1 Implementacja protokołu komunikacji

Protokół komunikacji między dwoma węzłami zaprojektowano i zaimplementowano z następującymi założeniami:

- pełna funkcjonalność przy jak najmniejszych wymaganiach sprzętowych,
- niezależność od warstwy sieciowej,
- zapewnienie uwierzytelniania i szyfrowania wiadomości.

1.1. Podstawowe struktury protokołu

W protokole wymieniane są rekordy, których zawartość poprzedzona jest nagłówkiem o następującej budowie:

- piersze dwa bajty definiują wersję protokołu i mają wartość 0x00 oraz 0x01,
- trzeci bajt definiuje typ rekordu,
- następne dwa bajty definiują długość zawartości rekordu (najbardziej znaczący bajt jako pierwszy).

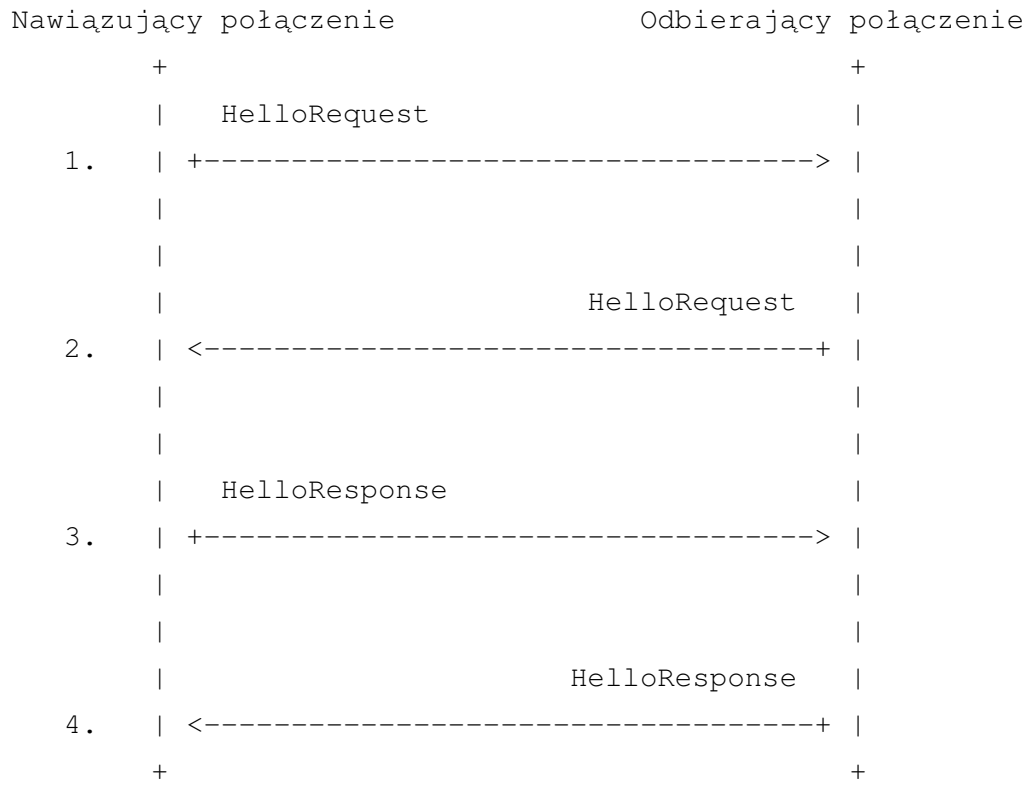
Zdefiniowane typy rekordów są przedstawione w tabeli 1.1.
Budowę przykładowych rekordów przedstawiono na rysunkach A.1, A.2 oraz A.3 w dodatku A.
Odbiorca rekordu powinien zweryfikować:

- zgodność wersji protokołu – wymagane bajty 0x00 oraz 0x01,
- prawidłowość bajtu określającego typ rekordu – wymagana wartość 0x00, 0x01 lub 0x02,
- zgodność zadeklarowanej długości zawartości rekordu z typem rekordu,
- w przypadku typów HelloResponse oraz EncryptedData – prawidłowość kodu uwierzytelniającego.

W przypadku niezgodności któregokolwiek elementu rekord powinien zostać zignorowany.

Nazwa rekordu	Wartość pola typ	Długość zawartości	Czy zawartość jest zaszyfrowana	Czy zawartość jest uwierzytelniona
HelloRequest	0x00	64 bajty	nie	nie
HelloResponse	0x01	96 bajtów	tak	tak
EncryptedData	0x02	zmienna, minimum 32 bajty	tak	tak

Tabela 1.1. Typy rekordów wraz z ich charakterystyką



Rys. 1.1. Kolejność wymiany rekordów w procesie nawiązywania połączenia

1.2. Nawiązywanie połączenia

Kolejność przesyłania rekordów w celu nawiązania połączenia przedstawiona jest na rysunku 1.1. Rekordy typu HelloRequest zawierają klucz publiczny węzła, który je wysyła. Węzeł, który odbiera HelloRequest, używa swojego klucza publicznego oraz klucza publicznego z odebranego rekordu do ustalenia sekretnego klucza.

Po ustaleniu wspólnego klucza węzły mogą wysłać rekord HelloResponse, który zawiera zaszyfrowany i uwierzytelniony klucz publiczny węzła wysyłającego rekord. Jeżeli węzeł odbierający rekord skutecznie potwierdzi, że rekord jest prawidłowo uwierzytelniony, a zdeszyfrowany klucz publiczny pokrywa się z kluczem przesłanym wcześniej w rekordzie HelloRequest, połączenie uznawane jest za nawiązane. Po nawiązaniu połączenia wymieniane mogą być tylko rekordy typu EncryptedData.

Istotne jest, że protokół nie zapewnia autentyczności danego klucza publicznego. Powinno to zostać zweryfikowane niezależnie, na przykład poprzez wyświetlenie skrótu klucza użytkownikowi i poproszenie go o potwierdzenie, że na obu urządzeniach uczestniczących w komunikacji jest wyświetlony taki sam klucz.

1.3. Generowanie współdzielonego klucza

Każdy z węzłów po odebraniu rekordu typu HelloRequest używa odebranego klucza publicznego oraz swojego klucza publicznego do ustalenia wspólnego sekretu przy użyciu algorytmu Elliptic Curve Diffie-Hellman (ECDH) oraz proponowanej przez National Institute of Standards and Technology (NIST) krzywej eliptycznej P-256 [9] (w RFC 5480 nazwaną krzywą secp256r1 [10]).

Z sekretu będącego wynikiem algorytmu ECDH liczony jest skrót przy użyciu algorytmu SHA-256. Następnie jest on dzielony na dwie części po 128-bitów. Pierwsza część staje się współdzielonym kluczem używanym do szyfrowania, druga część staje się współdzielonym kluczem używanym do uwierzytelniania.

Implementacja algorytmu ECDH z krzywą eliptyczną P-256 pochodzi z biblioteki *micro-ecc*. Jest to jedyna darmowa biblioteka implementująca ECDH.

1.4. Szyfrowanie i deszyfrowanie wiadomości

Szyfrowanie wiadomości odbywa się za pomocą szyfru blokowego AES ze 128-bitowym kluczem używanym w trybie Cipher Block Chaining (CBC). Wektor inicjalizacyjny jest losowy i dołączany do zawartości przesyłanego rekordu przed szyfrogramem. Tekst jawny jest dopełniany do pełnego bloku według algorytmu zdefiniowanego w PKCS#7 [11].

Właściwe kroki potrzebne do zaszyfrowania wiadomości wypisano poniżej.

1. Dopełnienie tekstu jawnego do pełnego bloku:
   - jeżeli długość tekstu jawnego jest wielokrotnością długości bloku, do tekstu jawnego doklejone musi być 16 bajtów o wartości 16,
   - w przeciwnym wypadku, gdy wymagane jest dopełnienie N bajtów, do tekstu jawnego doklejone musi być N bajtów o wartości N.
2. Zaszyfrowanie dopełnionego tekstu jawnego w trybie CBC z losowym wektorem inicjalizacyjnym.
3. Doklejenie wektora inicjalizacyjnego przed szyfrogramem.

Kod implementujący szyfrowanie wiadomości przedstawiono w listingu B.1 w dodatku B.

Właściwe kroki potrzebne do odszyfrowania wiadomości wypisano poniżej.

1. Oddzielenie wektora inicjalizacyjnego od szyfrogramu.

2. Zdeszyfrowanie szyfrogramu w trybu CBC przy wykorzystaniu oddzielonego wektora inicjalizacyjnego.

3. Pobranie wartości ostatniego bajtu zdeszyfrowanego ciągu:

   – wartość ta nazywana jest dalej N.

4. Zweryfikowanie poprawności dopełnienia:

   – ostatnie N bajtów musi mieć wartość N,

   – jeżeli dopełnienie jest nieprawidłowe, cały rekord jest ignorowany.

5. Usunięcie ostatnich N bajtów.

Kod implementujący odszyfrowanie wiadomości przedstawiono w listingu B.2 w dodatku B.

Implementacja algorytmu AES pochodzi z biblioteki *AVR-Crypto-Lib*. Jest to najlepiej udokumentowana, darmowa biblioteka implementująca algorytm AES. Implementacja trybu CBC oraz algorytmu dopełniania zostały zrealizowane w ramach pracy.

1.5. Uwierzytelnienie wiadomości

Uwierzytelnienie wiadomości odbywa się poprzez dołączenie do wiadomości Message Authentication Code (MAC). MAC dla danej wiadomości tworzony jest za pomocą szyfru blokowego AES ze 128-bitowym kluczem używanym w trybie Encrypt-last-block Cipher Block Chaining Message Authentication Code (ECBC-MAC). Wektor inicjalizacyjny wypełniony jest zerami i nie jest przesyłany. Uwierzytelniany jest kompletny szyfrogram wraz z wektorem inicjalizacyjnym użytym do szyfrowania, a nie tekst jawny. Długość szyfrogramu wraz z wektorem inicjalizacyjnym zawsze będzie wielokrotnością długości bloku, a więc nie jest stosowane dopełnianie.

Tryb ECBC-MAC to tryb Cipher Block Chaining Message Authentication Code (CBC-MAC), którego wynik jest dodatkowo szyfrowany innym kluczem niż ten użyty do CBC-MAC. W tej pracy do CBC-MAC użyty jest klucz służący do uwierzytelniania, a wynik CBC-MAC jest szyfrowany używając klucza służącego do szyfrowania.

Właściwe kroki potrzebne do obliczenia kodu uwierzytelniającego to:

1. obliczenie ostatniego bloku będącego wynikiem zaszyfrowania szyfrogramu wraz z wektorem inicjalizacyjnym w trybie CBC z wektorem inicjalizacyjnym wypełnionym zerami przy użyciu klucza przeznaczonego do uwierzytelniania,

2. zaszyfrowanie bloku przy wykorzystaniu AES i klucza przeznaczonego do szyfrowania.

Węzeł wysyłający dokleja kod uwierzytelniający przed szyfrogramem. Węzeł odbierający oddziela otrzymany kod od szyfrogramu, oblicza kod uwierzytelniający dla danego szyfrogramu i porównuje, czy

zgadza się on z kodem otrzymanym. Jeżeli kod obliczony różni się od kodu otrzymanego, cały rekord jest ignorowany.

Kod implementujący obliczanie MAC przedstawiono w listingu B.3 w dodatku B.

Implementacja trybu ECBC-MAC została zrealizowana w ramach pracy.

Bibliografia

[1] Massimo Banzi and Michael Shiloh. *Getting Started with Arduino: The Open Source Electronics Prototyping Platform*. Sebastopol: Maker Media, Inc., 2014.

[2] Martin McKeay i in. *Q3 2016 State of the Internet Security Report*. Spraw. tech. Akamai Technologies, Inc., 2016.

[3] *ATtiny4 / ATtiny5 / ATtiny9 / ATtiny10 - Datasheet Summary*. Atmel. 2016. URL: *http://www.atmel.com/Images/Atmel-8127-AVR-8-bit-Microcontroller-ATtiny4-ATtiny5-ATtiny9-ATtiny10_Datasheet-Summary.pdf* (dostęp dnia 2016-12-06).

[4] *ATxmega384C3 - Datasheet*. Atmel. 2016. URL: *http://www.atmel.com/Images/Atmel-8361-8-and-16-bit-AVR-XMEGA-Microcontrollers-ATxmega384C3_Datasheet.pdf* (dostęp dnia 2016-12-06).

[5] *ATmega16U4/ATmega32U4 - Datasheet*. Atmel. 2016. URL: *http://www.atmel.com/Images/Atmel-7766-8-bit-AVR-ATmega16U4-32U4_Datasheet.pdf* (dostęp dnia 2016-12-06).

[6] *Recommendation for Key Management Part 1: General*. National Institute of Standards and Technology, U.S. Department of Commerce. 2016. URL: *http://dx.doi.org/10.6028/NIST.SP.800-57pt1r4* (dostęp dnia 2016-12-07).

[7] Abdullah Al Hasib i Abul Ahsan Md Mahmudul Haque. „A comparative study of the performance and security issues of AES and RSA cryptography". W: *Convergence and Hybrid Information Technology, 2008. ICCIT'08. Third International Conference on*. T. 2. IEEE. 2008, s. 505–510.

[8] Nils Gura i in. „Comparing Elliptic Curve Cryptography and RSA on 8-bit CPUs". W: *Cryptographic Hardware and Embedded Systems - CHES 2004: 6th International Workshop Cambridge, MA, USA, August 11-13, 2004. Proceedings*. Wyed. Marc Joye i Jean-Jacques Quisquater. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, s. 119–132. ISBN: 978-3-540-28632-5. DOI: *10.1007/978-3-540-28632-5_9*.

[9] Cameron F Kerry. „Digital Signature Standard (DSS)". W: *National Institute of Standards and Technology* (2013).

[10] Sean Turner i in. „Elliptic Curve Cryptography Subject Public Key Information". W: (2009).

[11] Burt Kaliski. „Pkcs# 7: Cryptographic message syntax version 1.5". W: (1998).

Dodatek A. Przykładowe rekordy protokołu komunikacji

```
# Nagłówek:
0x00 0x01 # wersja protokołu
0x00      # typ rekordu: HelloRequest
0x00 0x40 # długość zawartości: 64 bajty

# Zawartość:
# 32 bajty współrzędnej X klucza publicznego
0x12 0x34 0x56 0x78 0x9A 0xBC 0xDE 0xF0
0x12 0x34 0x56 0x78 0x9A 0xBC 0xDE 0xF0
0x12 0x34 0x56 0x78 0x9A 0xBC 0xDE 0xF0
0x12 0x34 0x56 0x78 0x9A 0xBC 0xDE 0xF0

# 32 bajty współrzędnej Y klucza publicznego
0x12 0x34 0x56 0x78 0x9A 0xBC 0xDE 0xF0
0x12 0x34 0x56 0x78 0x9A 0xBC 0xDE 0xF0
0x12 0x34 0x56 0x78 0x9A 0xBC 0xDE 0xF0
0x12 0x34 0x56 0x78 0x9A 0xBC 0xDE 0xF0
```

Rys. A.1. Budowa rekordu typu HelloRequest

```
# Nagłówek:
0x00 0x01 # wersja protokołu
0x01      # typ rekordu: HelloResponse
0x00 0x60 # długość zawartości: 96 bajtów

# Zawartość:
# 16 bajtów kodu uwierzytelniającego
0x12 0x34 0x56 0x78 0x9A 0xBC 0xDE 0xF0
0x12 0x34 0x56 0x78 0x9A 0xBC 0xDE 0xF0

# Zaszyfrowany klucz publiczny (64 bajty)
# z uwzględnieniem dopełnienia PKCS#7 (16 bajtów)
0x12 0x34 0x56 0x78 0x9A 0xBC 0xDE 0xF0
0x12 0x34 0x56 0x78 0x9A 0xBC 0xDE 0xF0
0x12 0x34 0x56 0x78 0x9A 0xBC 0xDE 0xF0
0x12 0x34 0x56 0x78 0x9A 0xBC 0xDE 0xF0
0x12 0x34 0x56 0x78 0x9A 0xBC 0xDE 0xF0
0x12 0x34 0x56 0x78 0x9A 0xBC 0xDE 0xF0
0x12 0x34 0x56 0x78 0x9A 0xBC 0xDE 0xF0
0x12 0x34 0x56 0x78 0x9A 0xBC 0xDE 0xF0
0x12 0x34 0x56 0x78 0x9A 0xBC 0xDE 0xF0
0x12 0x34 0x56 0x78 0x9A 0xBC 0xDE 0xF0
```

Rys. A.2. Budowa rekordu typu HelloResponse

```
# Nagłówek:
0x00 0x01 # wersja protokołu
0x02      # typ rekordu: EncryptedData
0x00 0x20 # długość zawartości: 32 bajty

# Zawartość:
# 16 bajtów kodu uwierzytelniającego
0x12 0x34 0x56 0x78 0x9A 0xBC 0xDE 0xF0
0x12 0x34 0x56 0x78 0x9A 0xBC 0xDE 0xF0

# Zaszyfrowane dane (dopełnione do pełnego bloku AES)
0x12 0x34 0x56 0x78 0x9A 0xBC 0xDE 0xF0
0x12 0x34 0x56 0x78 0x9A 0xBC 0xDE 0xF0
0x12 0x34 0x56 0x78 0x9A 0xBC 0xDE 0xF0
0x12 0x34 0x56 0x78 0x9A 0xBC 0xDE 0xF0
```

Rys. A.3. Budowa rekordu typu EncryptedData

Dodatek B. Przykładowe bloki kodu biblioteki

```
size_t _seconn_crypto_encrypt(void *destination, void *source, size_t length,
    aes128_key_t enc_key) {
    uint8_t *dest = (uint8_t*)destination;
    uint8_t *src = (uint8_t*)source;

    rng(dest, 16); // losowy wektor inicjalizacyjny
    memset(&ctx, 0, sizeof(aes128_ctx_t));

    aes128_init(enc_key, &ctx);

    aes128_enc(dest, &ctx);

    size_t i = 0;
    for(; i+16 <= length; i += 16) {
        memcpy(dest+16+i, src+i, 16);
        _seconn_crypto_xor_block(dest+16+i, dest+i);
        aes128_enc(dest+16+i, &ctx);
    }

    size_t pad_length = 16 - (length % 16);
    memset(dest+16+i, pad_length, 16);
    memcpy(dest+16+i, src+i, length - i);
    _seconn_crypto_xor_block(dest+16+i, dest+i);
    aes128_enc(dest+16+i, &ctx);

    return i+32;
}
```

Listing B.1. Szyfrowanie CBC wraz z obsługą dopełnienia PKCS#7

```
size_t _seconn_crypto_decrypt(void *destination, void *source, size_t length,
    aes128_key_t enc_key) {
    uint8_t *src = ((uint8_t*)source);
    uint8_t *dest = (uint8_t*)destination;

    memset(&ctx, 0, sizeof(aes128_ctx_t));
    aes128_init(enc_key, &ctx);

    size_t i = 0;
    for(; i+16 < length; i += 16) {
        memcpy(dest+i, src+i+16, 16);
        aes128_dec(dest+i, &ctx);
        _seconn_crypto_xor_block(dest+i, src+i);
    }

    size_t pad_length = dest[i-1];
    for(size_t j = 2; j <= pad_length; j++) {
        if (dest[i-j] != pad_length) {
            return 0;
        }
    }

    return i-pad_length;
}
```

Listing B.2. Odszyfrowanie CBC wraz z obsługą dopełnienia PKCS#7

```
void _seconn_crypto_calculate_mac(uint8_t *mac, void *message,
size_t length, aes128_key_t mac_key, aes128_key_t enc_key) {
    memset(&ctx, 0, sizeof(aes128_ctx_t));
    aes128_init(mac_key, &ctx);

    uint8_t *block = mac;

    memset(block, 0, 16); // wektor inicjalizacyjny wypełniony zerami

    // obliczenie ostatniego bloku szyfrowania w trybie CBC
    size_t i = 0;
    for(; i+16 <= length; i += 16) {
        _seconn_crypto_xor_block(block, ((uint8_t*)message)+i);
        aes128_enc(block, &ctx);
    }

    // szyfrowanie ostatniego bloku osobnym kluczem
    memset(&ctx, 0, sizeof(aes128_ctx_t));
    aes128_init(enc_key, &ctx);
    aes128_enc(block, &ctx);
}
```

Listing B.3. Obliczanie MAC dla wiadomości