

1. Wstęp

Niniejszy dokument opisuje przebieg realizacji projektu z przedmiotu BEST. Założeniem projektu było istnienie grup projektujących, które miały za zadanie ukrycie komunikacji steganograficznej w normalnym ruchu sieciowym oraz grup detekujących, które miały zlokalizować fakt tajnej komunikacji. Ze względu na to, że grupy projektujące mogły szyfrować ukrytą transmisję, to nasz zespół jako grupa detekująca starał się raczej zlokalizować fakt zaistnienia ukrytej komunikacji, aniżeli odzyskać ukryte dane.

Dalsza część raportu jest podzielona na poszczególne grupy. Dla każdej grupy opisaliśmy nasze osiągnięcia dotyczące analizy zastosowanych metod steganografii. Przy czym staraliśmy się zaprezentować podejście wszechstronne, tj. dla każdej grupy przeprowadziliśmy inżynierię wsteczną aplikacji detektora, analizowaliśmy pliki pcap metodą manualną (statycznie i porównawczo), a także w części zautomatyzowaną.

2. Majewski i Natur

Inżynieria wsteczna

Zespół nr 7 jako jedyny zrealizował formalne wymogi projektu, ponieważ zastosował się do scenariusza, który wskazuje, iż aplikacja detektora ma być stanie odczytać tajne dane: "Rezultatem końcowym realizacji projektu przez grupy projektujące (...) jest:"

Aplikacja (.exe) będąca w stanie odczytać zapisane w ruchu sieciowym (plikach .pcap) tajne dane. Aplikacja ma działać w następujący sposób: można wywołać aplikację podając jako argument plik .pcap np. *aplikacja.exe 123.pcap* i w rezultacie działania aplikacja poinformuje użytkownika czy plik zawiera ukryte dane czy nie.

Rys. 1, wycinek oficjalnego scenariusza projektu

A więc zgodnie z rys. 1 aplikacja powinna odczytywać tajne dane. Wszystkie pozostałe 6 zespołów nie spełniło tego wymogu - tak wynika z przeprowadzonej przez nasz zespół inżynierii wstecznej. Jednak prawdopodobnie jest to nieścisłość scenariusza projektowego, a oficjalne założenia są inne.

Przechodząc do meritum, zespół nr 7 zrealizował aplikację detektora jako program C++, tak więc przeanalizowanie działania tego programu nie było proste, gdyż konieczna była analiza kodu asemblera oraz znajomość narzędzi służących do debugowania kodu. (początkowo próbowano dekompilacji z różnymi narzędziami typu Hopper, Rec Studio, snowman i inne, jednak wygenerowany kod w C był nawet trudniejszy do analizy niż asembler)

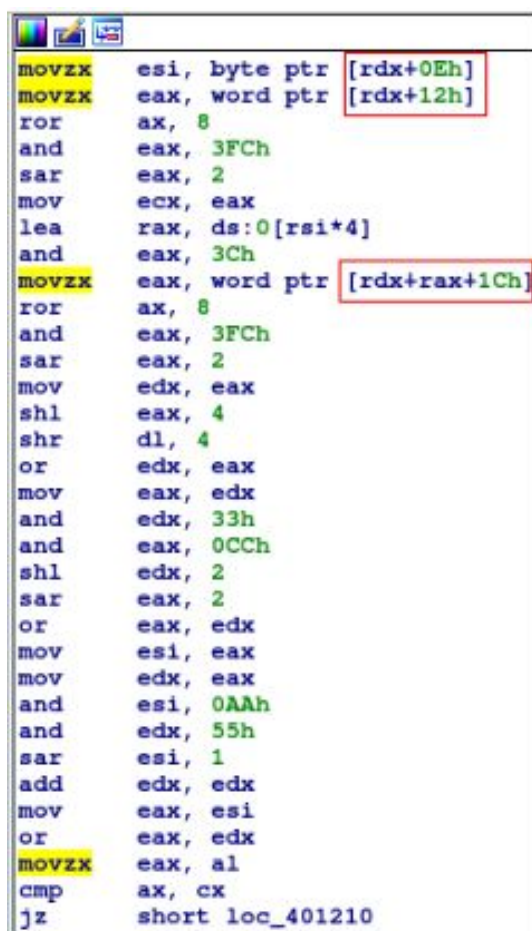
Pierwszym podjętym krokiem była analiza statyczna kodu za pomocą popularnego narzędzia IDA 7.0 freeware. Dzięki temu wykryto, że detektor korzysta z funkcji domyślnie dostarczanych dla systemów Linux, czyli `pcap_loop`, która jako jeden z argumentów przyjmuje callback, który zostanie wywołany celem przetwarzania pakietów z pliku pcap.



Rys. 2, wycinek grafu przeływów kodu asemblera (IDA 7.0)

Jak widać na rys. 2 przed wywołaniem funkcji pcap_loop komputer odpowiednio ustawia rejestry, które de facto przekażą argumenty do funkcji pcap_loop. Widać, że do funkcji przekazane zostaną dwa wartościowe (dla nas) argumenty, tj. callback o nazwie my_packet_handler (w rejestrze edx) oraz liczba pakietów do przetworzenia (0x4e20=20000) (w rejestrze esi).

Kolejnym naturalnym krokiem była analiza działania funkcji my_packet_handler. W celu uproszczenia opisu należy zauważyć, że wspomniana funkcja na początku działania sprawdza, czy dany pakiet korzysta z ipv4 oraz tcp. Natomiast, główna funkcjonalność jest widoczna na rys. 3.



```
movzx esi, byte ptr [rdx+0Eh]
movzx eax, word ptr [rdx+12h]
ror ax, 8
and eax, 3FCh
sar eax, 2
mov ecx, eax
lea rax, ds:0[rsi*4]
and eax, 3Ch
movzx eax, word ptr [rdx+rax+1Ch]
ror ax, 8
and eax, 3FCh
sar eax, 2
mov edx, eax
shl eax, 4
shr dl, 4
or edx, eax
mov eax, edx
and edx, 33h
and eax, 0CCh
shl edx, 2
sar eax, 2
or eax, edx
mov esi, eax
mov edx, eax
and esi, 0AAh
and edx, 55h
sar esi, 1
add edx, edx
mov eax, esi
or eax, edx
movzx eax, al
cmp ax, cx
jz short loc_401210
```

Rys. 3, przetwarzanie pakietów w funkcji my_packet_handler

Na pierwszy rzut oka można zauważyć, że pod adresami “rdx+Xh” (zaznaczone na czerwono) są umieszczone pewne zmienne lokalne, które zostają przekazane do odpowiednich rejestrów. W celu analizy co dokładnie kryją przeprowadzono analizę dynamiczną funkcji my_packet_handler za pomocą programu radare2. W celu uproszczenia zadania uruchomiono program wireshark [2] i zrobiono zrzut kilku pierwszych pakietów jako hexdump. Następnie, dzięki analizie dynamicznej sczytano wartości, na jakie wskazują: [rdx+0Eh], [rdx+12h] oraz [rdx+rax+1Ch], widoczne na rys. 3.

Dzięki odczytaniu wartości z rejestrów dla kilku pierwszych pakietów sprawdzono, że występują one w polach Identification pakietów IP oraz Window size segmentu tcp. Dzięki analizie kodu na rys. 3 stwierdzono, że pola Identification oraz window size są przetwarzane za pomocą operacji bitowych, a następnie ich wartości są porównywane. W kolejnym kroku sprawdzono jaką wartość przyjmuje rejestr eax (pierwotnie zawierał wartość pola Identification) po wykonaniu na nim operacji:

- and z 0x3C - linia 4 na rys. 3
- przesunięcia bitowego o 2 w prawo - linia 5 na rys.3

Okazało się, że jest to wartość hex będąca printowalnym znakiem (należy zaznaczyć, że przetwarzano zrzut zawierający ukryte dane). To zasugerowało, że w tym miejscu mogą być odkodowywane ukryte dane. Wiedząc, że w tym miejscu mogą być ukrywane dane postanowiono oskryptować proces debugowania. W tym celu uruchomiono serwer gdb za pomocą polecenia (gdzie test.bin jest aplikacją detektora):

gdbserver 127.0.0.1:3333 test.bin ../pcaps/pcap05.pcap

Następnie połączono się do niego za pomocą skryptu w języku Python:

```
import r2pipe

exe_filename = '/root/Downloads/best/majewski_natur/detector/test.bin'
r2 = r2pipe.open('gdb://127.0.0.1:3333',
                ['-d', '-b 16', '-e dbg.exe.path=' + exe_filename])
r2.cmd('db 0x00401210') # ustawienie breakpointu

with open('antygonia-decoded.txt', 'w') as to_write:
    to_write.write("")

while True:
    r2.cmd('dc') # kontynuacja działania programu do adresu breakpoint
    char = chr(eval(r2.cmd('dr?rcx'))) # odczytanie wartości rejestru rcx
    with open('antygonia-decoded.txt', 'a') as to_write:
        to_write.write(char)
```

Skrypt realizował prostą funkcjonalność - przechodził do momentu w programie, w którym rejestr rcx przechowywał printowalny znak, a następnie

zapisywał go do pliku "antygona-decoded.txt". Po zakończeniu działania skryptu plik antygona-decoded.txt wyglądał jak na rys. 4, a więc odkodowanie przesyłanych informacji zakończyło się pełnym sukcesem.

```
Dzielnie warujesz i wa3ujesz sprawê;  
Lecz jasne, że co<9c> przynosisz nowego.  
  
Strażnik  
  
Bo to niesporo na plac ze z3 1 wie<9c>ci1.  
  
Kreon  
  
Lecz mów już w końcu i wyno<9c> się potem!  
  
Strażnik  
  
A więc już powiem. Trupa kto<9c> co tylko  
Pogrzeba3 skrycie i wynios3 się chy3kiem;  
Rzuci3 gar<9c>æ ziemi i uczci3 to cia3o.  
  
Kreon  
  
Co mówisz? Któż by3 tak bardzo bezczelny?  
  
Strażnik  
  
Tego ja nie wiem, bo żadnego znaku  
Topora ani motyki nie by3o.  
Ziemia woko3o by3a g3adka, zwarta,
```

Rys.4, odkodowana Antygona

Stworzony skrypt nie jest doskonały, bo np. nie wyświetla polskich znaków. Ale pozwolił on na całkowite poznanie metody jaką były ukrywane dane i odkodowanie przesyłanych danych.

Reasumując, metoda steganograficzna polegała na tym, że:

1. W polu identification umieszczany był znak z antygony. Konkretnie był on umieszczony w bitach 7-14 pola Identification.
2. Ten sam znak był również kodowany w polu window size za pomocą większej ilości operacji arytmetycznych, które widać na rys. 3
3. Jeżeli zrzut zawierał tajne dane, to istniała możliwość odkodowania tej samej literki z pola identification oraz window size. Jeśli te wartości były równe, (zawierały ten sam znak) to oznaczało, że jest to ukryty znak.

Ukryte dane można odczytać używając dostarczonego detektora oraz przedstawionego powyżej skryptu - spełnia to w zupełności swoje zadanie. Jednak,

jeśli ktoś chciałby to zrobić za pomocą jednego programu, to wystarczy napisać skrypt Pythona przetwarzający pakiety, wzorując się na kodzie asemblera widocznym na rys.3.

Dla ścisłości warto dodać, że detektor wskazywał, że zrzut zawiera tajne dane, tylko jeśli ponad 100 pakietów po przetworzeniu (przez operacje arytmetyczne na rys. 3) zawierało takie same wartości Identification i window size.

Na koniec należy zaznaczyć, że komunikacja pomiędzy hostami odbywała się na dwóch zupełnie różnych adresach publicznych. Rozważany zespół wyjątkowo wykonał projekt zgodnie z oficjalnymi założeniami pod każdym kątem, dlatego naszym zdaniem powinien on zostać dodatkowo nagrodzony.

3. Przytuła i Pelka

Socjotechnika

W przypadku zespołu numer 2 zastosowano techniki socjotechniczne. Ustalono, że atak przygotowany był pod presją czasu, co sugeruje, że informacje pozyskane za pomocą inżynierii wstecznej wskazują na szkodliwy ruch (nie zostało użyte dla zmylenia przeciwnika). Wniosek ten argumentuje opłacalność wykonania inżynierii wstecznej, która w przypadku zastosowania przez zespół numer 2 język GO jest wyjątkowo pracochłonna. Podczas drugiego podejścia socjotechnicznego dowiedziano się, że trop znaleziony za pomocą inżynierii wstecznej jest prawidłowy.

Inżynieria wsteczna i analiza automatyczna

Jak wspomniano zespół nr 2 wyróżnił się tym, że napisał detektor w języku GO. Było to znaczące utrudnienie inżynierii wstecznej z uwagi na to, że:

- GO jest kompilowany i plik wykonywalny praktycznie nie zawiera informacji debugujących
- narzędzia do inżynierii wstecznej współpracują najlepiej z C i C++, a GO ma sporo informacji nakładkowych (względem C) wydłużających analizę
- GO mocno wspiera programowanie równoległe, a to niesamowicie utrudnia analizę dynamiczną
- IDA w wersji 7.0 nie wspierała właściwie plików binarnych GO

W tym przypadku ponownie próbowano dekompilacji za pomocą wielu narzędzi i technik, jednak rezultat był jeszcze gorszy niż w poprzednim przypadku. Dlatego ponownie zdecydowano się na deasemblację. W poprzednim zadaniu użyto programu IDA do analizy statycznej i radare2 do analizy dynamicznej. W tym przypadku zmieniono narzędzia na radare2 do analizy statycznej oraz gdb do analizy dynamicznej, ponieważ IDA nie analizowała odpowiednio pliku binarnego, a radare2 miał duże problemy z wielowątkowością podczas analizy dynamicznej.

Analiza statyczna pozwoliła ustalić, że w programie jest używana funkcja SetBPFFilter, która jako argument przyjmuje "dst host 192.168.1.45", co przedstawiono na rys. 5. (tutaj obsługa stringów jest inna niż w przypadku C++)

```

0x505b79 ;[gp]
; [0x2e0:8]=-1
; 736
mov rax, qword [local_2e0h]
mov qword [rsp], rax
; "dst host 192.168.1.45findrunnable: wrong pgcprocs inconsistency"
lea rcx, [0x00576e3f]
mov qword [local_8h], rcx
; [0x15:8]=-1
; 21
mov qword [local_10h], 0x15
call sym.github.com_google_gopacket_pcap.__Handle_.SetBPFFilter;[gn]

```

Rys. 5, filtracja pakietów

Po zapoznaniu się z kodem funkcji SetBPFFilter okazało się, że filtruje ona pakiety i odrzuca wszystkie, w których adresem docelowym nie jest 192.168.1.45. Dało to pewną wskazówkę na temat działania programu, jednak reszta była zbyt zawiła (i za długa), dlatego rozpoczęto analizę dynamiczną. Podczas analizy zauważono, że wartość zwracana (czyli stan rejestru) funkcji Endpoint.String to adres IP, który następnie był porównywany jak na rys. 6.



Rys. 6, Sprawdzenie adresu źródłowego pakietu

Z obrazka można wyczytać string adresu IP przekonwertowany do wartości hex (zaznaczone na czerwono). Prawdopodobnie z uwagi na długość, porównanie zostało rozłożone na 2 warunki. W każdym razie ten fragment kodu sprawdzał, czy adres hosta to 192.168.1.36, co można odczytać z rys. 6. Jeżeli warunek był prawidłowy to funkcja kontynuowała swoje działanie, a jeśli nie, to następowało przejście do analizy następnego pakietu.

Należy zaznaczyć, że w kodzie było jeszcze jedno porównanie adresu IP, sprawdzające, czy adres hosta to 192.168.1.45, jednak wydaje się to redundantne ze względu na to, iż i tak na początku został ustawiony filtr, który odrzucał pakiety bez udziału 192.168.1.45. Dla pewności sprawdzono, czy pakietów zakładających komunikację między adresami 192.168.1.36 i 192.168.1.45 jest tyle samo co wywołań funkcji oznaczającej spełnienie obu wyżej omówionych warunków. Wykonano to za pomocą programu gdb oraz poniższego skryptu pythona:

```
import dpkt
import socket

if __name__ == "__main__":
    pcap_name = '/home/viva/Pulpit/best/pelka_przytula/pcaps/dump2.pcap'
    with open(pcap_name) as pcap_file:
        pcap = dpkt.pcap.Reader(pcap_file)
        i = 0
        j = 0
        for ts, buf in pcap:
            i += 1
            eth = dpkt.ethernet.Ethernet(buf)
            if not isinstance(eth.data, dpkt.ip.IP):
                continue
            ip = eth.data

            if socket.inet_ntoa(ip.dst) == "192.168.1.45" and \
               socket.inet_ntoa(ip.src) == "192.168.1.36":
                j += 1
        print(j)
```

Skrypt pythona zwrócił, że liczba pakietów to 1514. Dlatego oczekiwano, że program tyle samo razy uderzy w odpowiedni breakpoint. Jak widać na rys. 7, 1513 kontynuacji działania programu nie spowodowało zakończenia programu, a zrobiło to 1514. (podobne działania wykonano dla kilku innych zrzutów). W ten sposób potwierdzono poprawność dotychczasowej analizy.

```
(gdb) c 1513
Will ignore next 1512 crossings of breakpoint 1. Continuing.

Thread 3 "main" hit Breakpoint 1, 0x0000000000506306 in main.n
56      in /home/marek/go/src/go-pcapAnalysis/main.go
(gdb) c
Continuing.
Hidden data is present in file.
[Thread 0x7ffff6dbc700 (LWP 1601) exited]
[Thread 0x7ffff75bd700 (LWP 1600) exited]
[Inferior 1 (process 1596) exited normally]
```

Rys. 7, analiza dynamiczna (gdb)

Ostatni fragment kodu, przez który przechodził pakiet, sprawdzał pewną wartość dotyczącą pakietu i porównywał ją z 0x4A, jeśli wartość była zgodna, to zapalana była odpowiednia flaga. Trudno było odgadnąć o jaką wartość chodzi, bo próba prześledzenia działania całego programu bit po bicie byłaby jeszcze bardziej pracochłonna. Dlatego próbowano znaleźć rzeczone 0x4A w bajtach któregoś pakietu, ostatecznie okazało się, że 0x4A odnosi się do długości całego pakietu. Jeśli długość któregoś z przetwarzanych pakietów wynosiła 0x4A (74), to program zwracał komunikat, że pcap zawiera ukryte dane. Co ciekawe program nie zatrzymywał się od razu po sprawdzeniu, że taki pakiet istnieje, tylko i tak przetwarzał wszystkie pakiety i po sprawdzeniu ostatniego pakietu sprawdzał odpowiednią flagę.

Również sprawdzono to doświadczalnie przez to, że załadowano plik pcap nie zawierający tajnych danych, ale posiadający przynajmniej jeden pakiet, w którym adresy IP to 192.168.1.45 i 192.168.1.36. Po wejściu do odpowiedniej instrukcji w funkcji i pojedynczej zamianie wartości odpowiedniego rejestru na 0x4A okazało się, że program, iż program zwraca, że pcap zawiera ukryte dane. Potwierdziło to poprawność przyjętego toku rozumowania.

Reasumując, program zwraca informację, że pcap zawiera ukryte dane, tylko jeśli istnieje przynajmniej 1 pakiet, w którym adresy ip to 192.168.1.36 i 192.168.1.45 oraz długość ramki łącznie z nagłówkiem Ethernet to 0x4A (74) B. Nie jest to zgodne z założeniami projektu, ponieważ ruch powinien być wysyłany poza sieć lokalną. W tym przypadku serwer atakującego nawet nie jest w innej sieci adresów prywatnych co mogłoby modelować adres publiczny. Co ciekawe atakujący umieszczają komunikację z serwerami korzystającymi z adresów publicznych w ruchu podkładowym. De facto właśnie ten ruch powinien zostać wykorzystany jako nośnik.

Analiza manualna

Po zastosowaniu filtra (`ip.addr == 192.168.1.36 and ip.addr == 192.168.1.45`) and `frame.len == 74` w programie Wireshark zaobserwowano, że tajne dane przesyłane są do hosta 192.168.1.36 (dalej nazywanego hostem A) do 192.168.1.45 (dalej nazywanego hostem B). Po zmianie filtra na `ip.src == 192.168.1.36 and ip.dst == 192.168.1.45 and !frame.len == 74` liczba wyświetlonych pakietów wynosiła 19, co wskazuje na niewielką liczbę wysyłanych ramek z A do B o wielkości innej niż 74B. Dla porównania liczba ramek wysyłanych w tej relacji o długości 74B wynosiła 1661. W plikach nie zawierających ukrytej transmisji danych liczba takich ramek również była znikoma.

Zakładając, że atakujący przesyła minimum danych wymaganych do zaliczenia projektu (10kB) to w każdym pakiecie musi przesłać średnio ok 8B (w poszczególnych pcapach liczba pakietów będąca komunikacją pomiędzy hostem A i B była porównywalna). Fakt ten sugeruje, że dane muszą być przesyłane w wielu polach jednocześnie, bądź w polu danych. Po analizie nagłówków IP oraz TCP zrezygnowano z pierwszej możliwości. Pole danych TCP zaś ma długość dokładnie 8B w każdym z segmentów. Niestety dane te są zaszyfrowane (w pliku `dump2.pcap` zaobserwowano zestawienie sesji SSL), co uniemożliwiło odczytanie Antygony z przechwyconego ruchu.

4. Milczarek i Bańka

Inżynieria wsteczna

Zespół nr 6 napisał swój detektor w języku technologii .NET, która jest w pełni odwracalna na podstawie informacji debugowania. Dzięki temu dekompilacja wymagała jedynie podania pliku exe do dekompiłatora "dotPeek". Na podstawie tego dowiedziano się, że sprawdzanie, czy dany pcap zawiera ukryte dane odbywa się za pomocą porównania hashy md5. Program jest dosyć długi, dlatego na rys.8. przedstawiono jedynie zbiór porównywanych hashy md5.

```
public class FileCheckerViewModel : PropertyChangedBase
{
    private List<string> desiredHashes = new List<string>()
    {
        "0ad788ad4e16fd9742f75b1b4b85ddac",
        "51f54b89be8edbee118e30250bc38d33",
        "1ac9826e88213009f1875217ac5a192b",
        "40d97917ddeac6b222270a78714c6c39",
        "481caec15e4db661014e7392470b0448",
        "d48a9970c01d8236130d52c129cb093b",
        "c7f5c7c15714e661af19911ae0e4adbe"
    };
    private List<string> pcaps;
```

Rys. 8, hashe pcapów zawierających tajne dane

Warto dodać, że ręcznie upewniono się, iż hashe pcapów zawierających tajne dane pokrywają się z tymi widocznymi powyżej.

Analiza manualna i automatyczna

Podczas manualnej analizy pakietów zauważono, że niektóre segmenty zawierają ustawioną flagę *marker* protokołu RTP. Segmenty te wymieniane są pomiędzy hostami 192.168.0.102 i 192.168.0.106 w przypadku zrzutów zawierających tajne dane. W przypadku pozostałych zrzutów (nie zawierających tajnych danych) komunikacja odbywa się pomiędzy innymi hostami, co prawdopodobnie nie jest zgodne z założeniami projektu. Wracając do flagi *marker*, uznano, że eksfiltrowane dane mogą być jawnie przesyłane za pomocą flagi *marker* - przykładowo obecność flagi=bit 1, brak flagi=bit 0. Skrypt sprawdzający ten pomysł jest widoczny poniżej:

```

import dpkt

def main():
    res = ""
    pcap_name = '/home/viva/Pulpit/best/milczrek_bank/Zrzuty/3.pcap'
    with open(pcap_name, 'rb') as pcap_file:
        pcap = dpkt.pcap.Reader(pcap_file)
        for ts, buf in pcap:
            eth = dpkt.ethernet.Ethernet(buf)
            if isinstance(eth.data, dpkt.ip.IP):
                ip = eth.data
                if isinstance(ip.data, dpkt.udp.UDP):
                    udp = ip.data
                    if udp.dport != 5060:
                        marker = udp.data[1] >> 7
                        res += str(marker)
    return res

if __name__ == "__main__":
    marker_bits = main()
    print(marker_bits)

```

Po uruchomieniu skryptu okazało się, że wynik zawiera zbyt długie ciągi bitów zerowych, aby możliwe było odkodowanie danych. Wynika z tego, że dane były, albo w pewien sposób obrabiane, albo flaga *marker* była używana do innych celów.

Ponadto segmentów z ustawioną flagą *marker* jest zbyt mało (7k w pliku numer 3), aby można było przesłać 10kB danych za jej pomocą. Jest to możliwe jedynie w przypadku zastosowania kompresji danych. Jednak zakładamy, że taka technika nie została wykorzystana. Niezależnie od tego, czy powyższa flaga została użyta do przenoszenia całości/części danych, czy jedynie oznacza pakiety przenoszące dane w innych polach to powoduje ona, że pakiety te wyróżniają się na tle standardowego ruchu. W efekcie zdobyto pewność, że pakiety te należy odrzucić w sieci, co uniemożliwi atakującemu przeprowadzenie ataku.

Kolejną obserwacją jest to, że segmenty przesyłane z hosta 192.168.0.106 do hosta 192.168.0.102 mają złą sumę kontrolną protokołu UDP (do tego celu należało włączyć sprawdzanie sumy kontrolnej w opcjach wiresharka). Wywnioskowano, że zainfekowanym hostem jest zatem 192.168.0.106 i należy go wyleczyć. Takich

pakietów jest 5040 i każdy z nich może przenosić 4B ukrytego ruchu, więc ustalono, iż jest to nośnik ataku.

W przypadku tego projektu również należy zwrócić uwagę, że ukryta komunikacja odbywała się w ramach sieci lokalnej, co nie jest zgodne z założeniami projektu. Dlatego eksfiltracja danych nie mogłaby mieć miejsca w rzeczywistym przypadku. Umieszczenie hosta przyjmującego eksfiltrowane dane w obrębie sieci lokalnej jest raczej nieprzydatne.

5. Bąk i Krzemiński

Inżynieria wsteczna

Zespół nr 3 swój detektor napisał w Javie, która również jest w pełni odwracalna. Do odzyskania kodu źródłowego użyto jednego dekompilatora javy [1]. Rezultat jest widoczny na rys. 9. Odczytano z niego, że detektor sprawdza, czy zrzut zawiera więcej niż 10 pakietów korzystających z portu 122 oraz więcej niż 10 pakietów używających portu 150. Jeśli tak, to oznacza, że zrzut zawiera tajne dane.

```
public static void main(String[] args) throws IOException {
    Pcap pcap = Pcap.openStream(args[0]);
    streams = new ArrayList();
    pcap.loop(new PacketHandler(){

        public boolean nextPacket(Packet packet) throws IOException {
            if (packet.hasProtocol(Protocol.TCP)) {
                TCPpacket tcpPacket = (TCPpacket)packet.getPacket(Protocol.TCP);
                int port = tcpPacket.getDestinationPort();
                if (port == startPort) {
                    counter1++;
                } else if (port == endPort) {
                    counter2++;
                }
            }
            return true;
        }
    });
    if (counter1 > 10 && counter2 > 10) {
        System.out.println("Zawiera ukryte dane");
    } else {
        System.out.println("Nie zawiera ukrytych danych");
    }
}

static {
    counter1 = 0;
    counter2 = 0;
    max = 0;
    progress = false;
    startPort = 122;
    endPort = 150;
    finishPort = 68;
}
```

Rys. 9, zdekompilowany program detektor (Java)

Analiza manualna

Zgodnie z wnioskami po inżynierii wstecznej, w programie Wireshark zastosowano filtr (tcp.port == 150 or tcp.port == 122). Fragment zrzutu widać na rys. 10, komunikacja odbywa się jedynie pomiędzy 192.168.0.X a 10.0.2.15. W zależności od zrzutu otrzymano od 400 do nawet 130 000 pakietów. Jest to znaczna różnica pokazująca, że tajne dane przesyłane są w niewielkiej liczbie pakietów, a co za tym idzie przesyłane muszą być w payloadzie. Nawet jeśli wszystkie 400-sta

pakietów przenosiłoby tajne dane to każdy z nich musiałby przenosić 25B danych. Adres IP hosta nasłuchującego na portach 122 i 150 ma wartość 10.0.2.15, co zrozumiane zostało jako „oznaczenie sieci publicznej” adresem z innej sieci prywatnej. Co prawda po użyciu filtru `!ip.src == 192.168.0.0/16 and !ip.src == 10.0.0.0/8` otrzymujemy pakiety skierowane na adres publiczny 104.17.152.222, jednak projekt nie umożliwia dodawania transmisji mającej na celu odwrócenie uwagi od kanału przesyłania skradzionych danych (dodatkowych potencjalnych kanałów). Dodatkowo sesję z serwerem 104.17.152.222 utrzymuje host 10.0.2.15, a w założeniach projektu atakujący przejął jednego hosta, więc nie może jednocześnie dodawać ruchu pochodzącego od dwóch różnych hostów. Co więcej ruch przesyłany na porty 122 i 150 jest tym bardziej dziwny, że nie są z nim związane żadne stałe połączenia TCP (brak flag SYN), a jedynie wysyłane są pakiety z danymi oraz host ten nie odpowiada na otrzymane wiadomości.

Kolejną niezwykle ciekawą informacją jest to, że w komunikacji istnieje specyficzny spoofing. Rozważanie należy zacząć od tego, że (według normalnej komunikacji) adresowi IP 10.0.2.15 odpowiada adres fizyczny 08:00:27:83:13:a8. Z kolei adres IP 104.17.154.222 jest mapowany na 52:54:00:12:35:02. Biorąc to pod uwagę i analizując rys. 9 można zauważyć, że na adres 10.0.2.15 przesyłane są pakiety z dwóch adresów fizycznych z czego jeden z nich jest adresem fizycznym 10.0.2.15! Wygląda to jakby do adresu 10.0.2.15 (w teorii) miał dochodzić spoofing zarówno z 104.17.154.222 jak i 10.0.2.15. Oznaczałoby to, że 10.0.2.15 spoofuje pakiety wysyłane na własny adres IP. Jest to co najmniej dziwne i nie do końca wiadomo jak to interpretować. Z jednej strony spoofowanie adresów IP mogłoby być ciekawym zabiegiem steganograficznym, zgodnie z rys. 10 ostatni oktet adresu 192.168.0.X mógłby być używany do przenoszenia tajnych danych, jednak z uwagi na to, że spoofing z adresu publicznego na adres prywatny nie jest możliwy (router wyciąłby ten ruch), to wywnioskowano, że nie jest to możliwe w rzeczywistym środowisku.

eth-src	eth-dst	Source	Destination	sport	dport
52:54:00:12:35:02	08:00:27:83:13:a8	192.168.0.146	10.0.2.15	80	122
52:54:00:12:35:02	08:00:27:83:13:a8	192.168.0.37	10.0.2.15	80	150
08:00:27:83:13:a8	52:54:00:12:35:02	192.168.0.62	10.0.2.15	44930	122
52:54:00:12:35:02	08:00:27:83:13:a8	192.168.0.68	10.0.2.15	80	150
52:54:00:12:35:02	08:00:27:83:13:a8	192.168.0.130	10.0.2.15	80	122
08:00:27:83:13:a8	52:54:00:12:35:02	192.168.0.161	10.0.2.15	44930	150
52:54:00:12:35:02	08:00:27:83:13:a8	192.168.0.105	10.0.2.15	80	122
08:00:27:83:13:a8	52:54:00:12:35:02	192.168.0.249	10.0.2.15	44930	150
52:54:00:12:35:02	08:00:27:83:13:a8	192.168.0.220	10.0.2.15	443	122
52:54:00:12:35:02	08:00:27:83:13:a8	192.168.0.239	10.0.2.15	80	150
08:00:27:83:13:a8	52:54:00:12:35:02	192.168.0.185	10.0.2.15	44930	122
52:54:00:12:35:02	08:00:27:83:13:a8	192.168.0.72	10.0.2.15	80	150
08:00:27:83:13:a8	52:54:00:12:35:02	192.168.0.23	10.0.2.15	44930	122
08:00:27:83:13:a8	52:54:00:12:35:02	192.168.0.178	10.0.2.15	44930	150
08:00:27:83:13:a8	52:54:00:12:35:02	192.168.0.123	10.0.2.15	44930	122
52:54:00:12:35:02	08:00:27:83:13:a8	192.168.0.79	10.0.2.15	80	150

Rys. 10, fragment komunikacji zawierającej spoofing

Podsumowując dostarczone przez zespół 5 zrzuty są przygotowane w sposób wprowadzający w rozdarcie wewnętrzne detekujących, ponieważ z jednej strony skradzione dane powinny być wysyłane na adres publiczny (104.17.152.222), z drugiej w sieci pojawiła się anomalia ruchowa w postaci segmentów wysyłanych z sieci 192.168.0.0/24 do hosta 10.0.2.15. Dlatego skłoniono się ku stwierdzeniu, że zrzuty ruchu prawdopodobnie nie spełniają do końca założeń projektowych.

6. Muczyński i guzek

Inżynieria wsteczna

Wszystkie pozostałe detektory były napisane języku Python. Odzyskanie kodu źródłowego w tym przypadku było w pewnym stopniu trudniejsze niż w Javie, czy C#. Ale na pewno nieporównywalnie prostsze niż w C++ i GO.

Schemat dekompilacji jest bardzo podobny w przypadku wszystkich skryptów Pythona, a więc dotyczy również rozdziałów 7 i 8. Do ułatwienia dekompilacji ściągnięto programy: `pyi-archive_viewer` oraz `uncompyle6`, dodatkowo ściągnięto skrypt `pyinstxtractor.py`. Schemat dekompilacji był następujący:

1. Załadowanie pliku wykonywalnego do `pyinstxtractor.py` (lub jeśli to nie zadziała to znacznie lepszy jest `pyi-archive_viewer`).
2. W powstałym katalogu pojawi się główny plik bez rozszerzenia, na potrzeby przykładu używana będzie nazwa pliku jako "main".
3. Zamiana początkowych bajtów pliku main na te odpowiadające innym plikom .pyc, tzw. magiczne numery. Można to zrobić na przykład wykorzystując linuksowy program "dd".
4. Zmiana nazwy pliku z main na main.pyc
5. Załadowanie pliku main.pyc do `uncompyle6`

W rezultacie otrzymywany jest kod źródłowy w języku Python, jak na rys. 11. Program detektora informuje, że ruch zawiera ukryte dane, tylko jeśli w wiadomości Server Hello (protokół TLS) niesiona jest informacja, że używane szyfrowanie to "AES_128_CBC_SHA256".

```

# uncompyle6 version 3.1.1
# Python bytecode 3.6 (3379)
# Decompiled from: Python 2.7.14+ (default, Mar 13 2018, 15:23:44)
# [GCC 7.3.0]
# Embedded file name: D:\studia\best\projekt\main.py
import dpkt, sys

def main(pcap_file):
    TLS_HANDSHAKE = 22
    with open(pcap_file, 'rb') as (fd):
        pcap_reader = dpkt.pcap.Reader(fd)
        for ts, buf in pcap_reader:
            eth = dpkt.ethernet.Ethernet(buf)
            if not isinstance(eth.data, dpkt.ip.IP):
                continue
            ip = eth.data
            if not isinstance(ip.data, dpkt.tcp.TCP):
                continue
            tcp = ip.data
            if not (tcp.dport != 443 and tcp.sport != 443):
                pass
            if not len(tcp.data) <= 0:
                if tcp.data[0] != TLS_HANDSHAKE:
                    pass
                else:
                    try:
                        records, bytes_used = dpkt.ssl.tls_multi_factory(tcp.data)
                    except (dpkt.ssl.SSL3Exception, dpkt.dpkt.NeedData):
                        continue

                    if len(records) <= 0:
                        pass
                    else:
                        for record in records:
                            if record.type == 22 and len(record.data) != 0 and record.data[0] == 2:
                                try:
                                    handshake = dpkt.ssl.TLSHandshake(record.data)
                                except dpkt.dpkt.NeedData:
                                    continue

                                if isinstance(handshake.data, dpkt.ssl.TLSClientHello):
                                    ch = handshake.data
                                    tls_ver = dpkt.ssl.ssl3_versions_str[ch.version]
                                    cipher_suite = dpkt.ssl.ssl_ciphersuites.BY_CODE[ch.cipher_suite]
                                    if ch.version == 771:
                                        if 'AES_128_CBC' in cipher_suite.name:
                                            return True
                                continue

            return False

if __name__ == '__main__':
    pass
if len(sys.argv) != 2:
    print(('Usage: {} pcap_file').format(sys.argv[0]))
else:
    if main(sys.argv[1]):
        print('Given file contains secret data.')
    else:
        print("There isn't any secret.")
# okay decompiling main1.pyc

```

Rys. 11, zdekompilowany program detektor (Python)

Analiza manualna i automatyczna

Pierwszą rzeczą jaką zanotowano podczas przeglądania pakietów jest to, że czas używany do wygenerowania losowej liczby jest dosyć losowy, rys. 12.

```
▼ TLSv1.2 Record Layer: Handshake Protocol: Server Hello
  Content Type: Handshake (22)
  Version: TLS 1.2 (0x0303)
  Length: 61
  ▼ Handshake Protocol: Server Hello
    Handshake Type: Server Hello (2)
    Length: 57
    Version: TLS 1.2 (0x0303)
    ▼ Random: 31d94b2567a3303bc23b31baaf854f96f41e5e1e18b5dad0...
      GMT Unix Time: Jul  2, 1996 18:15:33.000000000 CEST
      Random Bytes: 67a3303bc23b31baaf854f96f41e5e1e18b5dad0b62d6ccb...
```

Rys. 12, pakiet TLS (ruch zawierający tajne dane)

Zostało to porównane z realnym ruchem sieciowym, w którym czas ten był zgodny z aktualnym. Świadczy to o pewnej anomalii w ruchu sieciowym wygenerowanym przez zespół. Istnieje również możliwość, że czas ten jest używany do steganografii jako pewien klucz lub inna informacja. Poprawnie skonfigurowany firewall powinien wykryć taką anomalię i odrzucić ruch.

Kolejną informacją jaką zanotowano jest to, że długości payloadów (w ruchu zawierającym tajne dane) niesionych przez TLS powtarzają się i przyjmują “ładne” wartości. Skłoniło to nasz zespół do zastanowienia się, czy ma to związek z paddingiem. W związku z tym, że używany szyfr jest 128 bitowy, to optymalny padding powinien powodować, iż długości payloadów (w bajtach) niesionych przez TLS będą podzielne przez 16. Zostało to sprawdzone eksperymentalnie za pomocą skryptu:

```
import dpkt

def main(pcap_nr):
    APPLICATION_DATA = 23
    pcap_name = '/home/viva/Pulpit/best/muczynski_guzek/pcaps/{0}.pcap'.format(pcap_nr)
    with open(pcap_name, 'rb') as pcap_file:
        pcap = dpkt.pcap.Reader(pcap_file)
        i = 0
        j = 0
        for ts, buf in pcap:
            i += 1
            eth = dpkt.ethernet.Ethernet(buf)
            if isinstance(eth.data, dpkt.ip.IP):
                ip = eth.data
                if isinstance(ip.data, dpkt.tcp.TCP):
                    tcp = ip.data

                    if tcp.dport == 443 or tcp.sport == 443:
                        if len(tcp.data) > 0:
                            if tcp.data[0] == APPLICATION_DATA:
                                try:
                                    ssl_packets, _ = dpkt.ssl.tls_multi_factory(tcp.data)
```

```

except (dpkt.ssl.SSL3Exception, dpkt.dpkt.NeedData):
    continue
for ssl_packet in ssl_packets:
    if (len(ssl_packet.data)) % (128 / 8) != 0:
        return False
return True

if __name__ == "__main__":
    for i in range(1, 15):
        if main(i):
            print("pcap {0} padding IS optimal".format(i))
        else:
            print("pcap {0} padding IS NOT optimal".format(i))

```

Rezultat działania skryptu potwierdził przypuszczenia, że tylko zrzuty zawierające tajne dane posiadają maksymalny padding. W tym miejscu należy zaznaczyć, że RFC opisujące TLS specyfikuje używanie paddingu będącego wielokrotnością długości bloku użytego do szyfrowania [3] (tutaj opisane jako maksymalny padding). Jednak to RFC jest jedynie proponowanym standardem, dlatego nie istnieje wymóg dostosowywania się do niego. Innymi słowy, tylko komunikacja zawierająca ukryte dane realizuje wymogi oficjalnego RFC opisującego TLS.

Bity paddingu wydają się być dogodnym miejscem na przesyłanie informacji steganograficznych, tak więc wywnioskowano, że tajne dane były najprawdopodobniej przesyłane w bitach realizujących padding. Jednak niemożliwe jest odzyskanie przesyłanych danych ze względu na użyte szyfrowanie, a konkretnie na protokół Diffiego-Hellmana pozwalający na sekretną wymianę tajnych kluczy.

Na koniec, wartym odnotowania jest fakt, że cała komunikacja odbywa się na tym samym hoście, a konkretnie na interfejsie loopback, co jest ewidentnym błędem.

7. Krasowski i Zdunek

Inżynieria wsteczna

Zespół nr 1 zastosował najprostsze podejście do detekcji, tzn. sprawdzał piątą cyfrę nazwy plików, rys. 13. Jeśli cyfra ta jest parzysta to zawierał on ukryte dane. Było to maksymalne uproszczenie sobie zadania.

```
uncompyle6 version 3.1.1
# Python bytecode 3.6 (3379)
# Decompiled from: Python 2.7.14+ (default, Mar 13 2018, 15:23:44)
# [GCC 7.3.0]
# Embedded file name: checker-new1.py
import os.path, sys

def readPackets():
    if len(sys.argv) == 1:
        print('Podaj sciezke do pliku')
    else:
        file_path = sys.argv[1]
        if not os.path.isfile(file_path):
            print('Plik nie istnieje')
        else:
            file_name = os.path.basename(file_path)
            try:
                number = int(file_name[4])
                if number % 2 == 0:
                    print('Transmisja zawiera dane')
                else:
                    print('Brak danych w transmisji')
            except ValueError:
                print('Niepoprawny plik')

if __name__ == '__main__':
    readPackets()
# okay decompiling main.pyc
```

Rys. 13, zdekompilowany program detektor (Python)

Analiza manualna

Na podstawie analizy manualnej nie zaobserwowano żadnych anomalii w nagłówkach IP, RTP, SIP i SDP. Po otrzymaniu wyników ataku socjotechnicznego i zawężeniu analizy do RTP stwierdzono, że tajny kanał jest realizowany poprzez różnicę w timestampach, bądź przesyłany w payloadzie RTP. Druga z tych dwóch opcji jest niedozwolona w ramach projektu, więc przypuszczono pierwszą z nich.

Socjotechnika

Korzystając z socjotechniki pozyskano informacje o protokole, w którym zespół pierwszy ukrywa tajne dane. Jest nim RTP. Pozyskano również informacje, że skradzione informacje nie są szyfrowane, a jedynie kodowane w kreatywny sposób.

8. Bocheński i Kunikowski

Inżynieria wsteczna

Kod detektora zespołu nr 5 jest widoczny na rys. 14. Inżynieria wsteczna została przeprowadzona według podobnego schematu jak w rozdziale 6 i 7.

```
import logging
logging.getLogger('scapy.runtime').setLevel(logging.ERROR)
from scapy.all import *
from enum import IntEnum
import sys, os

class Masks(IntEnum):
    """
    Custom masks.
    """
    Modified = 4

class Detector(object):
    """
    This class allows to check whether .pcap file consists of packets with hidden
    data or not.
    """

    @staticmethod
    def __packet_is_modified(packet):
        """
        Checks whether currently analyzed packet has hidden data.
        :return: True, if currently analyzed packet has hidden data, or False otherwise.
        """
        if not packet.haslayer(IP):
            return False
        ip_packet = packet.getlayer(IP)
        return ip_packet.flags & Masks.Modified != 0

    @staticmethod
    def check_pcap_file(pcap_path):
        """
        Loads a .pcap file and checks if there is hidden message in it.
        :param pcap_path: .pcap file path to be loaded
        """
        print 'Checking in progress...'
        try:
            pcap = rdpcap(pcap_path)
        except:
            print 'Oops! That is not valid file!\n'
            print 'Your file is probably in .pcapng format, rather than libpcap\n'
            print 'The solution is simple:'
            print "In Wireshark: 'Save As': Wireshark / tcpdump - pcap "
            print "Or use tshark: '$tshark -r old.pcapng -w new.pcap -F libpcap'"
            return

        for packet in pcap:
            if Detector.__packet_is_modified(packet):
                print 'There ARE hidden data in ' + os.path.basename(pcap_path) + ' file!'
                return

        print 'There ARE NO hidden data in ' + os.path.basename(pcap_path) + ' file!'

if len(sys.argv) != 2:
    print 'Usage: detector.exe <pcap path>'
    sys.exit(1)
if not os.path.exists(sys.argv[1]):
    print 'File ' + sys.argv[1] + ' does not exist'
    sys.exit(1)
Detector.check_pcap_file(sys.argv[1])
```

Rys. 14, zdekompilowany program detektor (Python)

Z analizy kodu wynika, że program informuje o ukrytych danych, jeśli w którymś z pakietów **zarezerwowany** bit fragmentacji ustawiony jest na wartość 1. Jest to oczywisty błąd grupy projektującej, ponieważ każdy podstawowo skonfigurowany firewall wykryłby, że jest to złośliwa komunikacja i w konsekwencji odrzuciłby te pakiety.

Analiza manualna

Pierwszą rzeczą na jaką zwrócono uwagę jest to, że w komunikacji zawierającej ukryte dane segment TCP zawiera nieznaną opcję, widać to na rys. 15.

```
▼ Options: (20 bytes), No-Operation (NOP), Timestamps, No-Operat
  ▶ TCP Option - No-Operation (NOP)
  ▶ TCP Option - Timestamps: TSval 3305231272, TSecr 2091737239
    Unknown (0xf5) (6 bytes)
  ▶ TCP Option - No-Operation (NOP)
  ▶ TCP Option - No-Operation (NOP)
  ▶ TCP Option - End of Option List (EOL)
```

Rys. 15, nieznana opcja 0xf5

Uznano, że w tym miejscu mogłyby być przesyłane ukryte dane, jednak po bardziej wnikliwej analizie okazało się, iż wartość 0xf5 nie jest praktycznie zmieniana, dlatego porzucono ten trop.

Kolejną anomalią, na którą zwrócono uwagę jest to, że w komunikacji zawierającej ukryte dane zapalana jest flaga URG. W konsekwencji również pole Urgent Pointer przyjmuje wybrane wartości. Zrzuty nie zawierające tajnych danych nie posiadają ustawionej flagi URG. Jawnie wskazuje to, że w tym miejscu mogą być przesyłane dane.

Ostatnią zauważoną rzeczą są dziwne wartości pola Identification pakietu IP. Jest to bardzo wygodne miejsce do przesyłania ukrytych danych (użyte już zostało przez zespół Majewski i Natur). Ponownie - zrzuty nie posiadające tajnych danych mają poprawne (przyjmują mniej więcej kolejne wartości) wartości pola Identification.

Uznano, że po wykryciu tylu nieprawidłowości, nie ma sensu przeprowadzać automatycznej analizy, ponieważ dane najpewniej są przesyłane za pomocą kombinacji pola Identification oraz Urgent Pointer. Możliwości zakodowania danych za pomocą tych dwóch pól jest praktycznie nieskończoność, dlatego nawet nie próbowano ich odkodować. Na koniec warto zaznaczyć, że wszystkie nieprawidłowości zawierały się w komunikacji od hosta 10.5.125.107 do 10.5.250.250. Tak więc zdecydowano, że host z końcówką 107 był zainfekowany.

9. Podsumowanie efektów detekcji

Zespół sporządzający niniejszy raport wnikliwie przeanalizował możliwości odkrycia metod steganograficznych stosowanych przez poszczególne zespoły projektujące. Zastosowano wiele podejść:

- inżynierie wsteczną (statyczną i dynamiczną)
- częściowe oskryptowanie procesu inżynierii wstecznej
- statyczną i porównawczą analizę manualną zrzutów ruchu sieciowego
- analizę automatyczną wybranych zrzutów ruchu sieciowego
- socjotechnikę (z której uzyskano stosunkowo dużo informacji)

Osiągnięcia w analizie metod steganograficznych zostały zobrazowane w tabelce (uznano, że rozszerzenie tabelki o dodatkową kolumnę jest wartościowe):

Metoda/Grupa projektująca	Domniemana zasada działania	Zastosowany sposób detekcji	Czy nastąpiło wysłanie danych poza daną sieć lokalną?
Metoda 1 - Krasowski, Zdunek	Kodowanie za pomocą różnicy wartości timestamp nagłówka RTP	Inżynieria wsteczna, analiza manualna, socjotechnika	Tak
Metoda 2 - Przytuła, Pelka	Dane TCP - zaszyfrowane	Inżynieria wsteczna, analiza manualna, analiza automatyczna, socjotechnika	Nie
Metoda 3 - Bąk, Krzemiński	Dane TCP - zaszyfrowane	Inżynieria wsteczna, analiza manualna	Raczej tak (zależnie od metody steganografii)
Metoda 4 - Guzek, Muczyński	Bit paddingu w payloadzie TLS	Inżynieria wsteczna, analiza manualna, analiza automatyczna	Tylko localhost!
Metoda 5 - Bocheński, Kunikowski	Pole Urgent Pointer i pole Identification pakietu IP	Inżynieria wsteczna, analiza manualna	Tak (zakładając, że maska to 24), ale celem nie był adres publiczny
Metoda 6 - Milczarek, Bańka	suma kontrolna UDP/ ew. flaga marker RTP	Inżynieria wsteczna, analiza manualna, analiza automatyczna	Nie
Metoda 7 - Majewski, Natur	Dane ukrywane z wykorzystaniem pola Identification pakietu IP oraz sprawdzane za pomocą pola Window size segmentu TCP	Inżynieria wsteczna wspomagana własnymi skryptami	Tak (pomiędzy różnymi adresami publicznymi)

Uwagi ogólne

1. Wiele zespołów używało dwóch komputerów do generowania ruchu, bez użycia wirtualizacji lub narzędzi do generowania pakietów na poziomie L2. Powodowało to, że pakiety pochodzące z różnych adresów IP miały taki sam adres MAC, co mogłoby wskazywać na spoofing, nawet jeśli nie był on założony przez atakujących. Jest to dodatkowe utrudnienie dla grupy detekujących.
2. Wiele zespołów korzysta jedynie z lokalnych adresów IP. Jest to niezgodne z założeniami i stanowi kolejne utrudnienie dla detekujących.
3. Dysproporcja pomiędzy czasem poświęconym na wykonanie projektu jest bardzo duża. Grupy detekujące chcące przedstawić dokładną analizę muszą poświęcić znacznie więcej czasu w porównaniu z projektującymi.

Bibliografia

- [1] Lee Benfield , “*Java decompiler online*”, [dostęp online],
<http://www.javadecompilers.com/>
- [2] The Wireshark team, “*Wireshark · Go Deep.*”, [dostęp online],
<https://www.wireshark.org/>
- [3] IETF, “*The Transport Layer Security (TLS) Protocol Version 1.2*”, [dostęp online],
<https://tools.ietf.org/html/rfc5246#section-6.2.3.2>