Sprawozdanie Wstępne

Projekt: Mini-TLS (Wariant W1)

Zespół nr 39
Skład zespołu:
Łukasz Szydlik (lider)
Jan Szwagierczak
Dominik Śledziewski

1 Wstęp

Celem projektu jest implementacja bezpiecznego protokołu komunikacyjnego (mini TLS) w architekturze klient-serwer. Protokół zapewnia poufność oraz integralność przesyłanych danych przy użyciu kryptografii symetrycznej i asymetrycznej. Projekt realizowany jest w języku Python, a komunikacja odbywa się w odizolowanym środowisku Docker.

Wybrany wariant realizacyjny to **W1**: Wykorzystanie mechanizmu *Encrypt-then-MAC* dla zapewnienia integralności i autentyczności wiadomości.

2 Struktura Wiadomości

Komunikacja opiera się na wymianie obiektów w formacie JSON, przesyłanych przez gniazda TCP. Poniżej przedstawiono strukturę poszczególnych typów wiadomości.

2.1 Faza Handshake (Nieszyfrowana)

Wiadomości te służą do wymiany kluczy publicznych oraz ustalenia parametrów kryptograficznych niezbędnych do wygenerowania wspólnego sekretu. Są przesyłane jawnym tekstem.

1. ClientHello – inicjacja połączenia przez klienta. Wraz z kluczem publicznym przesyłane są parametry Diffie-Hellmana (P i G), co pozwala na ich ustalanie i unika hardcodowania po stronie serwera.

```
{
    "type": "ClientHello",
    "public_key": 123456789, // Klucz publiczny klienta (g^a mod p)
    "p": 2147483647,         // Liczba pierwsza P
    "g": 16807               // Generator G
}
```

2. ServerHello – odpowiedź serwera.

```
{
    "type": "ServerHello",
    "public_key": 987654321  // Klucz publiczny serwera (g^b mod p)
}
```

2.2 Faza Transferu Danych (Szyfrowana)

Po ustanowieniu sesji, wszystkie kolejne wiadomości przesyłane są w bezpiecznym kontenerze. Protokół rozróżnia warstwę transportową (to, co widać w sieci) oraz warstwę aplikacji (to, co jest szyfrowane).

Struktura transportowa (Encrypt-then-MAC): Każda zaszyfrowana wiadomość przesyłana przez sieć ma następującą postać:

```
{
    "ciphertext": "a1b2c3d4...",  // Zaszyfrowana tresc (hex)
    "mac": "e5f6..."              // HMAC obliczony z ciphertextu (hex)
}
```

Struktura warstwy aplikacji (Tekst jawny przed szyfrowaniem): Po poprawnym zweryfikowaniu MAC i odszyfrowaniu pola `ciphertext`, uzyskiwany jest JSON o jednej z dwóch postaci:

A. Zwykła wiadomość tekstowa:

```
{
    "type": "Message",
    "content": "Tajnia wiadomosc od klienta"
}
```

B. Komunikat zakończenia sesji:

```
{
    "type": "EndSession"
}
```

3 Wykorzystane Algorytmy i Scenariusz Użycia

3.1 Algorytmy kryptograficzne

W projekcie zastosowano następujące prymitywy kryptograficzne:

1. **Wymiana kluczy: Diffie-Hellman (DH)** Parametry grupowe (P i G) nie są zahardcodowane w kodzie serwera; są one proponowane przez klienta w fazie inicjalizacji połączenia. Przykładowe wartości to:

$$P = 2^{31} - 1 = 2147483647 \quad \text{(Liczba pierwsza Mersenne'a)}$$

$$G = 16807 \quad \text{(Generator)}$$

   Wspólny sekret S obliczany jest jako $S = (g^a)^b \pmod P$.

2. **Derywacja kluczy (KDF)** Ze wspólnego sekretu S wyprowadzane są dwa niezależne klucze przy użyciu funkcji skrótu SHA-256:

   - Klucz szyfrowania (K_{enc}) = SHA256(S || "ENC")
   - Klucz integralności (K_{mac}) = SHA256(S || "MAC")

3. **Szyfrowanie: XOR Cipher (Symulacja OTP)** Zaimplementowano szyfr strumieniowy. Generator liczb pseudolosowych (PRNG) jest inicjowany kluczem K_{enc}. Wygenerowany strumień klucza jest operowany funkcją XOR z bajtami wiadomości.

4. **Integralność: HMAC-SHA256** Kod uwierzytelniania wiadomości obliczany jest z wykorzystaniem funkcji skrótu SHA-256 i klucza K_{mac}.

3.2 Scenariusz połączenia

Poniżej przedstawiono przebieg przykładowej sesji:

1. **Inicjalizacja:** Klient wybiera parametry P i G, a następnie generuje parę kluczy DH (prywatny a, publiczny $A = G^a \pmod P$).
2. **ClientHello:** Klient wysyła JSON zawierający klucz publiczny oraz parametry grupy: `{"type": "ClientHello", "public_key": A, "p": P, "g": G}`.
3. **Odpowiedź serwera:** Serwer otrzymuje A, P, G. Na podstawie otrzymanych parametrów generuje swoją parę kluczy (prywatny b, publiczny B) i oblicza sekret S.
4. **ServerHello:** Serwer wysyła JSON: `{"type": "ServerHello", "public_key": B}`.
5. **Derywacja:** Klient otrzymuje B, oblicza sekret S. Obie strony generują niezależnie K_{enc} i K_{mac}. Stan sesji zmienia się na *Authenticated*.
6. **Komunikacja:** Klient chce wysłać tekst "Hello".
   - Tworzony jest JSON: `{"type": "Message", "content": "Hello"}`.
   - Treść jest szyfrowana $K_{enc} \rightarrow$ powstaje `ciphertext`.
   - Obliczany jest HMAC(K_{mac}, `ciphertext`).
   - Do serwera wysyłany jest JSON z `ciphertext` i `mac`.
7. **Zakończenie:** Klient wysyła zaszyfrowany komunikat `EndSession`. Serwer zamyka połączenie i usuwa klucze sesji. Połączenie TCP zostaje zakończone - ponowna komunikacja wymaga nawiązania nowego połączenia i przeprowadzenia pełnego handshake'u.

4 Realizacja Integralności i Autentyczności

Zgodnie z wariantem **W1**, zastosowano podejście **Encrypt-then-MAC**. Jest to metoda uważana za bezpieczniejszą niż *MAC-then-Encrypt*, ponieważ weryfikacja integralności następuje przed próbą odszyfrowania danych, co chroni przed atakami typu *padding oracle* lub błędami implementacji procedury deszyfrowania.

Procedura nadawania:

$$C = E_{K_{enc}}(M)$$

$$T = \text{HMAC}_{K_{mac}}(C)$$

$$\text{Wyślij } (C, T)$$

Procedura odbioru: Odbiorca otrzymuje parę (C', T').

1. Oblicza oczekiwany tag: $T_{expected} = \text{HMAC}_{K_{mac}}(C')$.
2. Porównuje $T_{expected}$ z T' przy użyciu funkcji `hmac.compare_digest` (odpornej na ataki czasowe).
3. Jeśli tagi są różne $\rightarrow$ Rzuca wyjątek `Integrity check failed` i przerywa przetwarzanie (nie deszyfruje).
4. Jeśli tagi są zgodne $\rightarrow$ Odszyfrowuje treść: $M' = D_{K_{enc}}(C')$.

Dzięki temu rozwiązaniu, potencjalny atakujący, który zmodyfikuje zaszyfrowaną wiadomość w locie (np. zmieniając jeden bit w polu `ciphertext`), spowoduje, że serwer odrzuci pakiet jeszcze przed próbą użycia funkcji XOR, co zapewnia pełną integralność komunikacji.