Certificate Authority CBZ - Dokumentacja kodu

Piotr Pazdan, Maksymilian Oliwa

Styczeń 2026

II rok Cyberbezpieczeństwo, WIEiT

I Wprowadzenie

Niniejszy dokument stanowi udokumentowanie bazy kodu wchodzącego w skład głównego komponentu projektu, czyli programu `ca-cbz`. Program ten umożliwia generowanie żądań podpisu certyfikatu (ang. *certificate signing request*, *CSR*) oraz samych certyfikatów, zarówno tych podpisanych własnoręcznie (ang. *self-signed certificate*), jak i podpisanych za pomocą odrębnego certyfikatu urzędu certyfikacji.

Ze względu na obszerność standardu definiującego infrastrukturę klucza publicznego, program nie wspiera go całego. Limit czasowy projektu zmusił nas do selektywnego podejmowania decyzji na temat wspierania konkretnych schematów, szyfrów, rozszerzeń itd. Niemniej jednak zbiór obecnie zaimplementowanych mechanizmów wystarcza w pełni na efektywne użytkowanie i korzystanie z możliwości PKI.

II Infrastruktura klucza publicznego

Ważną kwestią jest istotność samego konceptu programu. Aby zrozumieć jego przeznaczenie, należy być zaznajomionych z technicznym opisem mechanizmów PKI, głównie wcześniej wspomnianych certyfikatów. Zarysujmy zatem pewnien zestaw kroków które musimy spełnić, aby móc korzystać z bezpieczeństwa które oferuje nam standard - za przykład weźmy obsługę protokołu HTTPS przez serwer sieciowy.

Protokół HTTPS służy weryfikacji (uwierzytelnieniu) serwera komunikującego się z klientem. Aby takie uwierzytelnienie było możliwe, należy stworzyć pewien podmiot autoryzujący zaufane domeny odpowiednim certyfikatem. Rolę takiego podmiotu pełni urząd certyfikacji (ang. *certificate authority*), który w modelu łańcucha zaufania (ang. *chain of trust*) znajduje się na samej jego górze, tzn. jest zaufany z założenia.
Uwaga: w prawdziwym świecie w omawianym modelu mamy także do czynienia z tzw. pośrednimi urzędami certyfikacji (ang. *intermediate certificate authority*), które nie są zaufane z założenia - poświadcza za nimi urząd certyfikacji wyższego szczebla, a zaufane z założenia są główne urzędy certyfikacji (ang. *root certificate authority*) stojące najwyżej w hierarchi. W tym wprowadzeniu świadomie posłużymy się uproszczonym modelem zaufania, w którym urząd certyfikacji jest *de facto* głównym urzędem certyfikacji.

Urząd certyfikacji wystawia certyfikaty domenom, które:

1. prześlą żądanie podpisu certyfikatu do urzędu
2. dostarczą dowodu własności domeny widniejącej w żądaniu (odbywa się przeważnie poprzez wysłanie wiadomości z linkiem autoryzującym na parę adresów mailowych powiązanych z daną domeną)

Po pomyślnym przejściu procesu weryfikacji, certyfikat jest wystawiany na określony czas (z reguły parę miesięcy/lat).

Program `ca-cbz` zawiera funkcjonalność wspierającą obie strony w tym procesie. Oferuje on funkcjonalność potrzebną właścicielowi domeny ubiegającej się o certyfikat (generowanie żądań podpisu certyfikatu, *CSR*), jak i potrzebną urzędowi w celu wystawiania certyfikatów (generowanie certyfikatów podpisanych własnoręcznie oraz zwykłych certyfikatów).

Uwaga: Program **nie wspiera** generowania kluczy kryptograficznych - w tym celu zalecamy użycie gotowego i szeroko stosowanego oprogramowania *OpenSSL*.

III Formaty kodowania obiektów PKI

Dokumenty definiujące infrastrukturę klucza publicznego (RFC 5280, RFC 7468) zakładają, że pliki reprezentujące obiekty tej infrastruktury (klucze kryptograficzne, *CSR*, certyfikaty) będą reprezentowane w formacie *PEM* który jest powiązany z formatem *DER*, który to z kolei wykorzystuje format *ASN.1*. Poniżej krótki opis tych formatów:

- **ASN.1** - standard służący do opisu danych przeznaczonych do reprezentacji, kodowania lub dekodowania
- **DER** - jeden z wariantów kodowania ASN.1, tzn. konwersji abstrakcyjnych obiektów na ciąg bitowy (i *vice versa*)
- **PEM** - format ułatwiający transmisję danych zakodowanych w DER; przewiduje on zakodowanie danych DER kodowaniem Base64 oraz opatrzenie nagłówka oraz stopki pliku odpowiednimi etykietami (ang. *labels*), np. `-----BEGIN CERTIFICATE-----`, `-----END CERTIFICATE-----`

IV Odgórny wzgląd na kod

Baza kodu programu `ca-cbz` składa się z zawartości katalogu `src` oraz pliku `Makefile`, służącego do kompilacji.
Wewnątrz katalogu `src` znajdziemy poniższą zawartość:

- `asn1` - katalog zawierający kod implementujący kodowanie prymitywów zdefiniowanych w standardzie ASN.1
- `encryption` - katalog definiujący prymitywy kryptograficzne, takie jak:
  (a) szyfr *AES-CBC* w odmiennych wariantach długości klucza
  (b) funkcja *HMAC* obliczająca kod uwierzytelnienia wiadomości
  (c) funkcja *PBKDF2* - szeroko stosowana odmiana funkcji z rodziny KDF (ang. *key derivation function*) służąca do wyznaczania klucza odpowiedniej długości na podstawie hasła
- `hash` - katalog z zawartymi algorytmami wyznaczania skrótów wiadomości; znajdują się tu różne warianty algorytmów z rodziny *SHA*
- `pkcs` - najbardziej obszerny katalog; zawiera kod odpowiedzialny za implementację wszelkich mechanizmów powiązanych z *PKCS* (ang. *public-key cryptography standards*); znajduje się tu kod odpowiedzialny za zarządzanie kluczami kryptograficznymi, kodowanie obiektów PKCS zdefiniowanych w RFC 5280 oraz funkcje generujące i weryfikujące podpis cyfrowy.
- `tests` - katalog zawierający tzw. testy jednostkowe (ang. *unit tests*) weryfikujące sprawność różnych fragmentów kodu
- `utils` - katalog z funkcjonalonścią niesprecyzowanego przeznaczenia; głównie funkcje wspomagające sterowaniem wejścia-wyjścia, implementacja kodowania Base64 oraz funkcje związane z czyszczeniem buforów pamięciowych
- `main.cpp` - serce całego programu; znajduje się tu funkcja wejściowa `main` wraz z funkcjami pomocniczymi

V Szczegóły kodu

W tej sekcji przejdziemy przez wszystkie pliki zawarte w projekcie i omówimy detale z nimi związane.

5.1 `asn1/asn1.h`, `asn1/asn1.cpp`

W tych plikach zaimplementowana została cała funkcjonalność związana z obsługą standardu ASN.1. Z racji tego, że obiekty ASN.1 tworzą struktury (niekiedy rozległe i zawiłe), zdecydowaliśmy się na enkapsulację poszczególnych obiektów, efektywnie rozwiązując problem za pomocą programowania obiektowego.
By zrozumieć zawarte mechanizmy, warto przywołać strukturę pojedynczego obiektu według standardu ASN.1 - składa się on z pól `Type-Length-Value`, gdzie:

- `Type` - jednobajtowy typ obiektu (tag)
- `Length` - rozmiar obiektu (wartości), w bajtach
- `Value` - wartość obiektu o rozmiarze `Length`

Podstawowym zdefiniowanym obiektem jest klasa `ASN1Object`, która jest swoistym kontenerem na pojedynczy obiekt ASN.1. Ma ona zdefiniowane następujące pola:

- `_tag` - tag reprezentowanego obiektu (`Type`)
- `_length` - długość wartości obiektu, w bajtach (`Length`)
- `_value` - kontener `std::vector` zawierający ciąg bajtów wartości obiektu
- `_children` - kontener `std::vector` zawierający inne instancje obiektu `ASN1Object` (dzieci), które mają być zinterpretowane jako zawartość obiektu rodzica
- `_encoded` - kontener `std::vector` zawierający zakodowaną w DER reprezentację obiektu - służy jako *cache* podczas wielokrotnego pobierania wartości zakodowanego obiektu.

Uwaga: od tego momentu słowa *obiekt* oraz *klasa* **nie będą** wykorzystywane naprzemiennie; poprzez *obiekt* należy rozumieć abstrakcyjny obiekt reprezentowany przez standard ASN.1 (struktura `Type-Length-Value`); poprzez *klasę* należy rozumieć klasę `ASN1Object`, służącą do reprezentacji abstrakcyjnego obiektu ASN.1 w kodzie.

Klasa posiada również zdefiniowane konstruktory pobierające różne kombinacje powyższych pól. Przeciążane są również operatory `=` oraz `<`, co wykorzystywane jest przy sortowaniu obiektów (wymagane przez obiekt SET, wedle specyfikacji ASN.1). Do najważniejszych zdefiniowanych funkcji należą:

- `value` - zwraca referencję do bufora przetrzymującego wartość obiektu reprezentowanego przez instancję klasy
- `children` - zwraca referencję do bufora przetrzymującego *dzieci* danego obiekty reprezentowanego przez instancję klasy
- funkcje `encode` oraz `decode` - służą odpowiednio do zakodowania stanu danej instancji klasy do formatu DER, oraz do zdekodowania formatu DER do instancji klasy o odpowiadającym stanie

Zaimplementowanych zostało również wiele poszczególnych typów obiektów ASN.1, jak chociażby INTEGER czy OBJECT IDENTIFIER. Klasy reprezentujące te obiekty wykorzystują mechanizm dziedziczenia po klasie nadrzędnej `ASN1Object`, dodając jedynie specyficzne dla swojego obiektu konstruktory oraz przeciążając funkcje `value` oraz `decode`, by zwracały one wartości bardziej specyficzne w kontekście obiektów które reprezentują (przykładowo, funkcja `value` klasy `ASN1Integer` reprezentującej obiekt INTEGER nie zwraca bufora przetrzymującego wartość obiektu, lecz konkretną wartość liczbową reprezentowaną przez ten obiekt).

Klasy definiujące poszczególne obiekty są nazwane według schematu `ASN1{name}`, gdzie *name* to nazwa obiektu w standardzie ASN.1. Do zdefiniowanych w ten sposób klas dziedziczących po `ASN1Object` należą odpowiednio:

- `ASN1ObjectIdentifier`
- `ASN1Integer`
- `ASN1Sequence`
- `ASN1Null`
- `ASN1Boolean`
- `ASN1BitString`
- `ASN1OctetString`
- `ASN1Set`
- `ASN1UTCTime`
- `ASN1GeneralizedTime`

Wyjątek stanowi klasa `ASN1String`, która definiuje łańcuch znaków o dowolnym tagu obiektu - powstała po to, by móc reprezentować niewspierane warianty.
Uwaga: wymienione klasy stanowią reprezentacje tylko części z całości obiektów zdefiniowanych przez standard ASN.1; wedle standardu możliwe jest również definiowanie własnych typów obiektów, co jest stosowane przy tworzeniu obiektów PKCS.

W ramach obsługi obiektów czasu (`ASN1UTCTime`, `ASN1GeneralizedTime`) znajduje się tu również logika odpowiadająca za konwersje dat i obliczanie znaczników czasu.

5.2 `encryption/aes.h`, `encryption/aes.cpp`

Te pliki przechowują kod odpowiedzialny za szyfry z rodziny AES. Ze względu na trudności związane z własnoręczną implementacją szyfru (efektywne napisanie implementacji wymaga pisania kodu w assemblerze, by wykorzystać zestaw instrukcji procesora wspierających te schematy sprzętowo) zdecydowaliśmy się skorzystać z istniejących implementacji w bibliotece `libopenssl`. Zawartość plików sprowadza się więc do zdefiniowania typów poszczególnych rozmiarów klucza, oraz zarządzania pobranymi z biblioteki instancjami algorytmów.

5.3 `encryption/hmac.hpp`

Plik zawiera interfejs funkcji pseudolosowej (ang. *pseudo-random function*), który należy spełnić przy definiowaniu implementacji interfejsu. Znajduje się tu również szablon kryptograficznej funkcji HMAC, którego można użyć z różnymi wariantami zaimplementowanych funkcji skrótu.

5.4 `encryption/kdf.hpp`

Plik zawiera szablon funkcji kryptograficznej PBKDF2, której można użyć z różnymi wariantami funkcji pseudolosowej.

5.5 `hash/sha.h`, `hash/sha.cpp`

Te pliki przechowują kod odpowiedzialny za funkcje skrótu; podobnie jak w przypadku szyfrów AES, zdecydowaliśmy się pobierać instancje potrzebnych algorytmów z biblioteki `libopenssl`.

5.6 `pkcs/labels.h`

Plik przechowuje zdefiniowane nagłówki oraz stopki wspieranych obiektów PKCS; są to:

- klucz prywatny (szyfrowany i nieszyfrowany)
- żądanie podpisu certyfikatu (*CSR*)
- certyfikat

5.7 `pkcs/pkcs.h`, `pkcs/pkcs.cpp`

Są to dwa największe pliki pod względem linii kodu w całym projekcie; definiują one rozmaite obiekty PKCS na podstawie struktur zdefiniowanych w RFC 5280, wykorzystując przy tym m.in. wcześniej omówione klasy reprezentujące obiekty ASN.1. Dla każdej klasy zdefiniowane są funkcje pomocnicze zwracające poszczególne pola instancji tych klas, oraz funkcje odpowiedzialne za kodowanie do DER i dekodowanie z DER stanów poszczególnych instancji. Zdefiniowane są również wspierane algorytmy klucza prywatnego, schematy szyfrowania (ang. *encryption schemes*), funkcje KDF, HMAC, algorytmy szyfrowania (ang. *encryption algorithms*) oraz funkcje kodujące poszczególne prymitywy i ich kombinacje do formatu zgodnego ze specyfikacją PKCS.

5.8 `pkcs/private_key.h`, `pkcs/private_key.cpp`

Pliki te definiują klasę `RSAPrivateKey`, używaną przez program do obsługi prywatnych kluczy kryptograficznych opartych na schemacie RSA. Klasa zawiera konstruktory pozwalające na odczyt istniejącego klucza z pliku; obsługiwany jest zarówno wariant nieszyfrowany, jak i klucz szyfrowany - wówczas przy otwieraniu należy podać hasło rozszyfrowujące zawartość klucza.

5.9 `pkcs/public_key.h`, `pkcs/public_key.cpp`

Te dwa pliki definiują klasę `RSAPublicKey`, definiującą klucz publiczny oparty na schemacie RSA. Wykorzystywana jest ona w niektórych obiektach PKCS posiadających stosowną sekcję opisującą klucz publiczny podmiotu wystawiającego bądź odbierającego certyfikat.

5.10 `pkcs/sign.h`, `pkcs/sign.cpp`

W tych plikach zawarta jest logika dotycząca mechanizmów podpisu cyfrowego - zaimplementowana funkcja to `RSASSA-PKCS1-v1_5` zawarta w RFC 8017. W kodzie zdefiniowane są dwie funkcje: pierwsza, `RSASSA-PKCS1-v1_5_SIGN`, podpisujące określony bufor podanym kluczem; druga, `RSASSA-PKCS1-v1_5_VERIFY`, weryfikuje istniejący już podpis.

5.11 `utils/base64.h`, `utils/base64.cpp`

Tu znajduje się implementacja kodowania Base64.

5.12 `utils/endianness.hpp`

Krótki plik definiujący dwie funkcje modyfikujące dany typ przetrzymujący liczbę całkowitą do postaci *Big Endian* oraz *Little Endian*, w zależności od sprzętowej i systemowej specyfikacji domyślnej kolejności bajtów.

5.13 `utils/io.h`, `utils/io.cpp`

Pliki zawierające funkcje pomocnicze zarządzające buforami wejścia-wyjścia - to za ich pomocą dane są odczytywane z wejścia bądź zapisywane na dysk.

5.14 `utils/security.hpp`

Plik zawierający wszelakie funkcje związane z bezpieczeństwem; znajdują się tu zarówno funkcje czyszczące bufor, jak i funkcja bezpiecznego odczytywania danych z pliku `secure_read_file` (domyślny obiekt `std::istream` w C++ nie pozwala na zarządzanie buforem wewnętrznym, więc istnieje ryzyko niechcianego wycieku przy odczytywaniu krytycznych danych). Jest tu też bezpieczna implementacja funkcji porównującej dwa bufory pamięci ciągłej, odporna na ataki kanałem bocznym (czas).

5.15 `utils/utils.hpp`

Ten plik zawiera rozmaite funkcje które ciężko przenieść w inne miejsca - są tu m.in. funkcje odpowiadające za wypisywanie czytelnie sformatowanych bajtów na ekran, funkcje zarządzające wyjątkami w C++, znalazła się tu również naiwna implementacja dwukierunkowego słownika (ang. *bidirectional map*)

5.16 `main.cpp`

Tu główne skrzypce odgrywa funkcja `main` oraz jej pomocnicy, m.in. funkcja `handle_arguments` przetwarzająca przekazane do programu argumenty i odpowiednio sterująca wykonaniem programu. Są tu również funkcje generujące certyfikaty (`gen_self_signed_cert`, `gen_cert`) oraz żądania podpisu certyfikatu (`gen_csr`), które z kolei wykorzystują implementacje obiektów PKCS zawarte w plikach `pkcs/pkcs.h` oraz `pkcs/pkcs.cpp`.