Programowanie Sieciowe
Projekt
Sprawozdanie końcowe

Radosława Żukowska - Lider Zespołu
Aleksandra Szczypawka
Małgorzata Grzanka

18.01.2026 r.
Wersja sprawozdania: 1

Opis rozwiązania

Link do Repozytorium

Serwer

Serwer jest zaimplementowany jako wielowątkowa aplikacja TCP zdolna do obsługi wielu klientów jednocześnie - każdy klient jest obsługiwany w osobnym wątku. Implementacja znajduje się w pliku `Server/tcp_server.py`. Główna klasa `Server` zawiera pętlę akceptującą nowe połączenia TCP na porcie 1234 oraz osobny wątek obsługujący komendy administratora. Każdy klient jest obsługiwany przez osobny wątek.

- Komenda `LIST` wyświetla listę wszystkich połączonych klientów (ID, adres IP).
- Komenda `DELETE <ID>` kończy połączenie dla wskazanego klienta - wysyła wiadomość `END` i usuwa klienta.

Obsługa klientów

Każdy klient otrzymuje unikalny ID przyłączany do połączenia, a sesja TLS jest tworzona dla każdego klienta osobno. Maksymalna liczba klientów jest konfigurowalna przez parametr wiersza poleceń (domyślnie 10).

Obsługa protokołu

- Odbiera `ClientHello` (niezaszyfrowane) - odczytuje parametry algorytmu Diffie-Hellman'a (P, G) i klucz publiczny klienta.
- Wysyła `ServerHello` (niezaszyfrowane) - wysyła swój klucz publiczny DH.
- Po ustaleniu TLS: odbiera i wysyła tylko zaszyfrowane wiadomości (MSG, END).
- Wszystkie wiadomości po ustaleniu TLS są szyfrowane przez `encrypt_and_mac()` i weryfikowane przez `verify_and_decrypt()` z klasy `Session`.

Klient

Klient jest zaimplementowany jako aplikacja TCP z interfejsem wiersza poleceń w pliku `Client/tcp_client.py`, zdolna do nawiązywania połączenia TLS z serwerem. Klient wykorzystuje wielowątkowość - osobny wątek odbiera wiadomości od serwera i umieszcza je w kolejce, podczas gdy główny wątek obsługuje komendy użytkownika.

- Komenda `CONNECT` - nawiązuje połączenie TCP z serwerem, wysyła `ClientHello`, odbiera `ServerHello` i ustala sesję TLS.
- Komenda `MSG` - wysyła zaszyfrowaną wiadomość do serwera (treść: "Hello").
- Komenda `END` - wysyła wiadomość `EndSession` i resetuje połączenie. Aby ponownie połączyć się z serwerem, należy jeszcze raz użyć komendy `CONNECT`.
- Komenda `QUIT` - kończy działanie klienta.

Obsługa protokołu

- Aby nawiązać połączenie TLS z serwerem, klient ustawia parametry DH (P, G) używając domyślnych wartości i generuje swój klucz prywatny oraz publiczny.
- Następnie wysyła `ClientHello` z parametrami (P, G) i kluczem publicznym.
- Po otrzymaniu `ServerHello` od serwera, obliczany jest współdzielony klucz sesyjny używając algorytmu Diffie-Hellman'a.
- Po ustaleniu sesji TLS, wszystkie wiadomości (MSG, END) są szyfrowane przez `encrypt_and_mac()`, a odebrane wiadomości są weryfikowane i deszyfrowane przez `verify_and_decrypt()` z klasy `Session`.

Opis algorytmów

Wymiana kluczy Diffiego-Helmann

Algorytm Diffiego-Hellmana został zaimplementowany w klasie `Session` w module `common/session.py`.

Parametry algorytmu

Klient proponuje parametry P (modulus - liczba pierwsza) i G (generator) w wiadomości `ClientHello`. Serwer odczytuje parametry P i G z `ClientHello` i używa ich do obliczenia własnego klucza. Domyślne wartości to: `DEFAULT_DH_P = 4294967291` (liczba pierwsza), `DEFAULT_DH_G = 5` (generator) - zdefiniowane w klasie `Session`.

Proces wymiany kluczy

1. **Klient**:
    - Ustawia parametry P i G (domyślne `DEFAULT_DH_P`, `DEFAULT_DH_G`).
    - Generuje losowy klucz prywatny: `private_c = random(2, P-1)`.
    - Oblicza klucz publiczny: `public_c = G^private_c % P`.

- Wysyła `ClientHello`: `[P (4B)][G (4B)][public_c (4B)]`.

2. **Serwer**:
   - Odczytuje P i G z `ClientHello`.
   - Generuje losowy klucz prywatny: `private_s = random(2, P-1)`.
   - Oblicza klucz publiczny: `public_s = G^private_s % P`.
   - Wysyła `ServerHello`: `[public_s (4B)]`.
3. **Obliczenie współdzielonego klucza**:
   - Klient: `shared_key = public_s^private_c % P`.
   - Serwer: `shared_key = public_c^private_s % P`.
   - Oba obliczenia dają identyczny wynik dzięki właściwości algorytmu DH.
4. **Obliczanie klucza MAC**:
   - Współdzielony klucz jest konwertowany na 4 bajty (big-endian).
   - Klucz MAC jest obliczany jako: `mac_key = SHA256(shared_key)[:4]`.

Przykład obliczeń

Używamy parametrów domyślnych: P = 4294967291, G = 5

- Klient: `private_c = 1234567`, `public_c = 5^1234567 mod 4294967291`
- Serwer: `private_s = 9876543`, `public_s = 5^9876543 mod 4294967291`
- Współdzielony klucz: `shared_key = public_s^1234567 mod P = public_c^9876543 mod P`

Szyfrowanie wiadomości

Do szyfrowania wiadomości jest użyty prosty algorytm OTP zaimplementowany w metodzie `encrypt_message()` klasy `Session`. Polega on na wykonaniu operacji XOR pomiędzy szyfrowaną wiadomością a kluczem współdzielonym. Operacja XOR jest symetryczna: `decrypt(ciphertext) = encrypt(ciphertext)`, zatem odszyfrowanie wiadomości polega na ponownym wykonaniu tej samej operacji na zaszyfrowanej wiadomości.

Każdy bajt wiadomości jest XORowany z odpowiednim bajtem klucza. Klucz (4 bajty) jest cyklicznie powtarzany dla dłuższych wiadomości (operator modulo `i % key_len`).

```
def encrypt_message(self, plaintext: bytes) -> bytes:
    if self.shared_key is None:
        raise ValueError("Shared key not established")

    key = self.shared_key
    key_len = len(key)
    ciphertext = bytes([
        b ^ key[i % key_len]
        for i, b in enumerate(plaintext)
    ])
    return ciphertext
```

Mechanizm encrypt-then-mac

Mechanizm `encrypt-then-MAC` został zaimplementowany w metodach `encrypt_and_mac()` i `verify_and_decrypt()` klasy `Session`.

Proces wysyłania wiadomości (`encrypt_and_mac()`)

- Na początku, wiadomość do wysłania jest szyfrowana za pomocą opisanego w poprzedniej sekcji algorytmu szyfrowania, z użyciem wynegocjowanego klucza: `ciphertext = encrypt_message(plaintext)`.
- Następnie generowany jest kod MAC - są to pierwsze 4 bajty z wyrażenia `SHA256(mac_key + ciphertext)`. Klucz MAC to pierwsze 4B z funkcji skrótu SHA256 na współdzielony klucz: `SHA256(shared_key)[:4]`.
- Wysyłany jest pakiet składający się z `ciphertext + MAC`.

Proces odbierania wiadomości (`verify_and_decrypt()`)

- Na początku, wyodrębniamy kod MAC i zaszyfrowaną wiadomość z otrzymanego pakietu - `recv_mac = data[-4:]` (ostatnie 4 bajty), `ciphertext = data[:-4]` (reszta).
- Przed odszyfrowaniem wiadomości, należy sprawdzić poprawność kodu MAC. Wylicza się oczekiwany MAC - `expected_mac = SHA256(mac_key + ciphertext)[:4]` i porównuje z wcześniej wyodrębnionym.
- Jeśli kody nie zgadzają się, wiadomość nie jest odszyfrowywana i zgłaszany jest wyjątek `ValueError("MAC verification failed")`. Jeśli się zgadzają, wiadomość jest deszyfrowana za pomocą wyrażenia `plaintext = decrypt_message(ciphertext)`.

Działanie protokołu

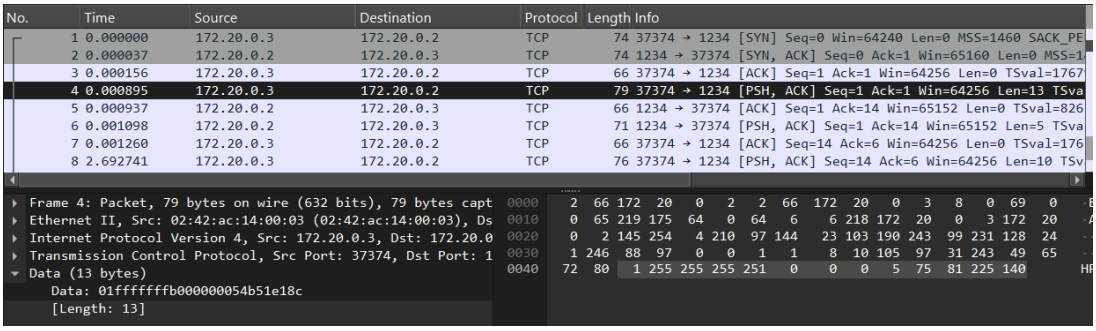
Kolejność przesyłania pakietów

Zweryfikowałyśmy poprawność kolejności przesyłania pakietów w zaimplementowanym przez nas protokołu mini-TLS. W ramach testu został uruchomiony serwer a następnie klient wykonał następujące akcje: CONNECT, MSG, END. Serwer znajdował się na porcie 1234, klient 37374. Interesujące nas pakiety są te z flagą PSH protokołu TCP. Jest 4, czyli tyle, ile się spodziewamy: 3 od klienta, 1 od serwera.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	172.20.0.3	172.20.0.2	TCP	74	37374 → 1234 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM TSval=1767972850 TSecr=0 WS=128
2	0.000037	172.20.0.2	172.20.0.3	TCP	74	1234 → 37374 [SYN, ACK] Seq=0 Ack=1 Win=65160 Len=0 MSS=1460 SACK_PERM TSval=826361936 TSecr=1767972850 WS=128
3	0.000156	172.20.0.3	172.20.0.2	TCP	66	37374 → 1234 [ACK] Seq=1 Ack=1 Win=64256 Len=0 TSval=1767972850 TSecr=826361936
4	0.000895	172.20.0.3	172.20.0.2	TCP	79	37374 → 1234 [PSH, ACK] Seq=1 Ack=1 Win=64256 Len=13 TSval=1767972851 TSecr=826361936
5	0.000937	172.20.0.2	172.20.0.3	TCP	66	1234 → 37374 [ACK] Seq=1 Ack=14 Win=65152 Len=0 TSval=826361937 TSecr=1767972851
6	0.001098	172.20.0.2	172.20.0.3	TCP	71	1234 → 37374 [PSH, ACK] Seq=1 Ack=14 Win=65152 Len=5 TSval=826361937 TSecr=1767972851
7	0.001260	172.20.0.3	172.20.0.2	TCP	66	37374 → 1234 [ACK] Seq=14 Ack=6 Win=64256 Len=0 TSval=1767972851 TSecr=826361937
8	2.692741	172.20.0.3	172.20.0.2	TCP	76	37374 → 1234 [PSH, ACK] Seq=14 Ack=6 Win=64256 Len=10 TSval=1767975542 TSecr=826361937
9	2.735913	172.20.0.2	172.20.0.3	TCP	66	1234 → 37374 [ACK] Seq=6 Ack=24 Win=65152 Len=0 TSval=826364672 TSecr=1767975542
10	4.145788	172.20.0.3	172.20.0.2	TCP	71	37374 → 1234 [PSH, ACK] Seq=24 Ack=6 Win=64256 Len=5 TSval=1767976995 TSecr=826364672
11	4.145842	172.20.0.2	172.20.0.3	TCP	66	1234 → 37374 [ACK] Seq=6 Ack=29 Win=65152 Len=0 TSval=826366082 TSecr=1767976995
12	4.146224	172.20.0.2	172.20.0.3	TCP	66	1234 → 37374 [FIN, ACK] Seq=6 Ack=29 Win=65152 Len=0 TSval=826366082 TSecr=1767976995
13	4.146453	172.20.0.3	172.20.0.2	TCP	66	37374 → 1234 [FIN, ACK] Seq=29 Ack=7 Win=64256 Len=0 TSval=1767976996 TSecr=826366082
14	4.146472	172.20.0.2	172.20.0.3	TCP	66	1234 → 37374 [ACK] Seq=7 Ack=30 Win=65152 Len=0 TSval=826366082 TSecr=1767976996

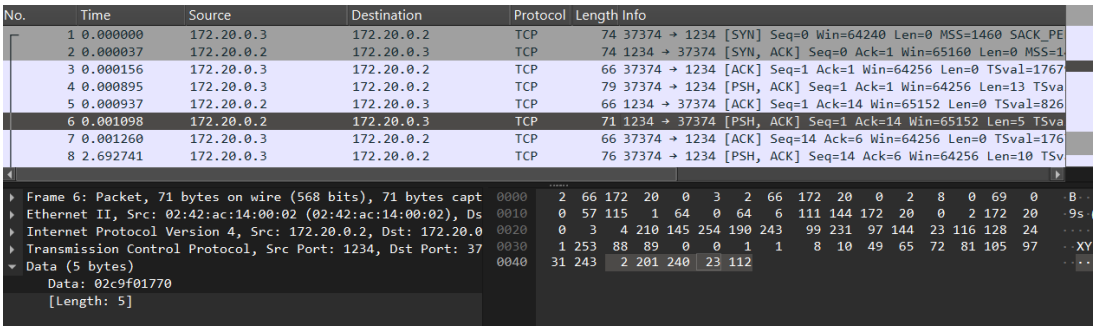
Rysunek 1: Ruch sieciowy podczas komunikacji

Rysunek 2 pokazuje zawartość pierwszego pakietu od klienta, nie jest on zaszyfrowany dlatego możemy odczytać, że jest to poprawny pakiet ClientHello, pierwszy bajt ma wartość 1, a w protokole jest używany do oznaczenia dokładnie tego typu wiadomości.



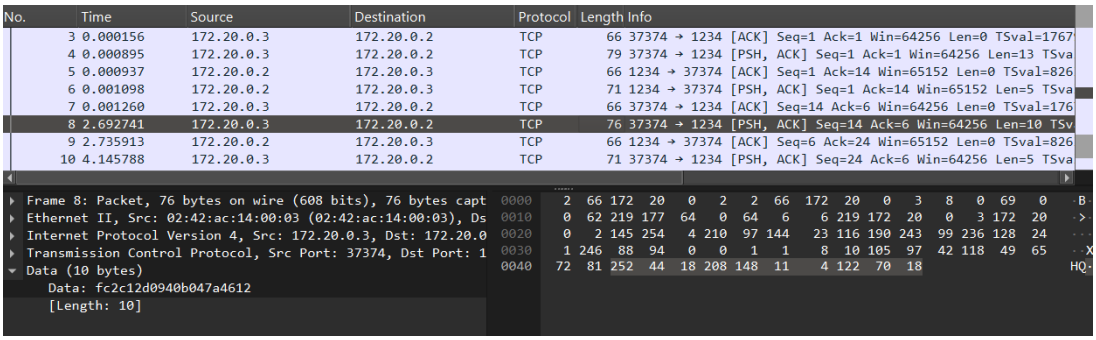
Rysunek 2: Wiadomość ClientHello

Rysunek 3 przedstawia wiadomość od serwera do klienta przesłaną w odpowiedzi na poprzednią, także nie jest zaszyfrowana, a pierwszy bajt ma wartość 2, taką jak oczekiwana wartość dla flagi wiadomości ServerHello.



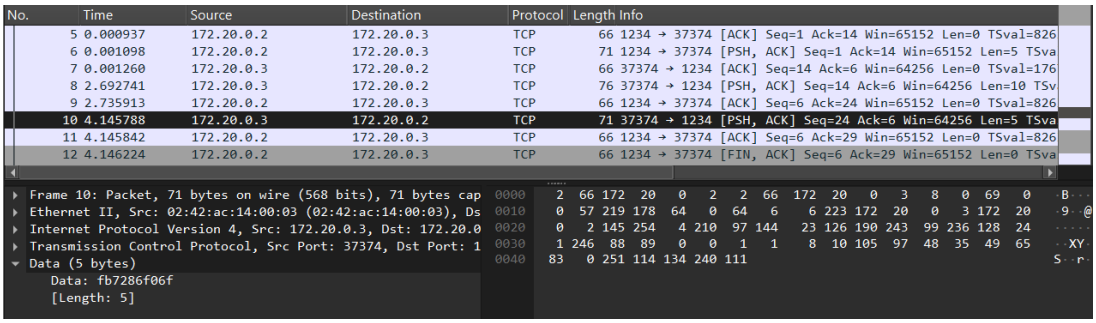
Rysunek 3: Wiadomość ServerHello

Rysunek 4 przedstawia kolejną wiadomość wysłaną od klienta do serwera. Pierwszy bajt nie ma wartości żadnej ze zdefiniowanych flag (1-4), jednak zgodnie z założeniami ta wiadomość powinna być już szyfrowana włącznie z informacją o jej typie, czyli spełnia je. W kolejnej sekcji pokazane jest odszyfrowywanie tej wiadomości.



Rysunek 4: Wiadomość zwyczajna od klient do serwera po nawiązaniu komunikacji

Także na Rysunku 5 na pierwszym bicie danych nie ma liczby od 1-4. Jest to pakiet, który został wysłany przez klienta poleceniem END, także jest szyfrowany. Odszyfrowanie jego także zostanie pokazane w następnej sekcji.



Rysunek 5: Wiadomość kończąca połączenie End

Poprawność szyfrowania pakietów

W ramach testu klucze oraz wartości MAC zostały zapisane do pliku przez serwer, po odebraniu ClientHello i wykonaniu potrzebnych obliczeń. Następnie te pliki zostały skopiowane do folderu `tests` w głównym katalogu programu. Wartości przesyłanych wiadomości zostały pobrane z logów programu `tcpdump` przy analizowaniu ich programem Wireshark. Następnie za pomocą programu `tests/test.py` uzyskane pliki zostały użyte do odszyfrowania wiadomości.

```
Zaszyfrowana wiadomość z MAC:
252 44 18 208 148 11 4 122 70 18
Zaszyfrowana wiadomość bez MAC:
252 44 18 208 148 11
Odszyfrowana wiadomość:
b'\x04Hello'
```

Rysunek 6: Zaszyfrowane i odszyfrowane dane wiadomości z Rysunku 4

```
Zaszyfrowana wiadomość z MAC:
251 114 134 240 111
Zaszyfrowana wiadomość bez MAC:
251
Odszyfrowana wiadomość:
b'\x03'
```

Rysunek 7: Zaszyfrowane i odszyfrowane dane wiadomości z Rysunku 5

Rysunki 6 i 7 przedstawiają odpowiednio analizę przesłanych danych z rysunków 4 i 5. Tak jak widać zaszyfrowana wiadomość nie zawiera na pierwszym miejscu wartości flagi. Dopiero odszyfrowanie wiadomości pozwala na poznanie rodzaju wiadomości, gdzie 4 to kod wiadomości regularnej w trakcie połączenia, a 3 to kod wiadomości zamykającej połączenie, co zgadza się z oczekiwaniami.

Napotkane problemy

- **Parametry P i G do algorytmu Diffie-Hellman'a** - Pierwotnie parametry P i G były zahardkodowane w klasie `Session`, co nie spełniało wymagań projektu. Ostatecznie, zaimplementowałyśmy mechanizm, w którym klient proponuje parametry P i G w `ClientHello`, serwer je akceptuje i używa do wymiany kluczy.

- **Monitorowanie pakietów przesyłanych między kontenerami dockerowymi** - Przechwytywanie pakietów w sieci Docker wymaga specjalnej konfiguracji Wireshark lub użycia narzędzi takich jak `tcpdump` wewnątrz kontenerów, dlatego że komunikacja pomiędzy kontenerami nie odbywa się na regularnych interfejsach sieciowych i takie wiadomości są niewidoczne dla programu Wireshark.
- **Limit klientów** - Pierwotnie limit klientów był zahardkodowany jako stała. Jako rozwiązanie, dodałyśmy parametr `max_clients` do konstruktora `Server`, możliwy do ustawienia przez argument wiersza poleceń.

Wnioski

Wszystkie wymagania projektu zostały spełnione. Nasz protokół mini-TLS zapewnia:

- bezpieczną wymianę kluczy poprzez algorytm Diffie-Hellman'a z bardzo prostą negocjacją parametrów,
- szyfrowanie wiadomości za pomocą wynegocjowanego klucza, algorytmem XOR cipher,
- mechanizm integralności encrypt-then-MAC (implementacja w metodach `encrypt_and_mac()` i `verify_and_decrypt()` klasy `Session`),
- obsługę wielu klientów jednocześnie,
- komunikację między serwerem, a klientem - otwieranie sesji, negocjacja parametrów, przesyłanie wiadomości oraz zamykanie sesji.
- panel sterowania klientem i serwerem w celu testowania rozwiązania.