Projekt PSI - Sprawozdanie końcowe

Zespół 37 - Jakub Bąba, Angelika Ostrowska, Aleksandra Szymańska

Założenia projektu

Implementację rozwiązania w postaci architektury klient-serwer opartej na protokole TCP dostarczymy w Pythonie. Komunikacja będzie sterowana z wiersza poleceń: klienci będą mogli wybrać między zainicjowaniem połączenia z serwerem, wysłaniem wiadomości i zakończeniem połączenia z serwerem, serwer będzie mógł zamknąć połączenie dla wybranego klienta. Dodatkowo serwer będzie w stanie obsłużyć wielu klientów (ich liczba będzie przekazywana jako argument wywołania) i wyświetli tych aktualnie połączonych. Projekt będzie uruchamiany w sieci dockerowej minimalną liczbą poleceń.

Do zrealizowania projektu wybraliśmy wariant W1, a więc jako mechanizm integralności i autentyczności wykorzystamy mechanizm encrypt-then-mac dla wysyłanych szyfrowanych wiadomości.

Serwer sam będzie przydzielał klientom kolejne ID. Klient ich nie wysyła, bo nie moglibyśmy zapewnić ich unikalności.

Struktura wiadomości

- ClientHello: “ClientHello|<podstawa>|<moduł>|<klucz_publiczny_klienta>”
- ServerHello: “ServerHello|<klucz_publiczny_serwera>”
- Wiadomość
- (szyfrowana): “<długość_zaszyfrowanej_wiadomości>|<treść_wiadomości>|<MAC>”
- EndSession (szyfrowana): “{EndSession}|<MAC>”

Podstawa (liczba całkowita) i moduł (liczba pierwsza) ustalane przez klienta będą używane do ustalenia klucza sesji w algorytmie wymiany kluczy Diffie-Hellmana, który opiszemy później.
Znaki | są używane jako separatory np. między wiadomością a kodem MAC.

Używamy długości przesyłanej wiadomości, żeby mieć pewność, że nie podzielimy jej w złym momencie, jeśli w zaszyfrowanej wiadomości znajdzie się separator |.

Wykorzystane algorytmy

Do wymiany kluczy wykorzystamy algorytm Diffie-Hellman key exchange. Klient przekaże podstawę oraz moduł i razem z serwerem wymienią klucze publiczne w pierwszych wiadomościach, by ustalić wspólny klucz sesji. Odbędzie się to w następujący sposób:

Klient wygeneruje losowy klucz prywatny, ustali podstawę i moduł do obliczenia klucza publicznego i przekaże te informacje (poza kluczem prywatnym) serwerowi, który tak samo losowo wygeneruje swój klucz prywatny i użyje otrzymanych podstawy i modułu do obliczenia klucza publicznego, który odeśle klientowi.

$$\text{Klucz publiczny} = \{\text{podstawa}\}^{\{\text{klucz prywatny}\}} \bmod \{\text{moduł}\}$$

Klucz sesji dla klienta będzie obliczany ze wzoru:

$$\{\text{klucz sesji}\} = \{\text{klucz publiczny serwera}\}^{\{\text{klucz prywatny klienta}\}} \bmod \{\text{moduł}\}.$$

Analogicznie dla serwera:

$$\{\text{klucz sesji}\} = \{\text{klucz publiczny klienta}\}^{\{\text{klucz prywatny serwera}\}} \bmod \{\text{moduł}\}.$$

Z właściwości obliczeń wynika to, że klucz sesji po obu stronach będzie taki sam.

Podsumowując:

- Klient przekazuje podstawę, moduł oraz swój klucz publiczny
- Serwer w ramach odpowiedzi przesyła swój klucz publiczny
- Obie strony posiadają swoje klucze prywatne, a także są w stanie obliczyć wspólny klucz sesji

Przykładowo, rozpoczęcie komunikacji może wyglądać następująco:

- Klient ustala wartość podstawy (23) oraz modułu (5)
- Klient ustala wartość klucza prywatnego (4) i wylicza klucz publiczny (4)
- Klient przesyła wiadomość ClientHello|23|5|4
- Serwer ustala wartość klucza prywatnego (3) i wylicza klucz publiczny (10)
- Serwer odsyła wiadomość ServerHello|10
- Obie strony wyliczają wartość klucza sesji, który jest wykorzystywany później (18)

W naszej implementacji szyfrowana jest wyłącznie treść wiadomości, a nie prefiks zawierający jej długość oraz znak separujący “|”. Numer wiadomości dodawany jest do ustalonego klucza sesji, a wynik tej operacji ustawiamy jako ziarno losowości dla funkcji random.randint. Strony komunikujące się wysyłają wiadomości dopiero po otrzymaniu odpowiedzi na poprzednią. Dzięki temu zapewniamy odporne na błędy zliczanie wiadomości oraz unikalność i losowość każdego klucza OTP.
Klucz OTP to tablica losowo wygenerowanych wartości w zakresie od 0 do 255. Podczas szyfrowania każdy znak wiadomości jest zamieniany na wartość odpowiadającą mu w kodowaniu Unicode, co umożliwia wykonanie operacji XOR z odpowiadającym mu znakiem w tablicy OTP. Ze względu na tę definicję nie możemy używać ograniczonego kodowania ASCII, które obsługuje jedynie 127 znaków. Dlatego stosujemy funkcję *bytes()*, aby przekształcić zaszyfrowaną wiadomość w formę odpowiednią do przesłania za pomocą protokołu TCP.
Po otrzymaniu zaszyfrowanej wiadomości należy wykonać analogiczną operację deszyfrowania. XOR jest operatorem symetrycznym, co oznacza, że ponowne zastosowanie tej operacji na zaszyfrowanej wiadomości przy użyciu tego samego klucza pozwala odzyskać oryginalną treść wiadomości. Każdy bajt zaszyfrowanej wiadomości jest

poddawany operacji XOR z wartością tablicy OTP na odpowiadającej mu pozycji, a wynik przekształcany jest na znak. Druga strona również zlicza otrzymane komunikaty, co pozwala jej na odtworzenie klucza OTP na podstawie klucza sesji oraz numeru wiadomości i prawidłowo odczytać zawartość otrzymanej zaszyfrowanej wiadomości.

Przykładowo, dla wyżej ustalonego klucza sesji (18), oraz numeru wiadomości (1), zaszyfrowanie może wyglądać następująco:

- Klient chce przesłać wiadomość “HelloDONE”, czyli [72, 101, 108, 108, 111, 68, 79, 78, 69], wiadomość ma długość 9
- Załóżmy że za pomocą ziarna (19) można wygenerować następujący klucz OTP: [10, 15, 12, 14, 9, 11, 13, 16, 18]
- Za pomocą operacji XOR wyliczamy zaszyfrowaną wiadomość [66, 90, 96, 98, 102, 79, 66, 94, 87]
- Klient wysyła wiadomość 9|BZ`bfOB^W|<MAC>
- Serwer także znając ziarno wylicza klucz OTP, używa operacji XOR i czyta wiadomość HelloDONE

Do wyznaczenia kodu MAC użyjemy algorytmu HMAC, który wykorzystuje klucz sesji, zaszyfrowaną wiadomość i funkcję hashującą SHA-256. Będzie on miał długość 32 bajtów. Najpierw hashowany jest klucz sesji poddany XORowaniu z ustaloną wartością wewnętrznego klucza ipad (54) i skonkatenowany z wiadomością, a docelową wartość otrzymujemy przez hashowanie klucza sesji XORowanego z ustaloną wartością zewnętrznego klucza opad (92) i skonkatenowanego z wynikiem poprzedniej operacji. W praktyce używamy funkcji z modułu *hmac*.

Sposób realizacji mechanizmu integralności i autentyczności

W naszym projekcie implementujemy mechanizmy zapewniające integralność i autentyczność poprzez encrypt-then-mac.
Najpierw szyfrujemy wiadomość, a następnie dołączamy do niej kod uwierzytelnienia wiadomości. Po otrzymaniu takiej wiadomości odbiorca będzie mógł używając klucza sesji sam obliczyć kod MAC i potwierdzić autentyczność oraz integralność otrzymanego komunikatu.
Jeśli kod MAC byłby niezgodny oznaczałoby to, że nadawca nie użył poprawnego klucza sesji lub wiadomość została zmieniona w trakcie przesyłania. Poza tym, dzięki zastosowaniu szyfrowania wiadomości, po jej przechwyceniu, treść nie zostanie ujawniona.

Instrukcja użytkowania

Przygotowane skrypty umożliwiają realizację zaszyfrowanej komunikacji w modelu encrypt-then-MAC w oparciu o protokół TCP.

- Aby przygotować kontenery z serwerami/klientami, należy wpisać komendę:

`docker compose up --build -d`

- Następnie, należy uruchomić serwer:

`docker compose exec server python3 server.py [PORT=8000] [MAX_POŁĄCZEŃ=5]`

W tym oknie uruchamia się interaktywna sesja serwera, na której można wpisać komendy:

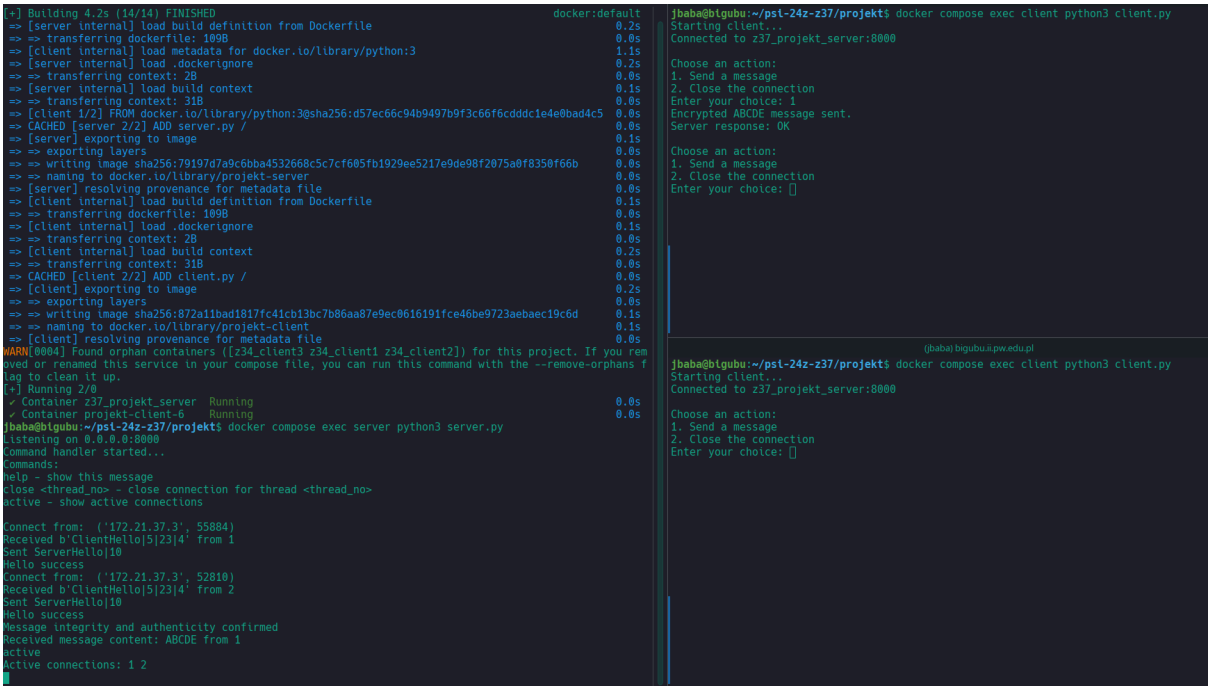
`help`	<--- wypisuje dostępne komendy
`close <thread_no>`	<--- zamyka połączenie z klientem o podanym numerze
`active`	<--- pokazuje aktywne połączenia

- W innych oknach można uruchomić klientów wpisując komendę:

`docker compose exec client python3 client.py [HOST='z37_projekt_server'] [PORT=8000]`

Pojawi się interaktywna aplikacja, w której można wysłać wiadomość, a także zamknąć połączenie.

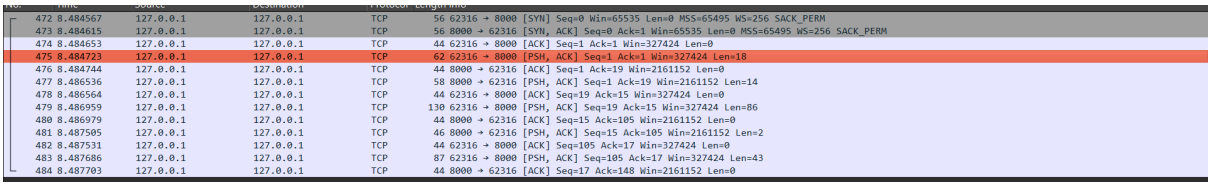
Przykładowe uruchomienie wygląda następująco:



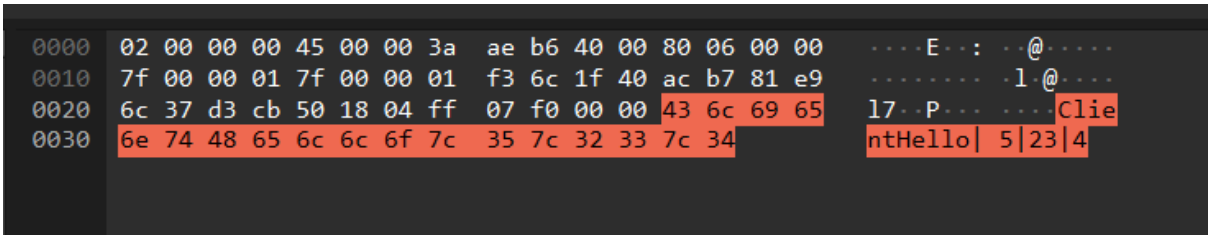
Dowód działania protokołu

Uruchomiliśmy program Wireshark i przechwytywanie pakietów z filtrem “tcp.port == 8000”. Widzimy komunikację między portem 8000 (serwer) a 62316 (klient). Poniżej przestawiamy zrzuty ekranu z poszczególnych etapów komunikacji:

1. Wiadomość ClientHello wysyłana przez klienta

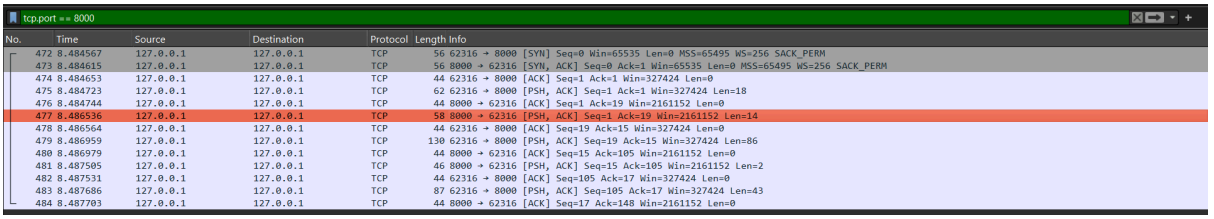


No.	Time	Source	Destination	Protocol	Length	Info
472	8.484567	127.0.0.1	127.0.0.1	TCP	56	62316 → 8000 [SYN] Seq=0 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM
473	8.484615	127.0.0.1	127.0.0.1	TCP	56	8000 → 62316 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM
474	8.484653	127.0.0.1	127.0.0.1	TCP	44	62316 → 8000 [ACK] Seq=1 Ack=1 Win=327424 Len=0
475	8.484723	127.0.0.1	127.0.0.1	TCP	62	62316 → 8000 [PSH, ACK] Seq=1 Ack=1 Win=327424 Len=18
476	8.484744	127.0.0.1	127.0.0.1	TCP	44	8000 → 62316 [ACK] Seq=1 Ack=19 Win=2161152 Len=0
477	8.486536	127.0.0.1	127.0.0.1	TCP	58	8000 → 62316 [PSH, ACK] Seq=1 Ack=19 Win=2161152 Len=14
478	8.486564	127.0.0.1	127.0.0.1	TCP	44	62316 → 8000 [ACK] Seq=19 Ack=15 Win=327424 Len=0
479	8.486959	127.0.0.1	127.0.0.1	TCP	130	62316 → 8000 [PSH, ACK] Seq=19 Ack=15 Win=327424 Len=86
480	8.486979	127.0.0.1	127.0.0.1	TCP	44	8000 → 62316 [ACK] Seq=15 Ack=105 Win=2161152 Len=0
481	8.487505	127.0.0.1	127.0.0.1	TCP	46	8000 → 62316 [PSH, ACK] Seq=15 Ack=105 Win=2161152 Len=2
482	8.487531	127.0.0.1	127.0.0.1	TCP	44	62316 → 8000 [ACK] Seq=105 Ack=17 Win=327424 Len=0
483	8.487686	127.0.0.1	127.0.0.1	TCP	87	62316 → 8000 [PSH, ACK] Seq=105 Ack=17 Win=327424 Len=43
484	8.487703	127.0.0.1	127.0.0.1	TCP	44	8000 → 62316 [ACK] Seq=17 Ack=148 Win=2161152 Len=0



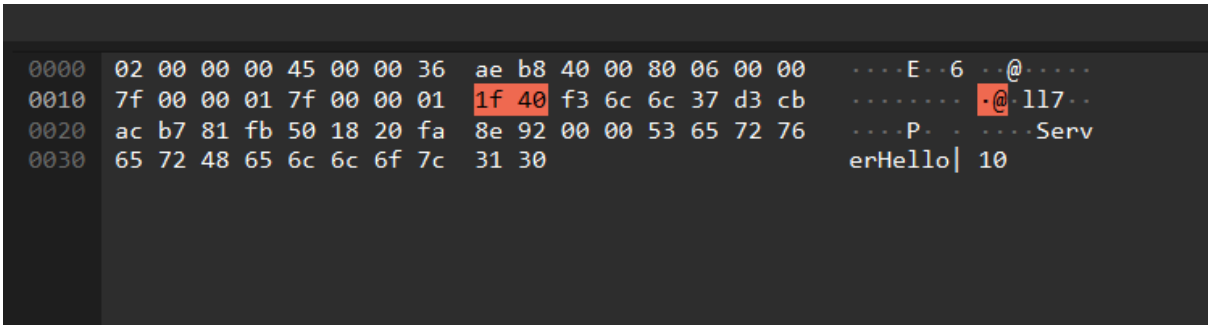
```
0000  02 00 00 00 45 00 00 3a  ae b6 40 00 80 06 00 00   ····E··:  ··@·····
0010  7f 00 00 01 7f 00 00 01  f3 6c 1f 40 ac b7 81 e9   ········  ·l·@····
0020  6c 37 d3 cb 50 18 04 ff  07 f0 00 00 43 6c 69 65   l7··P···  ····Clie
0030  6e 74 48 65 6c 6c 6f 7c  35 7c 32 33 7c 34         ntHello|  5|23|4
```

2. Wiadomość ServerHello wysyłana przez serwer



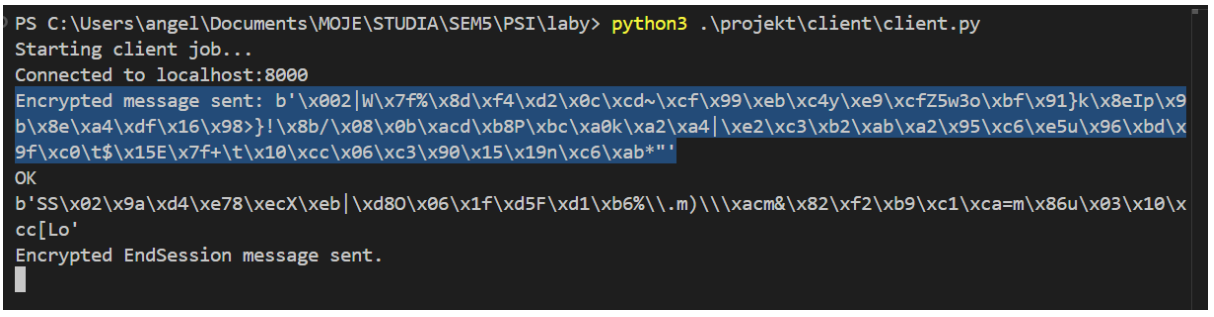
tcp.port == 8000

No.	Time	Source	Destination	Protocol	Length	Info
472	8.484567	127.0.0.1	127.0.0.1	TCP	56	62316 → 8000 [SYN] Seq=0 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM
473	8.484615	127.0.0.1	127.0.0.1	TCP	56	8000 → 62316 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM
474	8.484653	127.0.0.1	127.0.0.1	TCP	44	62316 → 8000 [ACK] Seq=1 Ack=1 Win=327424 Len=0
475	8.484723	127.0.0.1	127.0.0.1	TCP	62	62316 → 8000 [PSH, ACK] Seq=1 Ack=1 Win=327424 Len=18
476	8.484744	127.0.0.1	127.0.0.1	TCP	44	8000 → 62316 [ACK] Seq=1 Ack=19 Win=2161152 Len=0
477	8.486536	127.0.0.1	127.0.0.1	TCP	58	8000 → 62316 [PSH, ACK] Seq=1 Ack=19 Win=2161152 Len=14
478	8.486564	127.0.0.1	127.0.0.1	TCP	44	62316 → 8000 [ACK] Seq=19 Ack=15 Win=327424 Len=0
479	8.486959	127.0.0.1	127.0.0.1	TCP	130	62316 → 8000 [PSH, ACK] Seq=19 Ack=15 Win=327424 Len=86
480	8.486979	127.0.0.1	127.0.0.1	TCP	44	8000 → 62316 [ACK] Seq=15 Ack=105 Win=2161152 Len=0
481	8.487505	127.0.0.1	127.0.0.1	TCP	46	8000 → 62316 [PSH, ACK] Seq=15 Ack=105 Win=2161152 Len=2
482	8.487531	127.0.0.1	127.0.0.1	TCP	44	62316 → 8000 [ACK] Seq=105 Ack=17 Win=327424 Len=0
483	8.487686	127.0.0.1	127.0.0.1	TCP	87	62316 → 8000 [PSH, ACK] Seq=105 Ack=17 Win=327424 Len=43
484	8.487703	127.0.0.1	127.0.0.1	TCP	44	8000 → 62316 [ACK] Seq=17 Ack=148 Win=2161152 Len=0



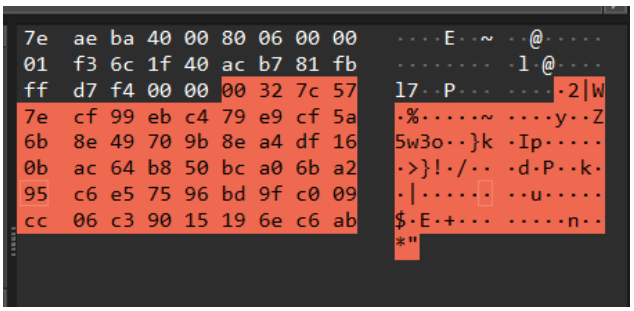
```
0000  02 00 00 00 45 00 00 36  ae b8 40 00 80 06 00 00   ····E··6  ··@·····
0010  7f 00 00 01 7f 00 00 01  1f 40 f3 6c 6c 37 d3 cb   ········  ·@·117··
0020  ac b7 81 fb 50 18 20 fa  8e 92 00 00 53 65 72 76   ····P···  ····Serv
0030  65 72 48 65 6c 6c 6f 7c  31 30                     erHello|  10
```

3. Zaszyfrowana wiadomość wysyłana przez klienta
   Widzimy, że ma ona taką postać:



```
PS C:\Users\angel\Documents\MOJE\STUDIA\SEM5\PSI\laby> python3 .\projekt\client\client.py
Starting client job...
Connected to localhost:8000
Encrypted message sent: b'\x002|W\x7f%\x8d\xf4\xd2\x0c\xcd~\xcf\x99\xeb\xc4y\xe9\xcfZ5w3o\xbf\x91}k\x8eIp\x9
b\x8e\xa4\xdf\x16\x98>}!\x8b/\x08\x0b\xacd\xb8P\xbc\xa0k\xa2\xa4|\xe2\xc3\xb2\xab\xa2\x95\xc6\xe5u\x96\xbd\x
9f\xc0\t$\x15E\x7f+\t\x10\xcc\x06\xc3\x90\x15\x19n\xc6\xab*"'
OK
b'SS\x02\x9a\xd4\xe78\xecX\xeb|\xd8O\x06\x1f\xd5F\xd1\xb6%\\.m)\\\xacm&\x82\xf2\xb9\xc1\xca=m\x86u\x03\x10\x
cc[Lo'
Encrypted EndSession message sent.
```

   i jest widoczna w Wiresharku:



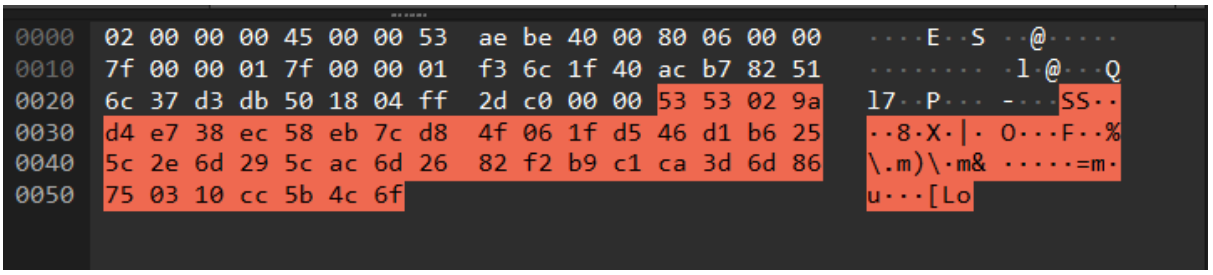
```
7e  ae ba 40 00 80 06 00 00    ····E··~ ··@·····
01  f3 6c 1f 40 ac b7 81 fb    ········ ·l·@····
ff  d7 f4 00 00 00 32 7c 57    l7··P··· ·····2|W
7e  cf 99 eb c4 79 e9 cf 5a    ·%·····~ ····y··Z
6b  8e 49 70 9b 8e a4 df 16    5w3o··}k ·Ip·····
0b  ac 64 b8 50 bc a0 6b a2    ·>}!·/·· ·d·P··k·
95  c6 e5 75 96 bd 9f c0 09    ·|····· ··u·····
cc  06 c3 90 15 19 6e c6 ab    $·E·+··· ·····n··
                               *"
```

Protocol	Length	Info
TCP	56	62316 → 8000 [SYN] Seq=0 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM
TCP	56	8000 → 62316 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM
TCP	44	62316 → 8000 [ACK] Seq=1 Ack=1 Win=327424 Len=0
TCP	62	62316 → 8000 [PSH, ACK] Seq=1 Ack=1 Win=327424 Len=18
TCP	44	8000 → 62316 [ACK] Seq=1 Ack=19 Win=2161152 Len=0
TCP	58	8000 → 62316 [PSH, ACK] Seq=1 Ack=19 Win=2161152 Len=14
TCP	44	62316 → 8000 [ACK] Seq=19 Ack=15 Win=327424 Len=0
TCP	130	62316 → 8000 [PSH, ACK] Seq=19 Ack=15 Win=327424 Len=86
TCP	44	8000 → 62316 [ACK] Seq=15 Ack=105 Win=2161152 Len=0
TCP	46	8000 → 62316 [PSH, ACK] Seq=15 Ack=105 Win=2161152 Len=2
TCP	44	62316 → 8000 [ACK] Seq=105 Ack=17 Win=327424 Len=0
TCP	87	62316 → 8000 [PSH, ACK] Seq=105 Ack=17 Win=327424 Len=43
TCP	44	8000 → 62316 [ACK] Seq=17 Ack=148 Win=2161152 Len=0

4. Komunikat zakończenia sesji wysyłany przez klienta
   Widzimy, że ma on taką postać:

```
PS C:\Users\angel\Documents\MOJE\STUDIA\SEM5\PSI\laby> python3 .\projekt\client\client.py
Starting client job...
Connected to localhost:8000
Encrypted message sent: b'\x002|W\x7f%\x8d\xf4\xd2\x0c\xcd~\xcf\x99\xeb\xc4y\xe9\xcfZ5w3o\xbf\x91}k\x8eIp\x9
b\x8e\xa4\xdf\x16\x98>}!\x8b/\x08\x0b\xacd\xb8P\xbc\xa0k\xa2\xa4|\xe2\xc3\xb2\xab\xa2\x95\xc6\xe5u\x96\xbd\x
9f\xc0\t$\x15E\x7f+\t\x10\xcc\x06\xc3\x90\x15\x19n\xc6\xab*"'
OK
b'SS\x02\x9a\xd4\xe78\xecX\xeb|\xd8O\x06\x1f\xd5F\xd1\xb6%\\.m)\\\xacm&\x82\xf2\xb9\xc1\xca=m\x86u\x03\x10\x
cc[Lo'
Encrypted EndSession message sent.
```

i jest widoczny w Wiresharku:



```
0000  02 00 00 00 45 00 00 53  ae be 40 00 80 06 00 00   ····E··S ··@·····
0010  7f 00 00 01 7f 00 00 01  f3 6c 1f 40 ac b7 82 51   ········ ·l·@···Q
0020  6c 37 d3 db 50 18 04 ff  2d c0 00 00 53 53 02 9a   l7··P··· -···SS··
0030  d4 e7 38 ec 58 eb 7c d8  4f 06 1f d5 46 d1 b6 25   ··8·X·|· O···F··%
0040  5c 2e 6d 29 5c ac 6d 26  82 f2 b9 c1 ca 3d 6d 86   \.m)\·m& ·····=m·
0050  75 03 10 cc 5b 4c 6f                              u···[Lo
```

Protocol	Length	Info
TCP	56	62316 → 8000 [SYN] Seq=0 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM
TCP	56	8000 → 62316 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM
TCP	44	62316 → 8000 [ACK] Seq=1 Ack=1 Win=327424 Len=0
TCP	62	62316 → 8000 [PSH, ACK] Seq=1 Ack=1 Win=327424 Len=18
TCP	44	8000 → 62316 [ACK] Seq=1 Ack=19 Win=2161152 Len=0
TCP	58	8000 → 62316 [PSH, ACK] Seq=1 Ack=19 Win=2161152 Len=14
TCP	44	62316 → 8000 [ACK] Seq=19 Ack=15 Win=327424 Len=0
TCP	130	62316 → 8000 [PSH, ACK] Seq=19 Ack=15 Win=327424 Len=86
TCP	44	8000 → 62316 [ACK] Seq=15 Ack=105 Win=2161152 Len=0
TCP	46	8000 → 62316 [PSH, ACK] Seq=15 Ack=105 Win=2161152 Len=2
TCP	44	62316 → 8000 [ACK] Seq=105 Ack=17 Win=327424 Len=0
TCP	87	62316 → 8000 [PSH, ACK] Seq=105 Ack=17 Win=327424 Len=43
TCP	44	8000 → 62316 [ACK] Seq=17 Ack=148 Win=2161152 Len=0

Powyższe zrzuty ekranu z Wiresharka udowadniają poprawne działanie programu.

Poprawna kolejność - najpierw dzieje się ClientHello, następnie ServerHello, a potem klient wysyła wiadomość.

Poprawna forma - teksty “ClientHello” i “ServerHello” są wysyłane jawnym tekstem, a właściwa wiadomość jak i “EndSession” są zaszyfrowane. Aby znaleźć je w ruchu pakietów musieliśmy wcześniej znać ich wartości, co dla celów testów wypisywaliśmy w konsoli.

Napotkane problemy

Napotkaliśmy problem związany z wybranymi separatorami zawartymi w wiadomości - “|” które m.in. oddzielają właściwą zaszyfrowaną treść wiadomości od informacji o jej długości i wartości MAC. Początkowo chcieliśmy dzielić otrzymaną wiadomość po separatorach.
Okazało się jednak, że pionowe kreski mogą też występować w zaszyfrowanej wiadomości. Z tego powodu w zrezygnowaliśmy z dzielenia po separatorach i bazujemy na długości poszczególnych fragmentów.

Napotkaliśmy też problem z odróżnieniem zwykłej wysyłanej przez klienta wiadomości od komunikatu zakończenia sesji. Zgodnie z naszymi założeniami zwykła wiadomość zaczyna się od 2 bajtów z rozmiarem wiadomości, a komunikat zakończenia sesji od zaszyfrowanego tekstu “EndSession” o długości 10 znaków. Musieliśmy odróżniać te 2 przypadki i inaczej na nie reagować.
Próbowaliśmy sprawdzać typ pierwszych dwóch bajtów wiadomości. W założeniu, jeśli byłby to int to mielibyśmy do czynienia ze zwykłą wiadomością i jej rozmiarem. Korzystaliśmy do tego z funkcji pythona int.from_bytes(). Okazało się jednak, że dowolne inne znaki niebędące liczbami po użyciu na nich funkcji int.from_bytes() zamieniały się w liczbę. Tak więc sprawdzenie czy pierwsze 2 bajty są liczbą zawsze okazywało się prawdą i nie rozwiązywało naszego problemu.
Z tego powodu zastosowaliśmy inne podejście - odszyfrowywanie pierwszych 10 bajtów wiadomości i sprawdzanie, czy brzmią one “EndSession”. Jeśli tak, mamy do czynienia z komunikatem żądającym zakończenia połączenia, w przeciwnym wypadku ze zwykłą wiadomością.

W kwestii równoczesnego obsługiwania wielu klientów używamy zmiennej reprezentującej maksymalną ilość klientów połączonych naraz z serwerem. Gdy maksymalna ilość połączeń jest przekroczona, klient oczekuje na możliwość połączenia się i łączy się kiedy miejsce się zwolni.

Wnioski

Podsumowując, udało nam się zrealizować komunikację serwer-klienci, różne typy komunikatów są prawidłowo rozróżniane, a wiadomości są prawidłowo szyfrowane. Mimo że użytkownik nie może zdefiniować treści wiadomości (zgodnie z wymaganiami), to wiemy, że

serwer działa także dla dużych wiadomości, przekraczających rozmiar bufora - wczytywana jest ona poprawnie w częściach.
Programy można uruchomić w kilku poleceniach, a po ich uruchomieniu obsługa jest bardzo prosta.
Rozwiązaniu nie brakuje luk w zabezpieczeniach - przechwytujący wiadomość mógłby zmienić jej pierwszą część zawierającą informacje o jej długości i nie zostałoby to wykryte przy sprawdzaniu kodu MAC. Dla każdego klienta moglibyśmy generować losowo moduł, bazę oraz klucz prywatny. Stałe wartości przekazują jednak idee stojącą za ich ustalaniem.