkowy),
* nieprawidłowy rozmiar sekcji kodu i danych w nagłówku PE,
* nieprawidłowy wskaźnik na sekcję z kodem w nagłówku PE.

Skanery heurystyczne po wykryciu każdej z wymienionych wyżej właściwości oznaczają analizowany plik odpowiednią flagą. Umiejętny dobór zestawu takich flag umożliwia względnie optymalny stopień wykrywalności niebezpiecznych plików. Metoda ta nie jest stosowana samodzielnie, gdyż nie określa dokładnego wariantu lub rodziny malware-u, na podstawie której można byłoby wybrać odpowiednią metodę leczenia.

# Projekt malware-scrambler

W celu realizacji zaplanowanych eksperymentów weryfikujących tezę postawionej w pierwszym rozdziale, został zaprojektowany i zaimplementowany system malware-scrambler. Jego głównym celem jest automatyczna modyfikacja wejściowego pliku wykonywalnego na podstawie zadanego scenariusza, zdefiniowanego w postaci pliku konfiguracyjnego. Program ma możliwość generowania pliku wykonywalnego, który będzie istotnie różnił się budową (zarówno na poziomie logicznym jak i na poziomie binarnym) od pliku wejściowego, ale będzie wykonywał dokładnie te same czynności, tak te widoczne dla użytkownika jak i pewne dodatkowe, ukryte operacje.

Kolejne podrozdziały opisują architekturę systemu, wymagania, które ją kształtowały, cykl życia modyfikowanego programu oraz główne koncepcje których zastosowanie wpłynęło pozytywnie na rozszerzalność, konfigurowalność   
i skalowalność aplikacji. W dalszej części opisane zostały modyfikacje, jakim poddawane są próbki wirusów. Inspirując się napotkanym w Internecie wpisem, autor systemu określa pojedynczą modyfikację mianem mutacji, a moduł programu wykonujący tą operację mianem silnika mutacyjnego. Zestaw kilku silników mutacyjnych, które tworzą pewien układ nazywany jest potokiem mutacyjnym.

## Wymagania systemu

Proces projektowania aplikacji malware-scrambler rozpoczęła faza gromadzenia podstawowych wymagań funkcjonalnych i niefunkcjonalnych. Dzięki temu przez cały etap implementacji oprogramowania były jasno zdefiniowane cele, które trzeba było uwzględniać przy ustalaniu kierunków rozwoju systemu. Zidentyfikowane zostały następujące wymagania funkcjonalne:

* wsparcie modyfikacji pliku przez szyfrowanie,
* dodawanie martwego kodu,
* usuwanie sekcji importów,
* automatyzacja procesu modyfikacji i generacji kodu binarnego,
* niezmienność zachowania aplikacji po modyfikacji.

Dodatkowo, ustalone zostały następujące wymagania niefunkcjonalne:

* obsługa maksymalnie wielu wariantów plików wykonywalnych,
* wysoka konfigurowalność ustawień modyfikacji plików wejściowych,
  + umożliwienie niezależnej modyfikacji poszczególnych modułów modyfikujących dane wejściowe,
  + umożliwienie rozszerzenia systemu o dodatkowe moduły modyfikujące bez ingerencji w istniejący kod,
* łatwość zrozumienia i modyfikacji ustawień programu.

Biorąc pod uwagę zastosowanie systemu jak i liczbę danych wejściowych, uznane zostało, iż czas modyfikacji pojedynczego pliku nie jest parametrem krytycznym i może zostać pominięty w fazie implementacji, w przypadku gdy jest w konflikcie z którymś z pozostałych wymagań lub gdy obniża możliwość debugowania. Z tego też powodu zdecydowano się na umieszczanie jak największej liczby komunikatów informacyjnych w postaci logu systemowego, wypisywanego na standardowe wyjście. Autor projektu zdaje sobie sprawię, iż takie podejście znacząco obniża wydajność programu, jednak doświadczenie pokazało, że liczba nietypowych lub nieudokumentowanych opcji w plikach zgodnych z formatem Portable Executable jest tak duża, iż czas poświęcony na szukanie potencjalnych błędów logicznych byłby niewspółmierny z czasem jaki można zaoszczędzić na pominięciu wypisywania dodatkowych informacji.

## Cykl życia modyfikowanych plików

Każdy plik modyfikowany jest całkowicie niezależnie od pozostałych, bez względu na powodzenie lub sam rezultat modyfikacji poprzednich plików. Proces rozpoczyna się od wczytania zawartości pliku wykonywalnego do pamięci komputera. W tym celu powstał szereg klas i struktur danych, które w możliwie elastyczny sposób odzwierciedlają fizyczny układ pliku PE.

Główna klasa, która reprezentuje pojedynczy plik Portable Executable, nazywa się PEFile. Jej polami składowymi są wszystkie standardowe pola i sekcje,   
które zawiera plik aplikacji, tj. pole związane ze wsteczną kompatybilnością   
z systemem DOS, nagłówek PE, katalog danych, kody poszczególnych sekcji oraz ich nagłówki. Klasa ta ponadto implementuje interfejs, który umożliwia dostęp do wszystkich wymienionych wyżej struktur. Wszystkie pola inicjalizowane są poprzez odczyt i skopiowanie odpowiednich ciągów bajtów, a nie poprzez zwykłe ustawienie wskaźnika na odpowiednie bajty w pliku. Dzięki temu, nawet jeżeli plik zostanie zamknięty, klasa PEFile wciąż będzie umożliwiała pracę z odczytanymi danymi. Dodatkową korzyścią płynącą z przechowywania modyfikowanego pliku w postaci zestawu osobnych struktur, które go tworzą w pamięci jest fakt, iż można go od tej pory dowolnie modyfikować, rozszerzać, zmieniać kolejność danych oraz usuwać je, bez konieczności modyfikowania za każdym razem pliku fizycznego i dbania o jego spójność.

Po zainicjalizowaniu, klasa PEFile w pełni reprezentuje bieżąco analizowany plik wykonywalny. Aby spełnić wymaganie dotyczące konfigurowalności   
i rozszerzalności typów modyfikacji plików PE wprowadzone zostało pojęcie mutacji, jako pojedynczej edycji pliku, ukierunkowanej na specyficzną cechę formatu PE. Klasy zawierające kod implementujący tą funkcjonalność nazwane zostały silnikami mutacyjnymi. Każda z nich implementuje funkcję z interfejsu IMutationEngine, która jako swoje parametry wejścia-wyjścia przyjmuje wskaźnik na klasę PEFile oraz mapę kluczy, która stanowi kontekst mutacji. Pierwszy parametr zapewnia dostęp do modyfikowanego pliku. Drugi zaś umożliwia przekazywanie danych pomiędzy poszczególnymi silnikami. Dzięki takiemu podejściu możliwe było utworzenie ze wszystkich silników mutacyjnych potoku, za pomocą którego transformowany jest plik PE. Odzwierciedlona jest wtedy idea,   
w której dane wyjściowe jednej modyfikacji są danymi wejściowymi następnej.

Po zakończeniu serii wywołań funkcji mutujących, następuje faza ponownego zapisu danych klasy PEFile do postaci pliku zgodnego z formatem Portable Executable. Najpierw ustalany jest całkowity rozmiar danych, a następnie na tej podstawie tworzony jest plik wypełniony samymi zerami o wyliczonym rozmiarze. Następnie zgodnie ze znanym standardem PE serializowane są te struktury, których położenie jest stałe w każdym pliku. Dotyczy to kodu systemu DOS, nagłówka PE oraz nagłówków sekcji. Następnie kopiowany jest kod wszystkich sekcji, z uwzględnieniem ustalonych dla nich adresów początkowych przez silnik SectionResizer. W ten sposób, po zakończeniu tej fazy powstaje na dysku kompletny i prawidłowy plik wykonywalny.

## Potok mutacyjny

Potok mutacyjny tworzony jest dynamicznie dla każdego pliku na podstawie pliku konfiguracyjnego, który parametryzuje działanie całej aplikacji malware-scrambler. Przykładowy plik konfiguracyjny widoczny jest na Rysunku 18. Po wczytaniu pliku wykonywalnego, odczytywana jest konfiguracja potoku mutacyjnego oraz tworzony jest zestaw odpowiednich klas. W celu umożliwienia łatwego rozszerzania dostępnych silników wykorzystany został wzorzec fabryki abstrakcyjnej. Implementacja samego potoku sprowadza się do utworzenia listy wskaźników na silniki mutujące. Umożliwia to też dynamiczną zmianę konfiguracji kolejnych faz mutacji, bez konieczności ponownej kompilacji całego projektu. Przy takim podejściu modyfikacja pliku polega na wykonaniu pętli, w której dla każdego elementu listy wywoływana jest funkcja *process* z interfejsu IMutationEngine. Po zakończeniu pracy z danym plikiem PE potok jest niszczony, aby zmiany   
w strukturze kontekstu nie miały wpływu na pozostałe pliki.



Rysunek 18. Przykładowy plik konfigurujący potok mutacyjny

## Silniki mutacyjne

W poprzednim rozdziale wprowadzone zostało pojęcie silnika mutacyjnego oraz potoku, który umożliwia ustawienie wielu instancji różnych silników w jedną ustaloną ścieżkę. Celem tego podrozdziału jest opisanie wszystkich zaimplementowanych koncepcji mutacji plików wykonywalnych, ich ograniczeń oraz wzajemnych interakcji. Wszystkie zaprojektowane mutacje zostały podzielone tematycznie ze względu na główną braną pod uwagę cechę pliku.

### Silniki pomocnicze

Poniżej przedstawione są wszystkie silniki, które nie stanowią nowej metody mutacji plików wykonywalnych, a jedynie są wykorzystywane przez inne moduły. W systemie reprezentowane są jako osobne silniki dla zachowania spójności   
z pozostałymi komponentami.

#### Zmiana rozmiaru i położenia sekcji

Fakt rozbioru pliku formatu PE na części i przechowywanie ich osobno   
w pamięci narzuca wymaganie, aby projektant systemu sam dbał o zachowanie wewnętrznej spójności i kompatybilności wszystkich wymaganych przez standard Portable Executable struktur. Największy problem w takim przypadku stanowi układ sekcji i nagłówków oraz wartości różnego rodzaju adresów, które występują zarówno w samym kodzie programu jak i w niektórych sekcjach danych. Dodatkowym utrudnieniem jest potrzeba zachowania tej spójności zarówno   
w fizycznym pliku na dysku, jak i w obrazie programu załadowanym do pamięci. Jak zostało to przedstawione w rozdziale poświęconym formatowi PE, sekcje mają inne wyrównanie naturalne na dysku niż w pamięci operacyjnej.

Od pierwotnych faz tworzenia architektury systemu malware-scrambler brana pod uwagę była możliwość modyfikacji rozmiaru sekcji lub zmiany ich kolejności. Brak automatycznego dbania o spójność po każdej modyfikacji szybko doprowadziłby do niemożliwości ponownego uruchomienia programu, nawet po poprawnej serializacji do pliku wykonywalnego. Możliwe błędy mogłyby się pojawić zarówno na etapie ładowania programu do pamięci RAM jak i dopiero   
w czasie wykonywania.

Wszystkie wymienione wyżej problemy doprowadziły do powstania modułu   
o nazwie SectionResizer. Jego głównym zadaniem jest uaktualnienie wszystkich adresów sekcji na podstawie ich aktualnych rozmiarów. Pierwsza sekcja otrzymuje adres początkowy taki jaki był pierwotnie. Dla każdej kolejnej sekcji jej adres początkowy jest wyliczany poprzez sumę adresu początkowego sekcji poprzedniej oraz jej rozmiaru. Następnie wartość ta jest zaokrąglana w górę do najbliższej wielokrotności wyrównania sekcji. Proces ten odbywa się równolegle dla wyrównania w pliku oraz w pamięci.

Dodatkowo, silnik ten ma za zadanie uniemożliwić rozszerzanie lub zmniejszanie sekcji w przypadku, gdy nie jest zdefiniowana sekcja relokacji. Problem, którego w ten sposób można uniknąć polega na tym, iż w przypadku gdy jedna sekcja ma zmieniony rozmiar istnieje ryzyko, iż wprowadzona różnica wymusi przesunięcie wszystkich sekcji następujących po niej o kolejną wartość wyrównania naturalnego. Aby lepiej zobrazować tę sytuację, można wyobrazić sobie przykład, w którym wyrównanie sekcji w pamięci wynosi 1000. Oznacza to, iż każda sekcja musi rozpoczynać się pod adresem będącym wielokrotnością tej liczby. Niech sekcja A ma swój początek pod adresem 1000, a jej całkowity rozmiar niech wynosi 950 bajtów. Oznacza to, iż kolejna sekcja B musi rozpoczynać się najwcześniej pod adresem 2000 lub każdym większym będącym wielokrotnością liczby 1000. Dla ustalenia uwagi przyjmijmy, iż wybierana jest zawsze najmniejsza możliwa wartość (zgodnie z tą zasadą działa silnik SectionResizer). W takim przypadku, pomiędzy końcem sekcji A, a początkiem sekcji B występuje wolna przestrzeń o wielkości 50 bajtów. Zazwyczaj wypełniona jest ona samymi zerami. W takiej sytuacji, jeżeli sekcja A zostanie poszerzona o 25 bajtów to początek sekcji B pozostanie bez zmian. Jeżeli jednak rozmiar sekcji A zostanie zwiększony   
o wartość większą niż 50 to sekcja B będzie musiała zostać przesunięta do kolejnej wielokrotności liczby 1000. W tym przypadku będzie to adres o wartości 3000. Warto się zastanowić jakie potencjalne zagrożenia może ta zmiana powodować.

Najgroźniejszym z nich jest możliwe unieważnienie wielu adresów, które mogą występować w sekcji B. Dotyczy to adresów bezwzględnych, które wskazują miejsce w sekcji B i następnych oraz adresów bezwzględnych odnoszących się do sekcji A i wcześniejszych. W przypadku, gdy dostępna jest w pliku PE sekcja relokacji, możliwe jest odnalezienie pozycji wszystkich adresów oraz dodanie do nich wartości, o którą przesunięta została sekcja B. Jeżeli jednak nie ma zdefiniowanych relokacji, modyfikacja ta wprowadza nieodwracalne zmiany, które uniemożliwiają poprawnie wykonać program.

Z tego powodu, silnik mutacyjny SectionResizer sprawdza także, czy żadna   
z sekcji nie jest przesuwana w momencie, gdy nie są znane dane relokacji. Gdy występuje taka sytuacja, zgłaszany jest odpowiedni wyjątek i kończone jest dalsze modyfikowanie programu. Użytkownik jest powiadamiany, iż bieżąca konfiguracja programu nie pozwala na poprawne zmodyfikowanie danego pliku PE. Zatem   
w potoku mutacyjnym po każdym silniku, który potencjalnie zmienia układ lub rozmiar sekcji musi następować moduł SectionResizer. Jest to jedyny sposób, aby zachować pełną spójność programu oraz wykrywać w możliwie najwcześniejszym etapie modyfikacje niepoprawne.

#### Przenoszenie danych w obszarze pliku

Pierwotnie, niektóre zdefiniowane silniki mutujące zakładały potrzebę przeniesienia pewnego fragmentu kodu lub struktury PE w inne miejsce   
w pamięci. Funkcjonalność ta była wielokrotnie duplikowana w każdym miejscu użycia. Modyfikacja sprawiała więc problem zmiany w wielu miejscach kodu. Dodatkowo wprowadzało to do kodu silników sporo logiki, która nie miała nic wspólnego z podstawową ideą modyfikacji. Z tego powodu zdecydowano się na uogólnienie tej idei i utworzenie generycznego modułu o nazwie CodeRelocationEngine. Implementacja sprowadza się do odczytania adresu źródłowego i docelowego przenoszonych danych oraz ich rozmiaru oraz fizycznym przekopiowaniu ich. Obu adresom przyporządkowywane są sekcje, które je obejmują. Konfiguracja silnika możliwa jest na dwa sposoby:

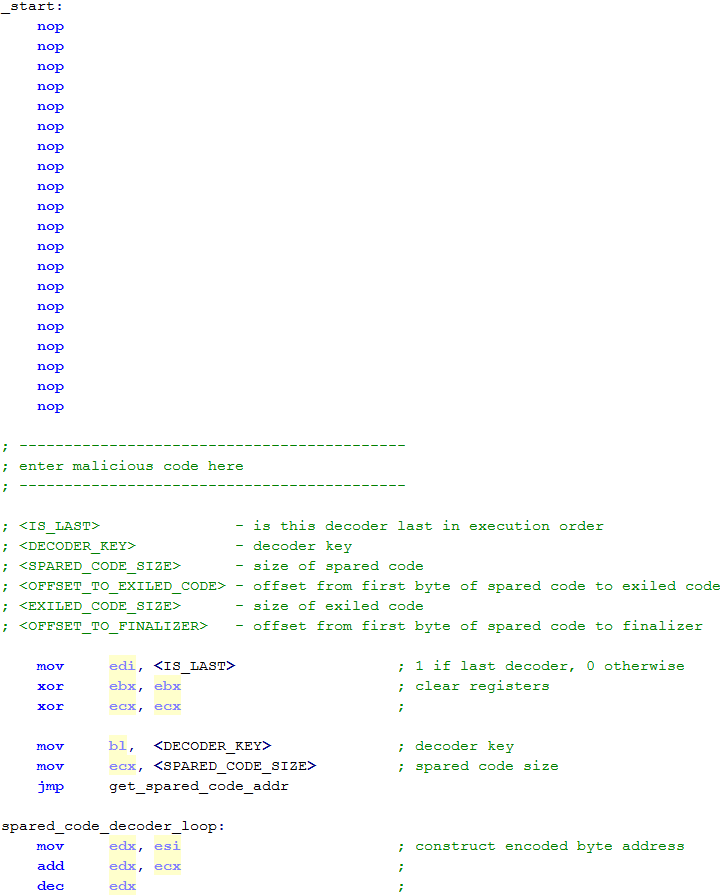
* poprzez wartości przekazane z innych silników przez kontekst mutacji
* poprzez obliczenie maksymalnego rozmiaru kodu wynikowego, który może powstać w wyniku działania modułu pomocniczego CodeProvider (opisany w dalszej części podrozdziału) i przekopiowaniu tej ilości   
  z początku sekcji kodu na koniec ostatniej sekcji.

Dzięki temu, silnik ten bardzo dobrze współpracuje z innymi modułami mutującymi i nie wymaga żadnych założeń lub ograniczeń w sposobie jego użycia. W niektórych przypadkach wymagane jest, aby przekazać następującym później silnikom wirtualny adres względny miejsca docelowego. Wartość ta również przekazywana jest poprzez kontekst.

#### Generacja kodu wykonywalnego

Większość zaprojektowanych silników mutacyjnych wymaga poza modyfikacją struktur pliku PE umieszczenie w ustalonym miejscu dodatkowego kodu wykonywalnego. Przykładem takiego przypadku może być potrzeba rozszerzenia pliku o kod dekodera, w przypadku gdy mutacją jest szyfrowanie kodu. Potrzebny kod musi być zgodny z architekturą komputera, na której będzie później wykonywany po uruchomieniu. Dodatkowym utrudnieniem jest fakt, iż musi on być dostępny w postaci binarnej, czyli po etapie kompilacji. Należało zatem opracować metodę generowania kodu maszynowego na podstawie jednego   
z istniejących języków programowania. Wprowadza to z kolei problem wygenerowania tylko i wyłącznie tego kodu, który został napisany przez użytkownika. Wiadomo bowiem, iż skompilowane programy, nawet jako pliki obiektowe przed etapem linkowania, zawierają wiele innych informacji, które są zbędne w przypadku doklejania do istniejącego kodu. Biorąc pod uwagę zdefiniowane wymagania zaproponowana została metoda, która umożliwia pisanie instrukcji w językach wysokiego poziomu (jak C lub C++) oraz w języku Assembler, przy jednoczesnym okrojeniu generowanych danych wyjściowych wyłącznie do binarnej postaci napisanych instrukcji. Dalsze prace   
z modułem podczas wykonywania eksperymentów potwierdziły jej prostotę, poprawność i przydatność.

Rozwiązanie polega na umieszczeniu wewnątrz generowanego kodu maszynowego pewnego wzorca binarnego, identyfikującego początek i koniec kompilowanych instrukcji. Jako łatwą do wykrycia w wynikowym pliku oraz dostępną we wszystkich zdefiniowanych wcześniej językach metodą okazało się umieszczanie fragmentów ustalonego kodu assemblera na początku i końcu pozostałych instrukcji. Dodatkowo wstawiane bloki składają się z 20 instrukcji *nop*. Wybór takiego wzorca podyktowany był trzema właściwościami. Pierwsza z nich to taka, że instrukcja *nop* (ang. no operation) nie ma żadnych efektów ubocznych wykonania. Nie zmieniają się wartości rejestrów ogólnego przeznaczenia, ani rejestr flag procesora. Oznacza to, iż instrukcję tą można umieścić w dowolnym miejscu kodu. Druga właściwość dotyczy długości binarnej reprezentacji tej instrukcji. Jest ona równa dokładnie jednemu bajtowi, co oznacza, iż można umieszczać w kodzie sekwencję bajtów o dowolnej długości nie powodujących żądnych efektów ubocznych (w przypadku gdyby długość instrukcji *nop* była inna niż 1, to sekwencja ta musiałaby być wielokrotnością tej liczby). Trzecią decyzją jaką należało podjąć był wybór długości wstawianego wzorca. Niektóre kompilatory używają tej samej instrukcji do wyrównywania generowanego kodu do wielokrotności 16. Z tego powodu uznano, iż każda długość wzorca powyżej tej liczby jest dobra i nie powinna być kiedykolwiek pomylona z kodem kompilatora. Arbitralnie wybrano liczbę 20 jako spełniającą opisane założenie.

 W systemie malware-scrambler został zaprojektowany specjalny moduł   
o nazwie CodeProvider, który odczytuje z pliku konfiguracyjnego ścieżkę do szablonu z kodem zawierającym kod użytkownika oraz wyżej opisane bloki wzorców. Przykładowy fragment takiego pliku znajduje się na Rysunku 19.

Rysunek 19. Fragment szablonu kodu modułu CodeProvider

Dodatkowo wczytywane są ustawienia dotyczące wybranego przez użytkownika kompilatora oraz flag kompilacji. Następnie formułowane jest polecenie systemowe, które po wywołaniu uruchamia zdefiniowany kompilator   
i przekazuje mu ścieżkę do pliku źródłowego. Wygenerowany plik obiektowy zapisywany jest pod taką samą nazwą jak szablon, ale z rozszerzeniem obj. Dzięki temu zawsze wiadomo gdzie znajdują się wyniki kompilacji. Kolejnym etapem jest odczytanie zawartości pliku obiektowego oraz wyszukanie dwóch bloków 20 instrukcji *nop*, z których każda ma wartość 90 w systemie heksadecymalnym. Ustalenie pozycji ostatniego bajtu pierwszego bloku oraz pierwszego bajtu drugiego bloku wyznacza początek i koniec wygenerowanych instrukcji użytkownika.

Mimo, iż opracowanie sposobu generowania kodu binarnego w sposób automatyczny jest niewątpliwym ułatwieniem dla użytkownika systemu, to nie rozwiązuje to problemu dostosowywania kodu generycznego do konkretnych zastosowań. Istnieje bowiem wiele przypadków, w których raz napisany kod działający dla jednego pliku będzie tak samo dobrze działał dla drugiego, po skorygowaniu pewnych stałych liczbowych. Przykładem może być zasięg instrukcji skoku lub ilość iteracji pętli. Aby umożliwić automatyczne definiowanie tych wartości przez sam program malware-scrambler w trakcie działania, wprowadzone zostały specjalne znaczniki, które po odczytaniu szablonu są zamieniane na konkretne wartości liczbowe. W ten sposób użytkownik może napisać jeden generyczny kod, a system zajmie się dostosowywaniem go dla każdego pliku wykonywalnego, który będzie poddawany modyfikacjom. Na Rysunku 18 można zauważyć kilka takich znaczników. Ich nazwy są ujęte   
w nawiasy kątowe „<” i „>” oraz składają się jedynie z wielkich liter oddzielonych znakami podkreślenia. Przykładowo, Znacznik <DECODER\_KEY> oznacza klucz, jaki będzie używany przy dekodowaniu. Jego wartość definiowana jest w sekcji silnika szyfrującego znajdującej się w pliku konfiguracyjnym. Implementacja modułu szyfrującego wczytuje tą wartość, a następnie odnajduje w szablonie kodu zdefiniowany znacznik i podmienia jego wartość na tą z konfiguracji.

Moduł CodeProvider z punktu widzenia architektury nie jest osobnym silnikiem mutującym, a jedynie elementem pomocniczym, który może zostać wykorzystany przez inne silniki. Z tego powodu jego samodzielne występowanie   
w definicji potoku mutacyjnego jest nieprawidłowe. Zostały zaimplementowane specjalne funkcje, które analizują strukturę potoku i wykrywają taką sytuację.

### Szyfrowanie

W rozdziale trzecim opisane zostały rozmaite metody i algorytmy, które są wykorzystywane w programach antywirusowych do wykrywania obecności i typu złośliwego oprogramowania. W większości opierają się one na znajomości struktury budowy malware-u. Popularnym sposobem jest tworzenie bazy sygnatur. Przy każdym skanowaniu liczona jest sygnatura podejrzanego pliku   
i porównywana z wpisami w bazie danych. Jeżeli istnieje jakaś zgodność, plik uznawany jest szkodliwy. Czasami jednak klasyfikowany jest tylko jako podejrzany i kierowany jest na dalsze testy bardziej zaawansowanymi metodami. Wadą tego podejścia jest brak odporności na nawet drobne modyfikacje kodu wirusów. Często zamiana kolejności instrukcji może wpływać na wynik analizy.

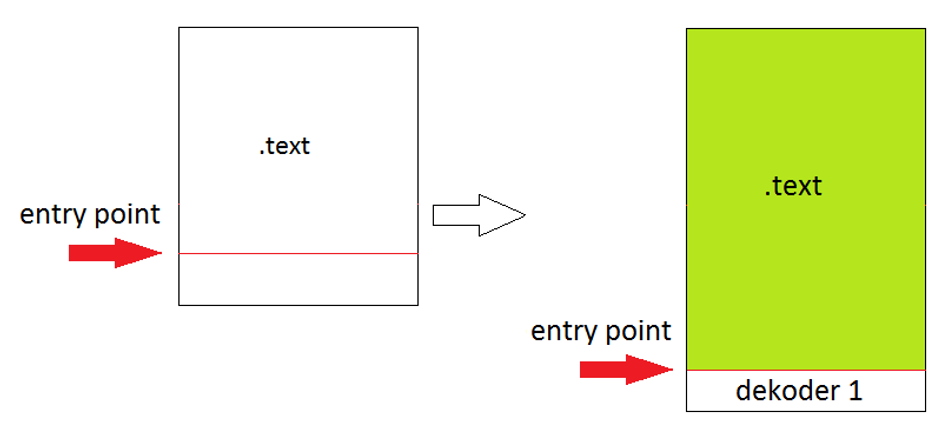
Z tego powodu zdecydowano się na opracowanie silnika, który będzie szyfrował kod plików wykonywalnych oraz doklejał w ustalonym miejscu kod odpowiednio kompatybilnego dekodera. W założeniu przy uruchamianiu tak zmodyfikowanego pliku sterowanie najpierw przekazane będzie do dekodera, który odszyfruje wszystkie dane, a następnie skoczy do oryginalnego punktu wejścia aplikacji. Dzięki temu niemożliwe będzie zastosowanie metody sygnatur lub podobnych do identyfikacji wirusa. Peter Shor w swojej książce [15] opisuje, iż znane są sposoby statycznego wykrywania kodu deszyfratorów. Mimo to, autor systemu stwierdził, iż koszt takich statycznych analiz może powodować dość duże opóźnienia w uruchamianiu aplikacji i przez to może nie być implementowany   
w każdym produkcie antywirusowym. Dodatkowo, nie wiadomo jaka jest skuteczność wykrywania tego typu modyfikacji w przypadku, gdy szyfrowanie pliku jest wielokrotne.

Najpopularniejszą metodą szyfrowania jest wykonanie operacji XOR na każdym bajcie kodu wykonywalnego z ustaloną wartością liczbową nazywaną dalej kluczem. Celem dekoder jest odtworzenie instrukcji przy znajomości jedynie klucza. Operacja XOR ma ciekawą właściwość, która polega na tym, że gdy na wyniku poprzedniej operacji XOR wykona się drugą operację tego typu z tym samym kluczem to otrzyma się ponownie pierwotną wartość liczbową. Przykładowo, jeżeli szyfrowanie polega na wykonaniu funkcji XOR z liczbą 12, to deszyfrowanie również będzie polegać na operacji XOR z liczbą 12. Istnieją również inne metody szyfrowania, jak np. dodawanie ustalonej liczby do każdego bajtu oraz późniejsze odejmowanie jej przy deszyfrowaniu. Zdecydowano się jednak na opcję z XOR, gdyż z doświadczenia autora projektu w pracy z kodem exploitów ten typ ukrywania obecności złośliwego kodu jest najpopularniejszy.

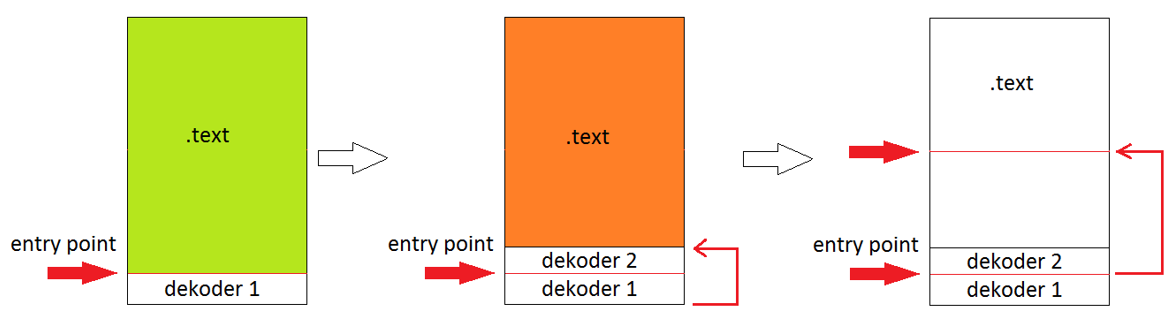
W systemie malware-scrambler zdefiniowane zostały dwa podejścia do szyfrowania plików wykonywalnych. Oba wspierają ideę wielokrotnego szyfrowania danych. Różnica między nimi polega na umiejscowieniu kodu deszyfratora.

#### Deszyfrator doklejany

Jak zostało to opisane powyżej, szyfrowanie w opracowywanej aplikacji sprowadza się do wykonania operacji XOR z ustalonym przez użytkownika kluczem. Odczytywany jest on z pliku konfiguracyjnego dla każdego wystąpienia silnika szyfrującego osobno, przez co wielokrotne szyfrowanie może być realizowane przy pomocy wielu różnych kluczy. Zwiększa to prawdopodobieństwo niewykrycia obecności kodu wirusa.

Pierwszym etapem jest ustalenie początku kodu wykonywalnego, który ma być zaszyfrowany. Zadanie to nie jest trywialne, gdyż sekcja kodu niekoniecznie musi zawierać tylko kod maszynowy. Istnieje wiele kompilatorów, które umieszczają w sekcji kodu również informacje na temat tablicy importów lub innych danych potrzebnych do załadowania i wykonania programu. Z tego powodu opracowana została specjalna klasa, której zadaniem jest ustalenie początku i końca kodu wykonywalnego w ramach danej sekcji kodu. Po ustaleniu granic szyfrowania silnik mutujący w pętli przeprowadza operację XOR na każdym bajcie kodu. Następnie odczytywany jest kod szablonu dekodera i podmieniane są w nim wszystkie znaczniki specyficzne dla tego silnika. Przykładem może być klucz szyfrujący lub rozmiar ukrytego kodu. W kolejnej fazie uruchamiany jest moduł CodeProvider, który generuje kod binarny i zwraca go do silnika szyfrującego   
w postaci wektora bajtów. Na koniec silnik mutujący dodaje tak utworzy ciąg kodu maszynowego na koniec sekcji kodu oraz ustawia punkt wejścia programu na początek doklejanego deszyfratora. Porównanie sekcji kodu przed i po modyfikacji przedstawione jest na Rysunku 20. Fakt zmiany wartości EP sprawia, iż deszyfrator musi umieć samodzielnie powrócić do oryginalnego punktu startowego modyfikowanej aplikacji. Również i ta właściwość implementowana jest przy pomocy podmieniania znaczników w szablonie. W przypadku szyfrowania wielokrotnego tylko ostatni (w kolejności wykonywania) silnik mutujący musi skakać do oryginalnego EP. Wszystkie inne muszą przekazywać sterowanie do silnika, który fizycznie jest tuż nad nim. Z tego wynika, iż silniki te muszą znać długość zarówno swoją jak i silnika następnego. Obie wartości zsumowane dają   
w wyniku przesunięcie jakie musi być wykorzystane w instrukcji krótkiego skoku, aby kolejny deszyfrator mógł rozpocząć pracę. Przypadek deszyfrowania wielokrotnego obrazuje Rysunek 21.

Rysunek 20. Efekt szyfrowania jednokrotnego z doklejaniem deszyfratora



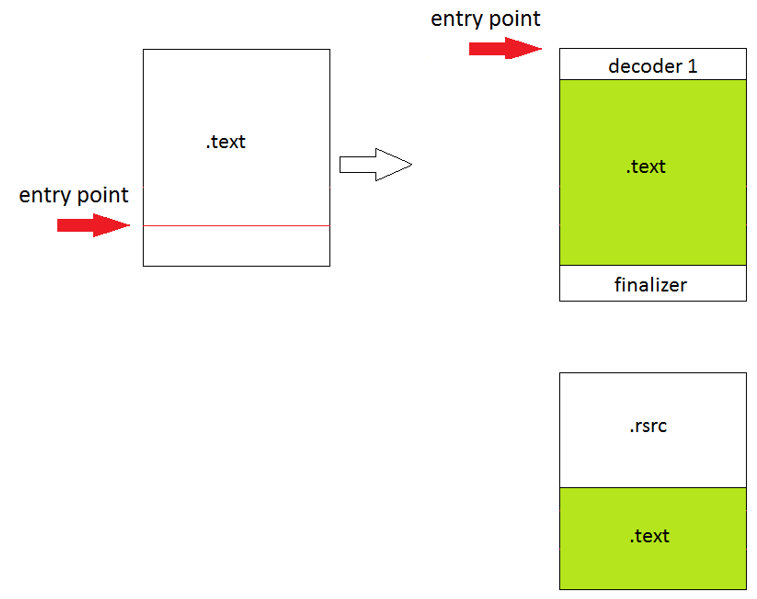
Rysunek 21. Deszyfrowanie wielokrotne

Jak widać, fakt istnienia kolejnego deszyfratora ujawnia się dopiero po zakończeniu pracy poprzedniego. Dzięki temu programy antywirusowe nie są   
w stanie ustalić z góry ile razy należy skanować plik w poszukiwaniu wzorca deszyfratorów. Po wykonaniu kodu dekodera 1 następuje krótki skok o tyle bajtów w stronę mniejszych adresów ile ma suma długości dekodera 1 i 2. Dekoder 2 jest ostatnim w kolejności wykonywalnym deszyfratorem, więc po zakończeniu jego pracy następuje skok do oryginalnej wartości punktu wejścia programu. Każdy z szyfratorów używa innego klucza dla operacji XOR, co na rysunku zobrazowane jest w postaci koloru danych zaszyfrowanych.

#### Deszyfrator nadpisujący

Opisany powyżej deszyfrator doklejający ma jedno poważne ograniczenie. Związane jest ono z ilością wolnej przestrzeni pomiędzy sekcją kodu i sekcją znajdującą się bezpośrednio za nią. Każdy z doklejanych deszyfratorów zajmuje pewną liczbę bajtów. Doklejane ich powoduje wypełnianie opisywanej wcześniej wolnej przestrzeni pomiędzy sekcjami. W przypadku, gdy łączna ilość kodu deszyfratorów jest większa niż ilość dostępnego miejsca musi nastąpić przesunięcie drugiej sekcji. Problem ten został opisany w podrozdziale o module SectionResizer. Zostało także powiedziane, iż taka operacja jest zabroniona. To bezpośrednio implikuje fakt, iż liczba możliwych do doklejenia deszyfratorów jest   
z góry ograniczona. Dodatkowo, ilość ta jest zawsze różna dla każdego modyfikowanego pliku. Stanowiłoby to zatem problem przy próbie porównania jakości wprowadzanych modyfikacji przy wielokrotnym szyfrowaniu, gdyż nie dałoby się z góry przewidzieć wystarczająco dużej liczby deszyfratorów dających się dokleić do wszystkich posiadanych próbek.

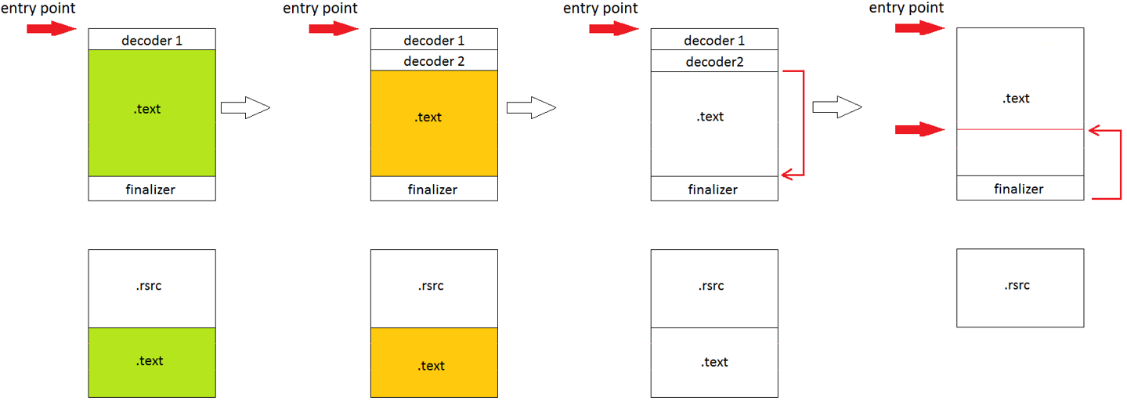
Z tego powodu opracowany został drugi moduł, który odpowiedzialny jest za szyfrowanie sekcji kodu. W tym przypadku brane było pod uwagę wymaganie, aby niezależnie od liczby szyfrowań sekcji kodu nie trzeba było przemieszczać sekcji znajdujących się bezpośrednio za nią. Opcja doklejania na końcu sekcji została zweryfikowana przez poprzedni moduł szyfrujący. Z kolei doklejanie na początku sekcji prowadzi do dokładnie tego samego problemu. Jedynym rozwiązaniem, które nie wprowadza dodatkowych ograniczeń na szyfratory jest nadpisywanie ustalonych fragmentów sekcji kodem deszyfratorów. Dzięki temu, jeśli wstawianie deszyfratorów rozpoczęte zostanie od pierwszego bajtu sekcji kodu, to ilość dostępnego miejsca na kolejne dekodery będzie równa pojemności tej sekcji.

Zaproponowane rozwiązanie wprowadza jednak problem odzyskiwania pierwotnej zawartości sekcji kodu. Po zakończeniu deszyfrowania kod dekoderów jest zbędny i powinien zostać zamieniony ponownie na oryginalne dane. Z tego powodu wprowadzone zostały dwa kolejne silniki mutujące. Zadaniem pierwszego jest oszacowanie wielkości wszystkich dekoderów i innych silników, które będą nadpisywały fragmenty sekcji kodu. Dokonywane jest to poprzez analizę wszystkich instancji modułu CodeProvider oraz próbnej kompilacji ich kodu. Słowo próbnej w tym przypadku oznacza, iż wszystkie znaczniki pojawiające się   
w szablonach z kodem zamieniane są na dowolną liczbę, której reprezentacja binarna zajmuje dokładnie tyle bajtów ile jest w danym miejscu oczekiwane. Dla przykładu, jeżeli w kodzie assemblera oczekiwana jest liczba która zajmie w pliku obiektowym 4 bajty, to moduł szacujący rozmiar silników wstawi liczbę, która wymaga dokładnie 32 bitów do zakodowania. W przeciwnym wypadku kompilator lub assembler mógłby dokonać optymalizacji, które spowodowałyby różnicę pomiędzy rzeczywistą wielkością kodu silnika, a tą oszacowaną. Następnie po uzyskaniu łącznej wielkości kodu nadpisującego, silnik kopiuje wyliczoną liczbę bajtów z początku sekcji kodu i umieszcza ją na końcu ostatniej sekcji w danym pliku wykonywalnym. Operacja ta nie ma żadnych ograniczeń, ponieważ ostatnia sekcja nie ma żadnego następnika, którego trzeba byłoby przesuwać. Na koniec do kontekstu mutacji wprowadzany jest klucz i wartość informujące pozostałe silniki   
o ilości skopiowanych danych. Zadaniem drugiego wprowadzonego silnika jest wygenerowanie kodu, który po zakończeniu działania pozostałych silników skopiowałby zachowane oryginalne dane z końca ostatniej sekcji ponownie na miejsce obecnie zajmowane przez dodane dekoderów. To oznacza, iż sam kod kopiujący nie może być umiejscowiony razem z innymi silnikami, gdyż   
w przeciwnym wypadku w pewnym momencie musiałby zacząć nadpisywać samego siebie. Z bardzo dużą dozą prawdopodobieństwa spowodowałoby to błąd wykonania programu. Zdecydowano się, iż kod modułu kopiującego będzie zawsze doklejany na końcu sekcji kodu, tak samo jak pierwsza wersja silnika szyfrującego. Ogólny schemat przedstawiający wygląd pliku przed i po zastosowaniu tej mutacji przedstawia Rysunek 22.

Rysunek 22. Efekt szyfrowania sekcji kodu metodą nadpisywania

Jak można zauważyć, po dokonaniu mutacji silnikiem szyfrującym   
z nadpisywaniem, na początku sekcji kodu pojawił się kod pierwszego deszyfratora. Dodatkowo, na końcu ostatniej sekcji w danym pliku (w tym przypadku sekcji zasobów) pojawiły się dane sekcji kodu, które zostały nadpisane przez wstawiane deszyfratory. Dla łatwiejszej identyfikacji fragmentów kodu wprowadzona została prosta nomenklatura. Dane, które nie są nadpisywane nazywane się kodem oszczędzonym (ang. spared), zaś dane skopiowane na koniec ostatniej sekcji nazywane są kodem wygnanym (ang. exiled). Kolor na obrazku sugeruje, iż zarówno kod oszczędzony jak i wygnany są szyfrowane. Musi to być uwzględnione przy projektowaniu szablonu kodu dekodera. Zmieniony zostaje również punkt wejścia w danym pliku.

Proces deszyfrowania kodu, w którym zastosowane zostało wielokrotne deszyfrowanie dekoderem nadpisującym przedstawia Rysunek 23.

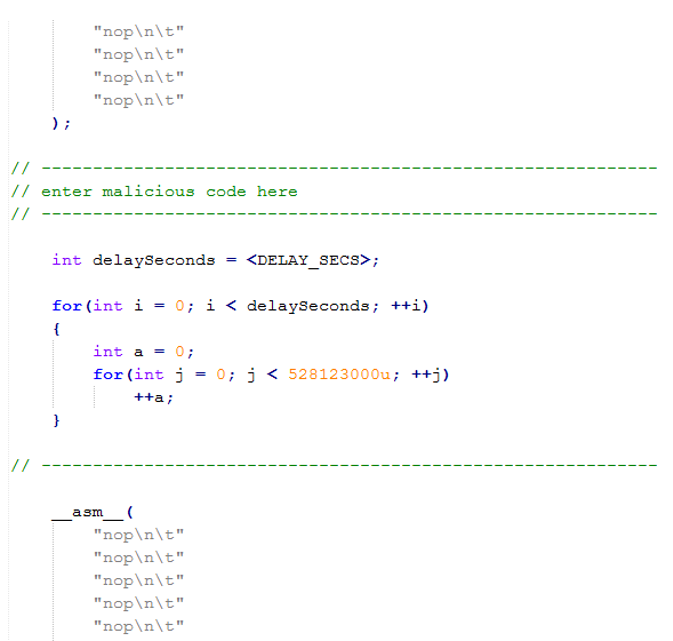


Rysunek 23. Proces deszyfrowania wielokrotnego deszyfratorem nadpisującym

Dla ustalenia uwagi przyjmijmy, iż wszystkie silniki nadpisujące dane sekcji kodu to moduły szyfrujące. W takim wypadku, na początku wykonywany jest pierwszy dekoder. Wykonuje od deszyfrowanie kodu oszczędzonego (którego rozmiar jest równy wielkości sekcji kodu pomniejszonej o rozmiar pierwszego dekodera). Następnie wykonywane jest deszyfrowanie danych wygnanych, których rozmiar przekazany został do silnika generującego kod poprzez kontekst mutujący. Dzięki temu deszyfrator mógł podmienić odpowiedni znacznik w szablonie kodu na prawidłową wartość oznaczającą rozmiar kodu do deszyfrowania. Po zakończonej pracy kod deszyfrujący sprawdza specjalną flagę zapisaną w samym sobie, oznaczającą czy jest ostatnim silnikiem nadpisującym. Jeżeli nie, to sterowanie przekazywane jest do adresu, który znajduje się bezpośrednio za bieżącym silnikiem, gdyż tam oczekiwany jest kod następnego modułu. Wtedy ponownie dokonywane jest deszyfrowanie kodu oszczędzonego oraz wygnanego (przy czym rozmiar tego pierwszego jest odpowiednio mniejszy od wartości używanej   
w poprzednim silniku). W przypadku, gdy stan wspomnianej flagi informuje, iż jest to ostatni silnik nadpisujący, następuje wykonanie skoku pod ustalony adres, pod którym oczekiwany jest kod silnika kopiującego. Ten z kolei, znając adres źródłowy i docelowy kodu wygnanego kopiuje każdy bajt, nadpisując tym razem kod dekoderów. Po zakończonej pracy, kod sekcji powinien być identyczny   
z zawartością pliku wykonywalnego bez modyfikacji. Następuje wtedy skok do oryginalnej wartości punktu wejścia programu. Dzięki temu aplikacja może kontynuować swoje normalne wykonanie.

### Wstawianie martwego kodu

W rozdziale opisującym znane z literatury metody wykrywania obecności złośliwego kodu wspomniana została metoda analizy behawioralnej, polegająca na czasowej emulacji wykonania danego pliku wykonywalnego w tzw. piaskownicy (ang. sandbox). Podczas emulacji dokonywana jest analiza wykonywanych przez aplikację instrukcji, żądań dostępu do zasobów systemowych, wywoływanych funkcji i innych widocznych skutków ubocznych działania. Doświadczalnie zostało przez projektanta sprawdzone, iż popularne darmowe skanery antywirusowe stosują emulację działania programu trwającą od 10 do 15 sekund. Po jej zakończeniu program jest ponownie uruchamiany, tym razem bez żadnej kontroli.

W systemie malware-scrambler opracowany został specjalny moduł   
o nazwie GarbageCodeInjector, którego jedynym celem jest wprowadzenie do modyfikowanego pliku specjalnego fragmentu kodu, który nie powoduje żadnych widocznych efektów wykonania, a jedynie zajmuje czas procesora. Celem jest wprowadzanie wymuszonego opóźnienia w wykonaniu prawdziwego kodu aplikacji. Dzięki temu, nawet jeżeli dany skaner antywirusowy uruchomi emulację wykonania pliku, to w teorii przez cały czas wykonywania pod kontrolą piaskownicy powinien wykonywać się wstrzyknięty kod śmieciowy (ang. garbage code). Silnik ten wykorzystuje technikę nadpisywania, opisaną przy okazji omawiania deszyfratora nadpisującego. Z tego powodu, nawet jeżeli zdefiniowany potok mutacyjny nie posiada wpisów o silnikach szyfrujących, to i tak wymagana jest obecność na końcu silnika kopiującego.

Rysunek 24. Szablon kodu opóźniającego

Definiowanie kodu opóźniającego wymaga dużej ostrożności w doborze wykonywanych instrukcji, aby nie wprowadzać widocznych efektów ubocznych wykonania. Popularną metodą wprowadzania aktywnych opóźnień czasowych, znaną głównie z programowania urządzeń wbudowanych takich jak mikrokontrolery, jest definiowanie pętli o dużej liczbie iteracji, w ciele której wykonywane są proste operacje matematyczne na kilku lokalnych zmiennych.   
Z uwagi na żmudność projektowania kodu wykonującego wymagane operacje w języku assembler, zdecydowano się na wykorzystanie języka wysokiego poziomu, np. C++. Przykładowy szablon kodu opóźniającego znajduje się na Rysunku 24. Można zauważyć, iż zaprojektowany kod zawiera specjalny znacznik, którego wartość oznacza żądaną liczbę sekund opóźnienia, jaka ma być przez ten kod wprowadzona. Liczba ta ustalana jest w pliku konfiguracyjnym w definicji potoku mutacyjnego. Autor projektu zdaje sobie sprawę, iż rzeczywiste opóźnienie może   
i zapewne będzie się różnić od założonego, gdyż liczba iteracji składająca się na jedną sekundę opóźnienia została dobrana arbitralnie na podstawie tylko kilku prób na jednym komputerze. Jednak celem silnika opóźniającego nie jest wierne odwzorowywanie ustawień programu. Możliwość konfiguracji czasu opóźnienia ma jedynie dać możliwość użytkownikowi zdefiniowanie rzędu wielkości wprowadzonego opóźnienia. Bardzo ważną kwestią, o której należy pamiętać   
w przy korzystaniu z każdego kompilatora do generowania opóźnień czasowych, jest wyłącznie wszelkich optymalizacji. W przeciwnym przypadku, analizator kodu wbudowany w kompilator może uznać, iż dany zestaw instrukcji nie ma żadnych efektów ubocznych i jego generacja jest zbędna. Przykładowo, dla kompilatora *g++* z pakietu GNU należy ustawić flagę *–O0*, aby całkowicie wyłączyć próby optymalizacji czasu wykonania generowanego kodu*.*

### Usuwanie tablicy IAT

Niektóre sposoby analizy wirusów komputerowych opierają się na identyfikowaniu wszystkich importowanych przez dany plik wykonywalny bibliotek i funkcji oraz porównywaniu ich ze znanymi wzorcami zachowań złośliwego oprogramowania. Brana pod uwagę jest też deklarowana natura oryginalnej aplikacji. Przykładem może być prosty edytor tekstu. Oczekiwane jest, iż będzie miał on wiele zależności od bibliotek związanych z wyświetlaniem okien   
i grafik na ekranie. Być może potrzebne będą pewne funkcje matematyczne do obliczania niektórych zależności liczbowych w formułach, zwanych często makrami. Co można zatem powiedzieć w momencie, gdy taka aplikacja potrzebuje dostępu do szerokiego wachlarza funkcji sieciowych? Wydaje się to całkiem podejrzane, gdyż jest niewiele przypadków użycia edytora tekstu, które wymagają dostępu do API sieciowego. Dzięki zautomatyzowaniu analizy importowanych bibliotek można w sposób statyczny wykrywać podejrzane pliki, które albo deklarują inną zawartość niż w rzeczywistości (tak jak to robią konie trojańskie) lub zostały zmodyfikowane przez replikację złośliwego oprogramowania (typowe zachowanie wirusów komputerowych).

Jednym ze sposobów przeciwstawienia się temu sposobu oznaczania plików jako podejrzane jest ukrycie prawdziwej natury złośliwego kodu poprzez usunięcie z pliku wykonywalnego wszelkich dowodów na importowanie bibliotek i funkcji, które są niezgodne z deklarowanym zachowaniem programu. Aby zapewnić temu rozwiązaniu generyczność postanowiono, iż najbardziej bezpiecznym jak   
i najprostszym rozwiązaniem będzie usuwanie wszystkich importowanych funkcji oraz ładowanie ich do pamięci operacyjnej w trakcie wykonywania programu przez samą aplikację, a nie program ładujący. Założeniem było wykorzystanie znanych z pracy dyplomowej inżynierskiej algorytmów odnajdywania potrzebnych adresów bazowych bibliotek oraz adresów importowanych funkcji, które są powszechnie wykorzystywane w exploitach. Aby w pełni zrozumieć kod ładujący biblioteki i funkcje do programu należy najpierw zapoznać się z podstawowymi technikami, jakie są wykorzystywane we wspomnianych exploitach. Dla ustalenia uwagi, opisany zostanie podstawowy zestaw akcji, jakie zewnętrzny kod musi wykonać w celu dynamicznego doładowania bibliotek. Jednak jako wyczerpujący punkt referencyjny polecany jest rozdział 2.2 z pracy dyplomowej inżynierskiej „Analiza podobieństwa shellcode’u z wykorzystaniem grafu przebiegu jego wykonania” [18].

Pierwszym etapem jest zazwyczaj deszyfrowanie kodu. W przypadku mutacji programem malware-scrambler czynność ta jest wykonywana przez inne moduły. Następnie potrzebne jest ustalenie adresu bazowego biblioteki kernel32.dll. Jest ona szczególna, gdyż jest ładowana automatycznie do przestrzeni adresowej każdego procesu, bez względu na to jak wygląda jego sekcja importów. Dodatkowo, system operacyjny zapewnia, iż w większości przypadków adres bazowy tej biblioteki będzie możliwy do odczytania w ustalonej z góry strukturze   
o wiadomym adresie. Od reguły tej jest jednak wyjątek w postaci nowej rodziny systemów operacyjnych Windows o numerach wersji 8 i wyższych. W ich przypadku, gdyby zastosować dokładnie tą samą metodę, otrzymalibyśmy adres bazowy biblioteki KernelBase.dll. Mimo, iż nazwa sugeruje podobieństwo do szukanej kernel32.dll, to jej zestaw eksportowanych funkcji jest nieco inny.   
W przypadku wstrzykiwanego dynamicznie kodu główną funkcją, której adres należy znaleźć jest LoadLibraryA. Dzięki niej, można w prosty sposób załadować   
i otrzymać adres bazowy dowolnej biblioteki, której nazwę przekaże się jako parametr wywołania. Jedynym ograniczeniem jest możliwość odnalezienia żądanej biblioteki w lokacjach dyskowych określonych przez zmienną środowiskową PATH. W bibliotece KernelBase.dll dostępna jest jedynie funkcja LoadLibraryExA, która praktycznie wykonuje tą samą czynność, jednak wymaga innego i większego zestawu parametrów wywołania. Nie stanowi to żadnego problemu, jeżeli atakujący dokona założenia co do wersji systemu używanej na atakowanych maszynach. Może również dokonywać sprawdzania i wywoływać różny kod w zależności od dostępnego środowiska, jednak zazwyczaj rezygnuje się z tego ze względy na fakt, że kod exploitów powinien mieć jak najmniejszy rozmiar w celu uniknięcia wykrycia. Po załadowaniu i otrzymaniu adresu nowej biblioteki, następuje proces iteracji po sekcji eksportów oraz porównywania nazw dostępnych funkcji z nazwą żądaną. W celu zachowania jak największej wydajności oraz aby uniknąć przechowywania stałych łańcuchów znakowych wewnątrz kodu, porównywanie dotyczy skrótów (ang. hash) tych nazw. Nie ma większego znaczenia jaka metoda zostanie wybrana do obliczania skrótów. Ważne, aby była taka sama zarówna dla funkcji danej biblioteki oraz funkcji, która jest aktualnie szukana. Typową metodą, spotykaną w próbkach opisanych w [18], jest akumulacja wartości ASCII poszczególnych znaków wraz z bitowym przesunięciem cyklicznym o 13 pozycji. Silnym argumentem za wykorzystaniem właśnie tego sposobu jest prostota oraz wsparcie ze strony assemblera x86. W momencie, gdy przy porównaniu obie wartości są takie same sprawdzany jest numer porządkowy danej nazwy, który używany jest jako indeks w tablicy eksportów (EAT). W ten sposób odszukany został adres funkcji, który można zapisać lub bezpośrednio przekazać do instrukcji *call*, po wcześniejszym załadowaniu na stos odpowiednich argumentów wywołania.

Podobnie jak w przypadku poprzednich silników mutacyjnych, wykorzystany został specjalny szablon z instrukcjami w języku assembler oraz odpowiednimi znacznikami, uzupełnianymi podczas generacji kodu dla każdego pliku osobno. Po wstrzyknięciu i uruchomieniu dodatkowych poleceń maszynowych, pierwszym etapem działania jest ustalenie adresów oryginalnego oraz bieżącego miejsca sekcji importów. Wygnany kod jest, podobnie jak w przypadku deszyfratora nadpisującego, zaszyfrowany metodą XOR, dlatego następnym etapem jest jego deszyfrowanie wraz z kopiowaniem do pierwotnej lokacji. Dzięki temu, bieżący proces ma ponownie prawidłową sekcję importów. Nie ma jednak podstawionych rzeczywistych adresów funkcji. Jej stan jest dokładnie taki sam jak w momencie przechowywania na dysku. Z tego powodu wszystkie następne akcje są pewnego rodzaju emulacją działania programu Windows Loader, który w normalnych warunkach ładuje i aktualizuje wymagane funkcje przy starcie programu w oparciu o sekcję importów.

Kolejnym krokiem jest obliczenie adresu biblioteki kernel32.dll oraz funkcji LoadLibraryA. W tym przypadku wykorzystany został fragment kodu znany   
z exploitów bez większych modyfikacji. Wprowadza to ograniczenie na środowisko uruchomieniowe tak zmodyfikowanego pliku (opisane wcześniej), jednak od samego początku zaplanowane było, iż próbki testowane będą na maszynie wirtualnej będącej pod kontrolą systemu Windows XP. Odnaleziony adres funkcji ładującej biblioteki współdzielonej jest następnie zapisywany do późniejszego wykorzystania. Kolejnym etapem jest iteracja po liście struktur o elementach typu IMAGE\_IMPORT\_DESCRIPTOR, która jest głównym trzonem sekcji importów. Dla każdej z nich obliczane są adresy nazwy wymaganej biblioteki, tablicy IAT oraz INT. Następnie wywoływana jest funkcja LoadLibraryA, do której przekazany jest znaleziony RVA nazwy biblioteki. W wyniku otrzymujemy adres bazowy biblioteki, która od tej pory jest na stałe załadowana do pamięci bieżącego procesu. W tym momencie następuje kolejna pętla iteracji, tym razem po strukturach typu IMAGE\_THUNK\_DATA, które jednoznacznie określają nazwę funkcji lub jej numer porządkowy. Aby sprawdzić, który wariant jest używany należy sprawdzić najbardziej znaczący bit wskaźnika znajdującego się w tej strukturze. Jeżeli jest ustawiony, to pozostałe bity oznaczają numer porządkowy. W przeciwnym wypadku jest to adres względny nazwy importowanej funkcji. Dla ustalenia uwagi, opisany dalej zostanie wariant drugi, gdyż importowanie przez liczbę porządkową jest uogólnieniem importowania przez nazwę (porównywanie nazw ma na celu odnalezienie odpowiedniej liczby porządkowej). Po uwzględnieniu powyższego założenia, kolejnym etapem jest obliczenie skrótu nazwy importowanego symbolu (symbol jest uogólnieniem importowanego typu danych, często używanego przy pracy z bibliotekami dynamicznymi zarówno w systemie Windows jak i Linux). Wybrany został algorytm z akumulacją i przesunięciem bitowym. Następnie wywoływana jest funkcja, która dla podanego skrótu oraz adresu bazowego biblioteki zwraca odpowiedni adres szukanej funkcji. Ten fragment kodu również został zaczerpnięty z implementacji exploitów z pewnymi modyfikacjami. Wykonuje on prostą iterację po sekcji eksportów danej biblioteki, oblicza skrót nazwy wszystkich eksportowanych symboli oraz zwraca adres tego, którego skrót jest identyczny z tym przekazanym do funkcji. W przypadku niepowodzenia program wyświetla komunikat o błędzie Jest to jednak sytuacja wyjątkowa   
i świadczy o błędzie w oryginalnym pliku wykonywalnym. W momencie, gdy do głównej pętli kodu zwrócony zostaje adres szukanej funkcji, jedyne co należy zrobić to odnaleźć odpowiednie miejsce w tablicy importów (IAT) oraz nadpisać dany rekord obliczonym adresem.

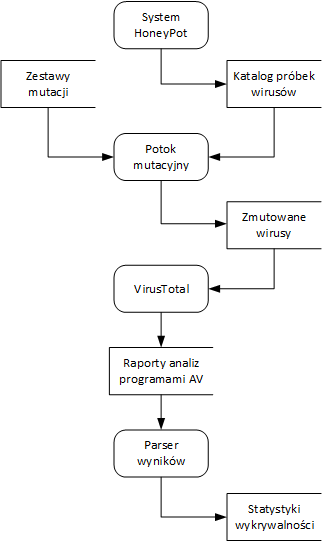
Opisany algorytm ma jednak jedno odstępstwo, które mimo, iż występuje dość rzadko, nie może zostać pominięte. W rozdziale drugim, przy okazji opisywania sekcji eksportów opisany został sposób tzw. eksportowania przekierowującego. Wymusiło to na projektancie systemu dopisania sprawdzania, czy adres obliczonej funkcji jest wewnątrz sekcji eksportów w doładowanej bibliotece. Jeżeli tak, to adres ten wskazuje na łańcuch znaków o formacie „<Nazwa biblioteki>.<Nazwa funkcji>”. Należy wtedy odpowiednio wydzielić nazwę biblioteki i funkcji oraz wykonać bardzo podobną operację wyszukiwania, jak w przypadku danych ze struktury IMAGE\_IMPORT\_DESCRIPTOR. Niestety wymaganie to zostało odkryte bardzo późno, więc zmiana wprowadziła sporo dodatkowego kodu, który musi być wygenerowany. Autor jest pewny, iż da się wykorzystać kod do wyszukiwania z pierwszego etapu, jednak ograniczenia czasowe wymusiły niestety podejście wprowadzające pewną redundancję, bez jednoczesnej modyfikacji (potencjalnie destruktywnej) istniejącego kodu.

## Weryfikacja poprawności działania

Aby można było uznać zaprojektowane modyfikacje plików jako wiarygodne należało wykonać serię testów, które w sposób jednoznacznie udowodniłyby poprawność działania zmutowanego pliku oraz niezmienność jego zachowania. Typowo, spośród docelowej grupy próbek wirusów komputerowych, na których przeprowadzane będą eksperymenty, powinien zostać wybrany reprezentatywny zestaw, który posłuży weryfikacji wprowadzanych zmian. Jednak przy pracy   
z grupą plików niebezpiecznych, jak wirusy komputerowe, należy zachować szczególną ostrożność i rozwagę przy uruchamianiu, czy nawet kopiowaniu ich   
z jednej lokacji dyskowej do drugiej. Z tego powodu, autor zdecydował się na bezpieczniejszą formę weryfikacji, polegającą na testowaniu działania zwykłych programów systemowych po poddaniu ich modyfikacjom. Takie podejście bazuje na założeniu iż wirusy komputerowe mają postać zwykłych plików wykonywalnych, i działanie plików bezpiecznych będzie implikowało również działanie wirusów. Do tego celu zostało wybrane sześć popularnych programów   
z systemu Windows: calc.exe explorer.exe, notepad.exe, regedit.exe, twunk\_32.exe oraz write.exe.

Dodatkowym testem była statyczna analiza zmutowanych plików   
z wykorzystaniem narzędzi CFF Explorer [19] oraz portalu PEdump [20]. Biorąc to pod uwagę, autor dokonał pobieżnej analizy losowych plików wirusów, aby zapoznać się z ich wewnętrzną budową, tj. układem sekcji, zależnościami od innych bibliotek, obecnością kodu rozpakowującego itp. Po zebraniu listy cech charakterystycznych, spośród ogólnodostępnych programów systemowych oraz zewnętrznych aplikacji użytkowych, , wyselekcjonowana została grupa, której elementy zawierały możliwie najwięcej zidentyfikowanych cech. W dalszym kroku statycznie zostało sprawdzone czy po wykonaniu mutacji za pomocą zaimplementowanego systemu wszystkie wyselekcjonowane w poprzednim kroku cechy są w pełni zgodne ze specyfikacją formatu PE. Autor zdaje sobie sprawę, iż stan jego wiedzy na temat formatu PE czy sposobu działania wirusów komputerowych nie jest tak duży, aby można było przyjąć ze 100% pewnością, iż wykonane modyfikacje są poprawne w momencie gdy nie niszczą w żaden sposób wybranej grupy plików, jednak przeprowadzone eksperymenty potwierdzają poprawność z dużym prawdopodobieństwem. Należy jednak mieć na uwadze, iż projektowany system nie jest produktem komercyjnym, lecz narzędziem badawczym, które ma jedynie wskazać analitykom kierunek dalszych badań   
i eksperymentów. Dodatkowo, założeniem było udowodnienie, iż możliwe jest zautomatyzowanie wstrzykiwania nawet względnie zaawansowanych modyfikacji, co obrazuje skalę problemu generacji złośliwego oprogramowania.

# Wyniki eksperymentów

Aby zweryfikować wpływ modyfikacji plików wykonywalnych na wykrywalność malware-u, zostały przeprowadzone eksperymenty, polegające na ułożeniu kilkudziesięciu zestawów różnych metod mutacji. Na ich podstawie tworzone były potoki mutacyjne, służące do przekształcania wszystkich dostępnych próbek, których źródłem były systemy HoneyPot działające w Instytucie Informatyki na Wydziale Elektroniki i Technik Informacyjnych Politechniki Warszawskiej. Następnie zmodyfikowane pliki były wysyłane automatycznie z wykorzystaniem odpowiedniego skryptu do analizy w serwisie VirusTotal [21]. Usługa ta oferuje bezpłatną możliwość skanowania nadesłanych plików, aktualnie zestawem ponad 50 różnych silników antywirusowych. Wszystkie wykorzystane skanery są regularnie aktualizowane. Wyniki analiz również były pobierane automatycznie przez ten sam skrypt i na ich podstawie automatycznie generowane były statystyki wykrywalności. Schemat obrazujący przebieg eksperymentów ukazuje Rysunek 25.

Rysunek 25. Schemat przebiegu eksperymentów

Zbiór plików testowych zawierał 518 próbek wirusów w postaci plików   
o formacie PE. Podczas przeprowadzenia próby kontrolnej sprawdzono, jaki jest procent wykrywalności niebezpiecznych plików przed zastosowaniem opracowanych modyfikacji. Podczas projektowania zestawów mutacyjnych główny nacisk położony został na wpływ metody i krotności szyfrowania kodu   
w połączeniu z pozostałymi metodami. Z tego powodu zestawy podzielone są na dwie części. Pierwsza opiera się na deszyfratorze doklejanym, podczas gdy druga opiera się na deszyfratorze nadpisującym. W każdej części znajdują się zestawy   
z szyfrowaniem 1-, 2-, 4- i 8-krotnym połączonym ze wszystkimi możliwymi kombinacjami pozostałych silników mutacyjnych, za wyjątkiem szyfratora doklejanego. W jego przypadku architektura systemu nie pozwalała na połączenie z silnikami innymi niż GarbageCodeInjector. Nazwy zestawów odzwierciedlają kombinację podstawowych silników mutacyjnych wchodzących w ich skład wg. poniższej zasady:

* **EncryptionEngine –** szyfrowanie z dekoderem doklejającym (por. rozdział 4.4.2.1),
* **CopyEncryptionEngine** – szyfrowanie z dekoderem nadpisującym (por. rozdział 4.4.2.2,
* **Nx** – szyfrowanie N-krotne,
* **delay20 –** kod śmieciowy, którego wykonanie trwa ok. 20 sekund (por. rozdział 4.4.3),
* **imports –** usuwanie sekcji importów (por. rozdział 4.4.4).

W oparciu o podany klucz zaprojektowane zostały następujące zestawy mutacyjne:

1. CopyEncryptionEngine1x\_delay20,
2. CopyEncryptionEngine1x,
3. CopyEncryptionEngine2x\_delay20,
4. CopyEncryptionEngine2x,
5. CopyEncryptionEngine4x\_delay20,
6. CopyEncryptionEngine4x,
7. CopyEncryptionEngine8x\_delay20,
8. CopyEncryptionEngine8x,
9. CopyEncryptionEnginex1\_delay20\_imports,
10. CopyEncryptionEnginex1\_imports,
11. CopyEncryptionEnginex2\_delay20\_imports,
12. CopyEncryptionEnginex2\_imports,
13. CopyEncryptionEnginex4\_delay20\_imports,
14. CopyEncryptionEnginex4\_imports,
15. CopyEncryptionEnginex8\_delay20\_imports,
16. CopyEncryptionEnginex8\_imports,
17. EncryptionEngine1x\_delay20,
18. EncryptionEngine1x,
19. EncryptionEngine2x\_delay20,
20. EncryptionEngine2x,
21. EncryptionEngine4x\_delay20,
22. EncryptionEngine4x,
23. EncryptionEngine8x\_delay20,
24. EncryptionEngine8x.

Indeks każdego z zestawów na powyższej liście używany jest dalej jako identyfikator przypadku testowego. Tabele 1 i 2 (zob. rozdział 11) prezentują średni procent wykrywalności zmutowanych wirusów przez dostępne skanery antywirusowe. Rysunek 26 oraz Tabela 3 przedstawiają średni procent wykrywalności plików modyfikowanych przez zdefiniowane zestawy mutacyjne (liczony jako średnia arytmetyczna wyników cząstkowych dla poszczególnych programów antywirusowych). Z analizy uzyskanych danych wynika, iż średnio najmniej wykrywalne były wirusy generowane przez zestaw 10 (CopyEncryption Enginex1\_imports). Pierwsza połowa zestawienia to zestawy zawierające szyfratory nadpisujące. Oznacza to, iż fakt kopiowania kodu pomiędzy sekcjami nie powoduje wielu alarmów w dostępnych produktach antywirusowych. Dodatkowo można zauważyć, że wysoką skutecznością cechują się silniki mutacyjne usuwające sekcję importów. Można przypuszczać, iż sekcja ta jest analizowana przez algorytmy heurystyczne i w przypadku braku funkcji importowanych skanery w większości uznają plik za bezpieczny. Ciekawym spostrzeżeniem jest, iż wielokrotne szyfrowanie nie przyniosło oczekiwanych rezultatów w postaci niższej wykrywalności. Pierwsze miejsca zajmują silniki   
z jedno- i dwukrotnym szyfrowaniem. Możliwe więc, iż silniki antywirusowe wyszukują statycznie deszyfratorów i po kilku pierwszych znalezionych próbują odczytać kod. Jeżeli jest on wciąż niepoprawny (bo jest wciąż ukryty przez kolejną warstwą szyfrowania) to uznają plik za niebezpieczny. Analiza pozycji zestawów zawierających wstawianie kodu śmieciowego sugeruje, iż nie miał on większego wpływu na wynik skanowania. Można spróbować to wytłumaczyć faktem, iż analizy behawioralne (emulacja kodu) ma sens tylko dla plików uruchamianych manualnie przez użytkownika. W przeciwnym wypadku pełny skan systemu trwałby wiele godzin.

Rysunek 26. Wykrywalność wirusów wg. zestawów mutacyjnych

Warto zwrócić uwagę na fakt, iż liczba zmutowanych plików przez każdy   
z zestawów jest średnio o 60 plików mniejsza niż liczba wszystkich próbek testowych. Różnica ta wynika głównie z fizycznej niemożliwości zmutowania niektórych plików przez zaprojektowane silniki. Wszystkie ograniczenia, które uniemożliwiały generację nowych wirusów, opisane zostały w następnym podrozdziale. Podczas testów łącznie wygenerowanych zostało 10775 nowych wirusów komputerowych i ich łączny czas wysyłania i skanowania w serwisie VirusTotal wyniósł ok. 4 dni.

Drugim aspektem branym pod uwagę podczas analizy wyników były indywidualne wyniki poszczególnych skanerów antywirusowych. Tabela 4 przedstawia średnią wykrywalność zmodyfikowanych wirusów (liczoną jako średnia arytmetyczna cząstkowych wyników dla poszczególnych zestawów). Rysunek 27 prezentuje te same dane posortowane wg. procentu wykrywalności. Powstał w ten sposób ranking skanerów antywirusowych, który szereguje produkty antywirusowe pod kątem wykrywania zaprojektowanych mutacji od najlepszego do najgorszego. Jak widać, najlepszym produktem okazał się AntiVir. Jednak różnica w skuteczności kilku pierwszych skanerów jest niewielka i można uznać iż wszystkie wykonują analizy równie skutecznie. Zaskoczeniem jest, iż większość popularnych silników antywirusowych znalazło się w połowie listy. Już na pierwszy rzut oka można zauważyć, iż pomiędzy skrajnymi produktami jest ogromna różnica. Prawdopodobnie najsłabsze programy antywirusowe są efektem prac badawczych lub hobbystycznych prywatnych osób.

Rysunek 27. Ranking programów antywirusowych

Ostatnim aspektem branym pod uwagę podczas weryfikacji zaprojektowanych mutacji plików PE był wpływ na ilość zgłoszonych fałszywych alarmów przez programy antywirusowe. W tym celu wybrany został zestaw 6 bezpiecznych plików, występujących w większości wersji systemu Windows:

1. explorer.exe,
2. notepad.exe,
3. regedit.exe,
4. twunk\_32.exe,
5. write.exe,
6. calc.exe.

Wszystkie wymienione programy zostały poddane tym samym zestawom mutacyjnym oraz wysłane do analizy w serwisie VirusTotal. Na Rysunku 28 zaprezentowano uzyskane wyniki dotyczące zgłaszanych fałszywych alarmów przez poszczególne silniki antywirusowe, a na Rysunku 29 przedstawiony jest procent fałszywych alarmów generowanych przez poszczególne zestawy mutacyjne. Analiza Rysunku 27 pokazuje, iż 9 na 10 najskuteczniejszych programów antywirusowych, miało również najwyższy procent zgłaszanych fałszywych alarmów. Jednym z możliwych powodów takiego zachowania jest mechanizm działania oparty na wykrywaniu zdefiniowanych wzorców kodu, np. sekwencji szyfrowania metodą XOR. Dodatkowo, analiza nazw sklasyfikowanych zagrożeń wskazuje, iż w wielu przypadkach plik został uznany za generyczną postać malware-u o nazwie „Cryptor” lub „Patch”, co świadczy   
o niemożliwości jednoznacznego określenia typu wirusa, a jedynie podejrzanego fragmentu kodu. Z kolei analiza Rysunku 28 sugeruje, iż największe podejrzenia wzbudza kod, który wykonuje kopiowanie w obrębie sekcji kodu (obecność głównie deszyfratorów nadpisujących).

Rysunek 28. Procent zgłaszanych fałszywych alarmów przez antywirusy

Rysunek 29. Procent fałszywych alarmów dla zestawów mutacyjnych

# Podsumowanie

W ramach pracy magisterskiej zostało eksperymentalnie potwierdzone, iż możliwa jest automatyczna modyfikacja wirusów komputerowych i generowanie setek nowych wersji w czasie zaledwie kilkudziesięciu minut. Unikalne pliki wirusów powodują, iż spora część dostępnych obecnie programów antywirusowych ma problemy z ich poprawną identyfikacją. Istnieją jednak implementacje skanerów heurystycznych i statycznych, które są w stanie wyszukiwać w złośliwym oprogramowaniu pewnych ustalonych wzorców   
i oznaczać wg. nich pliki jako podejrzane. Potwierdzone również zostało, iż mimo iż z naukowego punktu są znane pewne algorytmiczne rozwiązania zapewniające skuteczność w wykrywaniu złośliwego oprogramowania, to często celowo nie są one implementowane w komercyjnych produktach ochronnych, w celu zminimalizowania obciążenia systemu i czasu analizy. Dla każdego systemu antywirusowego możliwe jest też dobranie metody, która w najlepszy możliwy sposób będzie dopasowana do jego charakterystyki jego analiz. W ten sposób, atakujący znając typ oprogramowania na maszynie ofiary jest w stanie odpowiednio zmodyfikować wirusy, aby uniknąć wykrycia. Udowodniono także, iż proste modyfikacje pozwalają znacznie ograniczyć możliwości wykrycia obecności złośliwego oprogramowania, co potwierdza wspomniane wcześniej obawy, iż skanery antywirusowe coraz częściej nie zapewniają odpowiedniego poziomu bezpieczeństwa. Wypracowania metodologia testowania oprogramowania antywirusowego może w przyszłości posłużyć do tworzenia rankingów skanerów antywirusowych.

### Ograniczenia i limitacje programu

Projekt malware-scrambler od początku projektowany był jako projekt eksperymentalny (tzw. „proof of concept”), którego celem jest udowodnienie lub zbadanie pewnej tezy. Na wielu etapach tworzenia architektury lub ostatecznej implementacji podejmowane były decyzje, które miały zrównoważyć wysiłek pomiędzy zapewnieniem kodu generyczności i skalowalności oraz obsługi jak największej ilości przypadków szczególnych. Z tego powodu końcowa aplikacja ma kilka ograniczeń, które podyktowane brakiem pełnej wiedzy nt. formatu plików Portable Executable.

Pierwsze ograniczenie polega na braku możliwości przesuwania sekcji bez informacji o relokacjach. Ze zbioru 518 plików testowych, aż 64 nie mogły być   
z tego powodu modyfikowane. Autor projektu stwierdził, iż implementacja kodu, który mimo tego umiałby samodzielnie wyszukać odpowiednie adresy zapiane   
w kodzie i w ogólnych strukturach PE jest zbyt skomplikowana i wymagałaby bardzo dokładnej i żmudnej analizy kodu binarnego. Wymagana byłaby bardzo dokładna wiedza na temat sposobu generowania kodu przez kompilatory, docelowej architektury kodu oraz wszystkich możliwych miejsc występowania jakichkolwiek adresów. Nie zmienia to jednak faktu, iż implementacja takiego modułu prawdopodobnie jest możliwa i mogłaby być w pewnych sytuacjach skuteczna.

Kolejne znane ograniczenie związane jest z innymi programami, które wykonują zestaw modyfikacji plików wykonywalnych. Najbardziej powszechna grupa takich aplikacji nazywana jest pakerami, ze względu na ich główną funkcjonalność dotyczącą kompresji. Plik wykonywalny jest przekształcany w taki sposób, aby zajmował jak najmniej miejsca. Dodatkowo wstrzykiwany jest kod, który ma zapewnić dekompresję w trakcie uruchamiania i inicjalizacji wszystkich struktur potrzebnych do poprawnego działania. Doświadczenie pokazało, iż zazwyczaj programy te ingerują mocno w układ samych sekcji. Jeden   
z popularniejszych pakerów, UPX, tworzy trzy sekcje, które otrzymują nazwy UPX*n*, gdzie *n* jest kolejnym numerem porządkowym. Sekcje te nie mają za wiele wspólnego z oryginalnym układem danych w pliku przed kompresją. Sekcja kodu zazwyczaj jest przechowywana w środkowej sekcji. Dodatkowo, zadeklarowany rozmiar tej sekcji jest dokładnie zaokrąglony do wyrównania naturalnego sekcji   
w pamięci. Powoduje to, iż teoretycznie nie ma wolnej przestrzeni pomiędzy tymi sekcjami, mimo iż ręczna analiza binarna pokazuje sporą przestrzeń wypełnioną samymi zerami. Podczas pracy program malware-scrambler nie jest w stanie samodzielnie zidentyfikować takiej sytuacji. Podczas próby implementacji modułu wykrywającego prawdziwy rozmiar sekcji kodu należałoby udzielić odpowiedzi na szereg pytań, które mogłyby być prawdziwe tylko dla ograniczonej liczby plików. Przykładem może być ilość bajtów o wartości zero, które miałyby oznaczać dodatkowe wypełnienie. Nie wiadomo jaka miałaby być granica pomiędzy możliwym wypełnieniem pustej przestrzeni, a celowym wprowadzeniem bloku danych zainicjalizowanych zerami. Brak takiego sprawdzania powoduje, iż takie pliki nie mogły być zmodyfikowane przy pomocy silników, które doklejają kod na koniec sekcji kodu.

Kolejnym ograniczeniem są wprowadzone zależności pomiędzy silnikami mutacyjnymi. Jest to bezpośrednią konsekwencją wydzielania wspólnych odpowiedzialności do osobnych silników mutacyjnych. Autor w celu umożliwienia współpracy pomiędzy poszczególnymi modułami wprowadził kontekst, jako medium komunikacyjne dla kolejnych silników. To oznacza jednak, iż są przypadki gdy jeden moduł oczekuje pewnych danych zapisanych w kontekście przez inny moduł. Przykładem może być silnik kopiujący kod, który przekazuje do innych silników adres względny skopiowanego kodu oraz jednocześnie sam oczekuje przekazania mu informacji o miejscu źródłowym i rozmiarze kopiowanego kodu. To z kolei narzuca wymaganie na odpowiednią konfigurację potoku mutacyjnego. Biorąc pod uwagę charakter programu („proof of concept”) oraz wiedzę osób potencjalnie korzystających z systemu wydaje się to mało istotnym ograniczeniem. Niemniej, autor wprowadził funkcje zapewniające minimalną walidację skonstruowanego potoku mutacyjnego. Nie zwalnia to jednak użytkownika systemu z dogłębnym zapoznaniem się z zależnościami pomiędzy poszczególnymi silnikami

### Perspektywy dalszego rozwoju

Projekt malware-scrambler jest aplikacją kompletną, która daje użytkownikowi pewny zbiór akcji do wykonania na plikach wykonywalnych. Zmodyfikowane programy działają i wykonują swoje pierwotne cele. Aplikacja realizowana była pod kątem zapewnienia rozszerzalności i skalowalności co umożliwia łatwe dodawanie kolejnych rozszerzeń mogących stać się bazą dla przyszłych projektów badawczych. Przeprowadzone przez autora eksperymenty oraz uzyskana podczas opracowywania systemu wiedza pozwalają wskazać obiecujące kierunki dalszych badań. Zostały one krótko opisane poniżej.

Warto rozważyć problem rozszerzania lub przesuwania sekcji w przestrzeni wirtualnej. Dodanie wsparcia dla takiej modyfikacji wiązałoby się z otworzeniem szerokiego spektrum nowych możliwych mutacji. Możliwe stałoby się wstrzykiwanie dowolnie dużego kodu w praktycznie dowolne miejsca każdej sekcji. Rozwiązałoby to też problem związany z programami pakującymi i ich modyfikacją układu sekcji.

Kolejnym kierunkiem wartym uwagi jest sposób szyfrowania danych. Operacja XOR, jak to zostało udowodnione w poprzednich rozdziałach, jest techniką znaną zarówno w literaturze jak i w implementacjach silników antywirusowych. Być może istnieje nowy sposób ukrywania prawdziwej natury kodu, którego nie da się wykryć znanymi sposobami analizy statycznej i algorytmami heurystycznymi.

Istotną perspektywą dalszego rozwoju jest również wprowadzanie do wirusów komputerowych metod obrony przed analizą behawioralną. Zaproponowane rozwiązanie polegające na wprowadzeniu sztucznego opóźnienia wykonania kodu jest dość naiwne i potencjalnie łatwo wykrywalne. Oczywistym jest, iż można po prostu zdefiniować bardziej skomplikowany szablon „śmieciowego” kodu, przykładowo znane są złośliwe programy które uruchamiają się dopiero po wykryciu faktu kilkukrotnego naciśnięcia lewego przycisku myszki [22], jednak metodą znacznie lepszą byłoby wstrzykiwanie pojedynczych instrukcji lub ich niewielkich grup pomiędzy istniejący kod programu. W ten sposób kod programu jest zaciemniany zarówno dla analityków jak i analizatorów wbudowanych   
w programy antywirusowe. Dodatkowo, wartym rozważenia jest pomysł wstrzykiwania instrukcji potencjalnie nieobsługiwanych przez emulatory lub piaskownice. Zaliczać się do nich mogą między innymi instrukcje zmienno-przecinkowe.

# Bibliografia

|  |  |  |
| --- | --- | --- |
| [1] | ISO (International Organization for Standarization), „ISO,” 21 Czerwiec 2000. [Online]. Available: http://www.iso.org/iso/iso\_catalogue/catalogue\_tc/catalogue\_detail.htm?csnumber=14256. [Data uzyskania dostępu: 1 Marzec 2014]. |  |
| [2] | K. Mitnick, Sztuka podstępu. Łamałem ludzi, nie hasła., Gliwice: Helion, 2003. |  |
| [3] | S. Venkatachalam, „Detecting undetectable computer viruses,” 2010. |  |
| [4] | E. Al Daoud, I. Jebril i B. Zaqaibeh, „Computer Virus Strategies and Detection Methods,” 2008. |  |
| [5] | F. Parvez, V. Laxmi i G. M, „Mining CFG as API Call-grams to Detect Portable Executable Malware,” Jaipur. |  |
| [6] | R. Perdisci, A. Lanzi i W. Lee, „Classification of packed executables for accurate computer virus detection,” Elsevier, 2008. |  |
| [7] | W. Arnold i G. Tesauro, „Automatically generated Win32 heuristic virus detection,” Virus Bulletin Conference, 2000. |  |
| [8] | D. Yadron, „The Wall Street Journal,” 4 Maj 2014. [Online]. Available: http://online.wsj.com/news/article\_email/SB10001424052702303417104579542140235850578-lMyQjAxMTA0MDAwNTEwNDUyWj. [Data uzyskania dostępu: 9 Wrzesień 2014]. |  |
| [9] | Microsoft Corporation, „Virtual Address Space (Windows),” 2014. [Online]. Available: http://msdn.microsoft.com/en-us/library/windows/desktop/aa366912(v=vs.85).aspx. [Data uzyskania dostępu: 3 Marzec 2014]. |  |
| [10] | Microsoft Corporation, „4-Gigabyte Tuning (Windows),” 2014. [Online]. Available: http://msdn.microsoft.com/en-us/library/windows/desktop/bb613473(v=vs.85).aspx. [Data uzyskania dostępu: 3 Marzec 2014]. |  |
| [11] | ruined-sec, „Memory Management - Windows part 2,” 1 Kwiecień 2013. [Online]. Available: http://ruinedsec.wordpress.com/2013/04/01/memory-management-windows-part2/. [Data uzyskania dostępu: 3 Marzec 2014]. |  |
| [12] | R. Kath, „The Portable Executable File Format from Top to Bottom,” 1997. [Online]. Available: http://www.csn.ul.ie/~caolan/publink/winresdump/winresdump/doc/pefile2.html. [Data uzyskania dostępu: 3 Marzec 2014]. |  |
| [13] | Microsoft, „Microsoft Portable Executable and Common Object File Format Specification,” Microsoft, Redmong, 2013. |  |
| [14] | ClamAV, „Creating signatures for ClamAV,” [Online]. Available: https://github.com/vrtadmin/clamav-devel/blob/master/docs/signatures.pdf. [Data uzyskania dostępu: 27 Sierpień 2014]. |  |
| [15] | P. Szor, Wirusy - Rozpoznawanie i obrona, Warszawa: Wydawnictwo Naukowe PWN, 2006. |  |
| [16] | Y. Liao, „PE-header-based Malware Study and Detection”. |  |
| [17] | Z. Shafiq, M. Tabish, F. Mirza i M. Farooq, „PE-Miner: Mining Structural Information to Detect Malicious Executables in Realtime,” Islamabad. |  |
| [18] | J. Sejdak, Analiza podobieństwa shellcode'u na podstawie grafu przebiegu jego wykonania, Warszawa, 2012. |  |
| [19] | NTCore, „CFF Explorer,” 2014. [Online]. Available: http://www.ntcore.com/exsuite.php. [Data uzyskania dostępu: 9 Wrzesień 2014]. |  |
| [20] | PEdump, „PEdump,” 2014. [Online]. Available: http://pedump.me/. [Data uzyskania dostępu: 9 Wrzesień 2014]. |  |
| [21] | Virus Total, „Virus Total,” 2014. [Online]. Available: https://www.virustotal.com/. [Data uzyskania dostępu: 9 Wrzesień 2014]. |  |
| [22] | C. Wueest, „www.symantec.org,” [Online]. Available: http://www.symantec.com/content/en/us/enterprise/media/security\_response/whitepapers/threats\_to\_virtual\_environments.pdf . [Data uzyskania dostępu: 24 Sierpień 2014]. |  |
| [23] | M. Pietrek, „An In-Depth Look into the Win32 Portable Executable File Format,” Luty 2002. [Online]. Available: http://msdn.microsoft.com/en-us/magazine/cc301805.aspx. [Data uzyskania dostępu: 3 Marzec 2014]. |  |
| [24] | M. Pietrek, „An In-Depth Look into the Win32 Portable Executable File Format, Part 2,” Marzec 2002. [Online]. Available: http://msdn.microsoft.com/en-us/magazine/cc301808.aspx. [Data uzyskania dostępu: 3 Marzec 2014]. |  |
| [25] | B. Rad, M. Maslin i S. Ibrahim, „Camouflage in Malware: from Encryption ot Metamorphism,” IJCSNS, 2012. |  |

# Spis obrazków

[Rysunek 1. Separacja przestrzeni adresowych poprzez pamięć wirtualną 5](#_Toc398070116)

[Rysunek 2. Mapa pamięci procesu w systemie Windows 8](file:///C:\Users\Kuba\Desktop\Praca%20magisterska.docx#_Toc398070117)

[Rysunek 3. Odwzorowanie bibliotek współdzielonych w przestrzeni procesów 9](file:///C:\Users\Kuba\Desktop\Praca%20magisterska.docx#_Toc398070118)

[Rysunek 4. Struktura pliku Portable Executable ( [23] [24]) 10](file:///C:\Users\Kuba\Desktop\Praca%20magisterska.docx#_Toc398070119)

[Rysunek 5. Definicja nagłówka DOS 11](file:///C:\Users\Kuba\Desktop\Praca%20magisterska.docx#_Toc398070120)

[Rysunek 6. Struktura nagłówka PE 12](file:///C:\Users\Kuba\Desktop\Praca%20magisterska.docx#_Toc398070121)

[Rysunek 7. Nagłówek pliku 12](file:///C:\Users\Kuba\Desktop\Praca%20magisterska.docx#_Toc398070122)

[Rysunek 8. IMAGE\_OPTIONAL\_HEADER 14](file:///C:\Users\Kuba\Desktop\Praca%20magisterska.docx#_Toc398070123)

[Rysunek 9. Porównanie wyrównania sekcji w pliku i w pamięci 16](file:///C:\Users\Kuba\Desktop\Praca%20magisterska.docx#_Toc398070124)

[Rysunek 10. Nagłówek sekcji 18](file:///C:\Users\Kuba\Desktop\Praca%20magisterska.docx#_Toc398070125)

[Rysunek 11. IMAGE\_EXPORT\_DIRECTORY 21](file:///C:\Users\Kuba\Desktop\Praca%20magisterska.docx#_Toc398070126)

[Rysunek 12. Eksportowanie symboli przez moduł PE 22](#_Toc398070127)

[Rysunek 13. Struktura IMAGE\_IMPORT\_DESCRIPTOR 23](file:///C:\Users\Kuba\Desktop\Praca%20magisterska.docx#_Toc398070128)

[Rysunek 14. IMAGE\_THUNK\_DATA 24](file:///C:\Users\Kuba\Desktop\Praca%20magisterska.docx#_Toc398070129)

[Rysunek 15. Schemat importowania symboli 25](file:///C:\Users\Kuba\Desktop\Praca%20magisterska.docx#_Toc398070130)

[Rysunek 16. Struktura używana w procesie weryfikacji importowanych symboli 25](file:///C:\Users\Kuba\Desktop\Praca%20magisterska.docx#_Toc398070131)

[Rysunek 17. Przykłady instrukcji martwego kodu [25] 31](file:///C:\Users\Kuba\Desktop\Praca%20magisterska.docx#_Toc398070132)

[Rysunek 18. Przykładowy plik konfigurujący potok mutacyjny 40](#_Toc398070133)

[Rysunek 19. Fragment szablonu kodu modułu CodeProvider 45](file:///C:\Users\Kuba\Desktop\Praca%20magisterska.docx#_Toc398070134)

[Rysunek 20. Efekt szyfrowania jednokrotnego z doklejaniem deszyfratora 48](file:///C:\Users\Kuba\Desktop\Praca%20magisterska.docx#_Toc398070135)

[Rysunek 21. Deszyfrowanie wielokrotne 49](file:///C:\Users\Kuba\Desktop\Praca%20magisterska.docx#_Toc398070136)

[Rysunek 22. Efekt szyfrowania sekcji kodu metodą nadpisywania 51](file:///C:\Users\Kuba\Desktop\Praca%20magisterska.docx#_Toc398070137)

[Rysunek 23. Proces deszyfrowania wielokrotnego deszyfratorem nadpisującym 52](file:///C:\Users\Kuba\Desktop\Praca%20magisterska.docx#_Toc398070138)

[Rysunek 24. Szablon kodu opóźniającego 54](file:///C:\Users\Kuba\Desktop\Praca%20magisterska.docx#_Toc398070139)

[Rysunek 25. Schemat przebiegu eksperymentów 61](file:///C:\Users\Kuba\Desktop\Praca%20magisterska.docx#_Toc398070140)

[Rysunek 26. Wykrywalność wirusów wg. zestawów mutacyjnych 64](file:///C:\Users\Kuba\Desktop\Praca%20magisterska.docx#_Toc398070141)

[Rysunek 27. Ranking programów antywirusowych 66](file:///C:\Users\Kuba\Desktop\Praca%20magisterska.docx#_Toc398070142)

[Rysunek 28. Procent zgłaszanych fałszywych alarmów przez antywirusy 68](file:///C:\Users\Kuba\Desktop\Praca%20magisterska.docx#_Toc398070143)

[Rysunek 29. Procent fałszywych alarmów dla zestawów mutacyjnych 69](#_Toc398070144)

# Spis tabel

[Tabela 1. Procent detekcji zmutowanych wirusów dla zestawów 1 - 12 79](#_Toc398070145)

[Tabela 2. Procent detekcji zmutowanych wirusów dla zestawów 13 - 24 82](#_Toc398070146)

[Tabela 3. Średnia wykrywalność wirusów mutowanych przez dany zestaw 86](#_Toc398070147)

[Tabela 4. Skuteczność wykrywania poszczególnych skanerów AV 86](#_Toc398070148)

# Załączniki

Tabela 1. Procent detekcji zmutowanych wirusów dla zestawów 1 - 12

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| AVG | 7,71 | 61,67 | 7,71 | 62,78 | 7,71 | 62,11 | 7,71 | 68,87 | 4,28 | 15,09 | 6,76 | 14,64 |
| Ad-Aware | 96,04 | 97,36 | 99,56 | 97,80 | 99,34 | 97,8 | 99,34 | 95,14 | 94,82 | 70,50 | 96,17 | 79,95 |
| AegisLab | 0,00 | 0,22 | 0,00 | 0,00 | 0,00 | 0,00 | 0,00 | 0,00 | 0,00 | 0,00 | 0,00 | 0,00 |
| Agnitum | 3,08 | 4,40 | 3,52 | 4,40 | 3,52 | 4,85 | 3,52 | 8,17 | 3,60 | 3,83 | 3,60 | 3,83 |
| AhnLab-V3 | 20,70 | 16,96 | 21,15 | 19,16 | 18,50 | 20,04 | 21,15 | 21,19 | 2,93 | 0,45 | 2,93 | 2,93 |
| AntiVir | 98,46 | 98,90 | 98,90 | 98,24 | 98,90 | 98,90 | 98,68 | 98,45 | 78,15 | 78,15 | 77,70 | 77,70 |
| Antiy-AVL | 1,10 | 1,10 | 0,88 | 1,32 | 0,66 | 1,32 | 1,10 | 5,30 | 0,90 | 0,90 | 0,68 | 0,90 |
| Avast | 31,06 | 32,16 | 31,50 | 31,72 | 32,16 | 32,60 | 32,16 | 30,91 | 30,41 | 30,86 | 31,08 | 24,55 |
| Baidu-International | 0,00 | 0,00 | 0,00 | 0,00 | 0,00 | 0,00 | 0,00 | 6,62 | 0,00 | 0,00 | 0,00 | 0,00 |
| BitDefender | 98,90 | 98,02 | 99,56 | 97,80 | 99,34 | 97,80 | 99,12 | 99,12 | 96,62 | 70,50 | 96,17 | 84,23 |
| Bkav | 98,02 | 84,58 | 94,49 | 90,97 | 95,59 | 88,77 | 97,14 | 86,09 | 90,32 | 6,53 | 94,14 | 6,98 |
| ByteHero | 0,22 | 4,85 | 4,40 | 5,07 | 4,63 | 4,63 | 4,63 | 5,96 | 2,03 | 4,50 | 3,83 | 1,13 |
| CAT-QuickHeal | 6,61 | 6,61 | 6,61 | 6,39 | 6,61 | 6,61 | 6,61 | 9,71 | 6,98 | 6,98 | 6,98 | 6,76 |
| CMC | 3,74 | 4,63 | 4,63 | 4,40 | 4,63 | 4,40 | 4,18 | 4,41 | 2,48 | 2,48 | 2,48 | 2,25 |
| ClamAV | 24,45 | 24,45 | 24,45 | 24,01 | 24,23 | 24,45 | 24,45 | 24,28 | 24,10 | 24,10 | 24,10 | 23,87 |
| Commtouch | 12,11 | 55,51 | 12,11 | 52,64 | 12,11 | 50,22 | 12,11 | 50,11 | 52,03 | 51,35 | 51,80 | 49,32 |
| Comodo | 57,71 | 83,26 | 57,71 | 83,70 | 57,93 | 83,92 | 56,83 | 82,78 | 43,02 | 60,36 | 51,35 | 79,05 |
| DrWeb | 3,96 | 37,22 | 4,63 | 31,94 | 4,63 | 77,97 | 4,40 | 34,22 | 4,95 | 32,21 | 4,28 | 31,08 |
| ESET-NOD32 | 39,21 | 63,00 | 41,63 | 58,81 | 41,63 | 93,39 | 41,41 | 92,27 | 9,00 | 4,50 | 9,00 | 4,28 |
| Emsisoft | 98,90 | 94,93 | 99,56 | 97,80 | 98,68 | 97,80 | 98,90 | 98,68 | 96,17 | 70,50 | 96,17 | 82,66 |
| F-Prot | 15,86 | 68,72 | 15,64 | 59,03 | 15,86 | 53,96 | 15,86 | 53,64 | 52,03 | 13,51 | 51,8 | 51,35 |
| F-Secure | 95,15 | 93,17 | 93,83 | 96,70 | 96,7 | 97,14 | 93,39 | 90,95 | 89,64 | 2,93 | 92,34 | 64,19 |
| Fortinet | 1,10 | 1,10 | 1,10 | 1,10 | 1,10 | 1,10 | 1,10 | 1,32 | 0,90 | 24,55 | 0,90 | 0,90 |
| GData | 98,02 | 96,70 | 99,56 | 97,80 | 99,12 | 97,80 | 99,12 | 99,12 | 96,17 | 24,55 | 96,40 | 82,21 |
| Ikarus | 53,96 | 58,71 | 55,51 | 58,59 | 55,51 | 58,37 | 55,29 | 64,02 | 13,29 | 43,24 | 13,06 | 13,29 |
| Jiangmin | 3,74 | 3,96 | 3,74 | 3,52 | 3,74 | 3,74 | 3,74 | 3,31 | 2,93 | 0,00 | 2,93 | 2,93 |
| K7AntiVirus | 24,01 | 24,01 | 24,01 | 23,79 | 24,01 | 24,01 | 24,01 | 30,68 | 19,59 | 4,50 | 24,55 | 24,55 |
| K7GW | 24,01 | 24,01 | 24,01 | 23,79 | 24,01 | 24,01 | 24,01 | 30,68 | 24,55 | 2,48 | 24,55 | 24,55 |
| Kaspersky | 3,08 | 90,75 | 3,30 | 91,85 | 2,86 | 92,51 | 7,49 | 96,26 | 0,90 | 8,33 | 0,90 | 40,77 |
| Kingsoft | 0,88 | 0,88 | 0,88 | 0,88 | 0,88 | 0,88 | 0,88 | 0,44 | 0,00 | 70,50 | 0,00 | 0,00 |
| Malwarebytes | 4,40 | 4,40 | 4,40 | 4,40 | 4,40 | 4,40 | 4,64 | 4,64 | 4,50 | 4,50 | 4,28 | 4,28 |
| McAfee | 7,05 | 53,74 | 2,20 | 53,52 | 48,46 | 48,68 | 48,24 | 54,75 | 1,35 | 2,48 | 1,13 | 2,48 |
| McAfee-GW-Edition | 14,54 | 61,01 | 9,91 | 60,57 | 10,13 | 56,17 | 9,91 | 61,59 | 8,11 | 8,33 | 7,88 | 8,56 |
| MicroWorld-eScan | 98,24 | 96,70 | 98,90 | 97,80 | 99,34 | 97,80 | 98,69 | 99,12 | 94,37 | 70,50 | 96,17 | 83,33 |
| Microsoft | 2,20 | 93,17 | 2,20 | 94,05 | 2,20 | 94,49 | 2,20 | 94,26 | 2,52 | 45,27 | 2,25 | 44,82 |
| NANO-Antivirus | 2,64 | 3,52 | 5,51 | 3,96 | 5,51 | 4,85 | 5,95 | 5,74 | 7,43 | 8,56 | 7,21 | 8,33 |
| Norman | 42,73 | 44,71 | 42,73 | 45,15 | 42,73 | 45,15 | 42,73 | 51,88 | 2,52 | 3,15 | 2,03 | 2,93 |
| Panda | 37,22 | 38,11 | 37,44 | 37,44 | 37,89 | 38,11 | 37,67 | 37,75 | 0,00 | 0,23 | 0,00 | 0,23 |
| Qihoo-360 | 74,01 | 76,87 | 68,50 | 80,62 | 77,53 | 68,94 | 66,96 | 67,77 | 66,89 | 63,51 | 73,87 | 62,61 |
| Rising | 9,25 | 54,19 | 9,03 | 55,29 | 9,25 | 8,81 | 9,25 | 8,83 | 7,66 | 7,66 | 7,66 | 7,66 |
| SUPERAntiSpyware | 8,59 | 8,59 | 8,59 | 8,37 | 8,59 | 8,59 | 8,59 | 8,17 | 8,56 | 12,16 | 11,71 | 12,16 |
| Sophos | 55,07 | 44,27 | 54,85 | 49,56 | 55,07 | 48,46 | 51,98 | 42,16 | 93,47 | 47,75 | 92,79 | 40,77 |
| Symantec | 41,19 | 45,59 | 46,26 | 44,71 | 46,48 | 46,48 | 46,04 | 46,80 | 21,62 | 21,17 | 21,17 | 18,24 |
| Tencent | 0,88 | 7,71 | 0,88 | 5,95 | 2,64 | 2,64 | 1,10 | 2,43 | 0,68 | 1,13 | 0,90 | 0,90 |
| TheHacker | 0,66 | 0,66 | 0,66 | 0,66 | 0,66 | 0,66 | 0,44 | 0,44 | 0,45 | 0,00 | 0,00 | 0,23 |
| TotalDefense | 3,08 | 3,08 | 3,08 | 3,08 | 3,08 | 3,08 | 3,08 | 6,84 | 2,03 | 2,03 | 2,03 | 2,03 |
| TrendMicro | 51,98 | 56,17 | 56,39 | 55,51 | 56,61 | 56,83 | 56,71 | 61,81 | 56,98 | 56,31 | 54,28 | 45,72 |
| TrendMicro-HouseCall | 53,30 | 56,83 | 53,30 | 56,39 | 55,73 | 55,95 | 56,17 | 60,49 | 56,08 | 56,08 | 52,48 | 47,52 |
| VBA32 | 98,90 | 81,72 | 99,56 | 83,26 | 98,90 | 83,26 | 98,02 | 83,22 | 98,42 | 34,68 | 99,32 | 33,78 |
| VIPRE | 6,17 | 93,39 | 6,61 | 92,73 | 6,83 | 92,78 | 6,83 | 88,08 | 6,53 | 6,76 | 6,53 | 6,76 |
| ViRobot | 1,10 | 1,10 | 1,10 | 1,10 | 1,10 | 1,10 | 1,10 | 1,10 | 1,13 | 1,13 | 1,13 | 1,13 |
| Zillya | 0,88 | 0,88 | 0,88 | 0,88 | 0,88 | 0,88 | 0,88 | 0,44 | 1,13 | 1,13 | 1,13 | 0,90 |
| nProtect | 39,87 | 42,51 | 41,19 | 42,73 | 42,07 | 43,39 | 39,87 | 39,29 | 2,93 | 3,60 | 2,93 | 3,38 |

Tabela 2. Procent detekcji zmutowanych wirusów dla zestawów 13 - 24

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | |
| AVG | 6,76 | 14,86 | 6,76 | 45,72 | 96,69 | 78,19 | 98,00 | 78,15 | 99,11 | 77,06 | 98,65 | 75,45 | |
| Ad-Aware | 96,62 | 99,10 | 99,10 | 98,65 | 98,23 | 59,69 | 95,99 | 59,38 | 95,53 | 58,57 | 98,19 | 89,64 | |
| AegisLab | 0,00 | 0,00 | 0,00 | 0,00 | 0,00 | 0,22 | 0,00 | 0,00 | 0,00 | 0,00 | 0,00 | 0,00 | |
| Agnitum | 3,60 | 3,60 | 3,60 | 3,83 | 3,09 | 4,63 | 3,12 | 4,64 | 3,13 | 4,68 | 3,16 | 4,28 | |
| AhnLab-V3 | 2,93 | 2,93 | 2,93 | 2,93 | 23,84 | 24,67 | 23,61 | 25,17 | 25,28 | 24,72 | 23,93 | 22,08 | |
| AntiVir | 78,15 | 77,93 | 78,15 | 77,93 | 98,90 | 99,34 | 98,44 | 99,12 | 98,88 | 98,89 | 98,87 | 99,10 | |
| Antiy-AVL | 0,90 | 1,13 | 0,90 | 0,68 | 1,10 | 1,32 | 1,11 | 1,32 | 1,12 | 1,34 | 1,13 | 1,13 | |
| Avast | 30,63 | 31,76 | 31,08 | 31,76 | 32,23 | 32,60 | 32,96 | 32,67 | 33,11 | 32,74 | 30,70 | | 33,11 |
| Baidu-International | 0,00 | 0,00 | 0,00 | 0,00 | 0,00 | 0,22 | 0,00 | 0,22 | 0,00 | 0,22 | 0,00 | | 0,23 |
| BitDefender | 96,62 | 99,10 | 99,10 | 98,65 | 98,23 | 60,35 | 98,44 | 59,60 | 98,43 | 58,57 | 98,19 | | 89,64 |
| Bkav | 92,34 | 7,43 | 89,41 | 7,21 | 94,70 | 88,77 | 97,77 | 84,77 | 93,74 | 81,74 | 90,97 | | 86,49 |
| ByteHero | 3,83 | 4,50 | 3,15 | 4,73 | 2,43 | 4,85 | 4,01 | 4,64 | 4,03 | 2,89 | 4,74 | | 3,15 |
| CAT-QuickHeal | 6,98 | 6,76 | 6,98 | 6,98 | 5,30 | 5,23 | 5,34 | 5,30 | 5,37 | 5,34 | 5,87 | | 5,40 |
| CMC | 2,25 | 2,25 | 0,90 | 0,90 | 3,53 | 3,52 | 3,79 | 3,97 | 3,58 | 3,79 | 3,17 | | 3,83 |
| ClamAV | 24,1 | 23,87 | 24,10 | 24,1 | 24,72 | 24,67 | 24,94 | 24,50 | 25,06 | 24,92 | 25,06 | | 25,23 |
| Commtouch | 52,03 | 52,25 | 52,03 | 52,48 | 12,14 | 58,15 | 12,25 | 56,73 | 12,30 | 52,56 | 12,42 | | 52,25 |
| Comodo | 56,76 | 82,66 | 52,48 | 68,69 | 56,95 | 69,38 | 56,79 | 73,73 | 59,51 | 79,29 | 59,37 | | 48,65 |
| DrWeb | 9,23 | 33,33 | 9,01 | 32,66 | 2,87 | 79,07 | 3,34 | 80,79 | 6,49 | 81,51 | 4,74 | | 81,31 |
| ESET-NOD32 | 10,14 | 44,37 | 10,59 | 90,32 | 37,75 | 97,14 | 39,87 | 92,27 | 40,27 | 96,66 | 39,28 | | 97,07 |
| Emsisoft | 96,62 | 99,10 | 99,10 | 98,42 | 97,79 | 60,35 | 98,00 | 59,60 | 98,43 | 57,91 | 97,97 | | 89,64 |
| F-Prot | 52,03 | 52,25 | 52,03 | 52,48 | 15,89 | 68,72 | 16,04 | 59,60 | 16,11 | 53,90 | 16,25 | | 53,60 |
| F-Secure | 95,27 | 98,42 | 96,62 | 98,42 | 74,92 | 60,57 | 98,00 | 56,73 | 97,99 | 57,46 | 96,84 | | 88,74 |
| Fortinet | 0,90 | 0,68 | 0,90 | 0,90 | 1,10 | 1,10 | 1,11 | 1,10 | 1,12 | 1,11 | 1,13 | | 1,13 |
| GData | 96,85 | 99,10 | 99,10 | 98,65 | 98,23 | 60,35 | 98,00 | 59,60 | 98,43 | 58,57 | 97,97 | | 89,64 |
| Ikarus | 13,29 | 13,29 | 13,29 | 13,51 | 55,63 | 58,81 | 55,68 | 58,94 | 55,93 | 59,24 | 55,98 | | 59,01 |
| Jiangmin | 2,93 | 2,93 | 2,93 | 2,93 | 2,87 | 2,86 | 2,89 | 2,87 | 291 | 2,89 | 2,93 | | 2,93 |
| K7AntiVirus | 24,55 | 24,32 | 24,55 | 24,55 | 24,28 | 24,23 | 24,50 | 24,28 | 24,61 | 24,50 | 24,83 | | 24,77 |
| K7GW | 24,55 | 24,32 | 24,55 | 24,55 | 24,28 | 24,23 | 24,50 | 24,28 | 24,61 | 24,50 | 24,83 | | 24,77 |
| Kaspersky | 0,90 | 41,89 | 0,90 | 43,47 | 3,31 | 90,53 | 3,12 | 85,43 | 3,36 | 89,98 | 3,39 | | 91,22 |
| Kingsoft | 0,00 | 0,00 | 0,00 | 0,00 | 0,00 | 0,00 | 0,00 | 0,00 | 0,00 | 0,00 | 0,00 | | 0,00 |
| Malwarebytes | 4,50 | 4,28 | 4,50 | 4,28 | 4,64 | 4,63 | 4,68 | 4,64 | 4,25 | 4,45 | 4,03 | | 4,05 |
| McAfee | 1,35 | 2,25 | 1,13 | 97,75 | 2,65 | 59,91 | 2,45 | 60,71 | 2,46 | 59,47 | 2,48 | | 59,46 |
| McAfee-GW-Edition | 8,11 | 8,11 | 8,11 | 8,56 | 11,92 | 61,01 | 11,80 | 61,81 | 12,08 | 61,47 | 12,19 | | 61,71 |
| MicroWorld-eScan | 96,62 | 99,10 | 99,10 | 98,65 | 98,23 | 60,35 | 98,00 | 59,60 | 98,21 | 58,57 | 97,97 | | 89,64 |
| Microsoft | 2,25 | 45,27 | 2,25 | 45,05 | 2,43 | 95,81 | 2,23 | 95,36 | 2,37 | 96,21 | 2,26 | | 95,95 |
| NANO-Antivirus | 7,43 | 8,33 | 7,43 | 8,33 | 2,43 | 2,42 | 0,22 | 2,43 | 0,22 | 2,45 | 0,23 | | 2,48 |
| Norman | 2,25 | 2,93 | 2,25 | 3,15 | 50,77 | 57,71 | 51,22 | 57,40 | 51,45 | 58,13 | 51,69 | | 58,78 |
| Panda | 0,00 | 0,23 | 0,00 | 0,22 | 22,30 | 24,45 | 29,18 | 24,72 | 30,43 | 27,84 | 25,51 | | 24,10 |
| Qihoo-360 | 77,70 | 69,37 | 63,74 | 55,86 | 82,34 | 78,19 | 85,52 | 81,02 | 75,62 | 74,61 | 70,43 | | 75,23 |
| Rising | 7,66 | 7,43 | 7,66 | 7,21 | 9,27 | 54,85 | 9,80 | 54,75 | 11,19 | 55,90 | 11,29 | | 11,04 |
| SUPERAntiSpyware | 9,23 | 11,71 | 20,05 | 36,94 | 13,91 | 16,08 | 19,15 | 13,91 | 17,23 | 19,15 | 8,80 | | 12,16 |
| Sophos | 89,19 | 48,65 | 94,59 | 44,14 | 52,54 | 48,02 | 55,01 | 46,80 | 54,36 | 48,78 | 53,50 | | 49,10 |
| Symantec | 21,62 | 21,4 | 20,05 | 21,62 | 46,80 | 46,92 | 44,99 | 47,02 | 46,53 | 46,55 | 46,50 | | 46,62 |
| Tencent | 0,90 | 1,13 | 0,90 | 0,90 | 1,10 | 4,63 | 1,11 | 6,18 | 1,12 | 1,34 | 1,13 | | 1,13 |
| TheHacker | 0,00 | 0,23 | 0,68 | 0,23 | 0,44 | 0,44 | 0,67 | 0,66 | 0,45 | 0,45 | 0,45 | | 0,45 |
| TotalDefense | 2,03 | 2,03 | 2,03 | 2,03 | 2,21 | 2,20 | 2,23 | 2,21 | 2,24 | 2,23 | 2,26 | | 2,25 |
| TrendMicro | 56,98 | 56,76 | 52,03 | 56,76 | 56,73 | 57,05 | 56,35 | 57,17 | 57,49 | 57,68 | 58,01 | | 58,11 |
| TrendMicro-HouseCall | 56,98 | 52,48 | 56,08 | 55,63 | 56,73 | 56,39 | 54,79 | 56,95 | 56,82 | 55,68 | 57,56 | | 57,43 |
| VBA32 | 99,77 | 34,68 | 99,77 | 34,46 | 96,47 | 86,34 | 97,55 | 85,87 | 97,54 | 85,97 | 96,61 | | 86,49 |
| VIPRE | 6,53 | 6,53 | 5,18 | 6,76 | 6,84 | 92,73 | 5,57 | 92,72 | 6,49 | 92,65 | 6,32 | | 92,79 |
| ViRobot | 1,13 | 0,90 | 1,13 | 1,13 | 1,10 | 1,10 | 1,11 | 1,10 | 1,12 | 1,11 | 1,13 | | 1,13 |
| Zillya | 1,13 | 0,90 | 1,13 | 0,23 | 2,43 | 2,20 | 1,78 | 2,21 | 2,46 | 2,00 | 1,58 | | 1,58 |
| nProtect | 2,93 | 3,38 | 2,93 | 3,60 | 41,94 | 43,17 | 42,32 | 43,93 | 42,28 | 43,21 | 41,76 | | 43,02 |

Tabela 3. Średnia wykrywalność wirusów mutowanych przez dany zestaw

|  |  |  |
| --- | --- | --- |
| Zestaw mutacyjny | Średnia wykrywalność | Liczba zmutowanych wirusów |
| 1 | 31,62 % | 454 |
| 2 | 43,41 % | 454 |
| 3 | 31,86 % | 454 |
| 4 | 43,47 % | 454 |
| 5 | 33,03 % | 454 |
| 6 | 43,63 % | 454 |
| 7 | 32,73 % | 454 |
| 8 | 43,78 % | 453 |
| 9 | 28,07 % | 444 |
| 10 | 24,41 % | 444 |
| 11 | 28,60 % | 444 |
| 12 | 24,90 % | 444 |
| 13 | 29,02 % | 444 |
| 14 | 28,91 % | 444 |
| 15 | 29,00 % | 444 |
| 16 | 32,07 % | 444 |
| 17 | 33,53 % | 453 |
| 18 | 41,52 % | 454 |
| 19 | 33,95 % | 449 |
| 20 | 41,11 % | 453 |
| 21 | 34,01 % | 447 |
| 22 | 40,97 % | 449 |
| 23 | 33,48 % | 443 |
| 24 | 42,98 % | 444 |

Tabela 4. Skuteczność wykrywania poszczególnych skanerów AV

|  |  |
| --- | --- |
| Silnik antywirusowy | Procent wykrywalności wirusów |
| Ad-Aware | 91.43 % |
| AegisLab | 0.01844 % |
| Agnitum | 3.998 % |
| AhnLab-V3 | 15.57 % |
| AntiVir | 92.02 % |
| Antiy-AVL | 1.224 % |
| Avast | 32.36 % |
| AVG | 46.4 % |
| Baidu-International | 0.3173 % |
| BitDefender | 91.45 % |
| Bkav | 81.09 % |
| ByteHero | 4.295 % |
| CAT-QuickHeal | 6.442 % |
| ClamAV | 24.45 % |
| CMC | 3.345 % |
| Commtouch | 39.3 % |
| Comodo | 70.91 % |
| DrWeb | 29.96 % |
| Emsisoft | 91.4 % |
| ESET-NOD32 | 50.19 % |
| Fortinet | 1.04 % |
| F-Prot | 42.41 % |
| F-Secure | 91.4 % |
| GData | 91.46 % |
| Ikarus | 42.89 % |
| Jiangmin | 3.178 % |
| K7AntiVirus | 24.68 % |
| K7GW | 24.64 % |
| Kaspersky | 39.56 % |
| Kingsoft | 0.2877 % |
| Malwarebytes | 4.439 % |
| McAfee | 28.27 % |
| McAfee-GW-Edition | 26.91 % |
| Microsoft | 40.47 % |
| MicroWorld-eScan | 91.44 % |
| NANO-Antivirus | 4.752 % |
| Norman | 34.1 % |
| nProtect | 29.69 % |
| Panda | 21.38 % |
| Qihoo-360 | 73.73 % |
| Rising | 18.53 % |
| Sophos | 58.69 % |
| SUPERAntiSpyware | 12.99 % |
| Symantec | 38.3 % |
| Tencent | 2.199 % |
| TheHacker | 0.4445 % |
| TotalDefense | 2.608 % |
| TrendMicro | 57.32 % |
| TrendMicro-HouseCall | 56.04 % |
| VBA32 | 83.63 % |
| VIPRE | 35.44 % |
| ViRobot | 1.107 % |
| Zillya | 1.487 % |