KARTY SMART CARD W KRYPTOGRAFII

JAK TO ZROBIĆ WE WŁASNYM ZAKRESIE?

SPIS TREŚCI

Grzegorz Kmita, Maciej Kasprzyk

22.12.2020

Kryptografia, Cyberbezpieczeństwo, semestr 3

WPROWADZENIE

Ten projekt ma na celu przybliżenie w jaki sposób można korzystać z karty inteligentnej w warunkach domowych. Jak zapisać i odczytać dane z karty i jakie narzędzia są do tego niezbędne. Podczas tworzenia projektu naszym celem było uzyskanie dostępu do karty - zapis i odczyt, jak również użycie zdobytej wiedzy w praktycznych zastosowaniach - szyfrowanie i deszyfrowanie plików używając klucza zapisanego na karcie

Do komunikacji z kartą będziemy używali biblioteki winscard.h dostarczoną z projektem PCSC-lite, do którego linki odnośniki znajdują się w bibliografii oraz w dalszej części tego dokumentu. Aby program scard.c działał, będzie potrzebna instalacja jednej biblioteki, która zostanie wyspecyfikowana w sekcji przygotowanie środowiska.

Do szyfrowania plików będziemy używać algorytmu AES z 128-bitowym kluczem.

CO BĘDZIE POTRZEBNE?

W naszym projekcie korzystamy z czytnika ACR39U oraz kart pamięciowych SLE5542. Dla takiego zestawu został stworzony program i tylko w takim układzie możemy zagwarantować jego poprawne działanie. Program był testowany w systemie Kali Linux. Aby program zadziałał potrzebne będą:

1. Czytnik ACR39U
2. Karta pamięciowa SLE5542
3. Wybrana dystrybucja systemu Linux
4. Pliki dołączone do projektu
5. Biblioteka zainstalowana wg instrukcji w sekcji przygotowanie środowiska

PRZYGOTOWANIE ŚRODOWISKA

Aby program scard.c mógł się skompilować potrzebna będzie instalacja podanej biblioteki z repozytorium wybranej dystrybucji Linuxa

- **libpcsclite-dev** (sudo apt-get install -y libpcsclite-dev)

Dodatkowo trzeba pamiętać, aby nie modyfikować, przenosić lub usuwać folderów PCSC oraz AES. Foldery te zawierają niezbędne w procesie kompilacji pliki służące do komunikacji z czytnikiem oraz kartą, jak również do szyfrowania i deszyfrowania plików.

SCARD.C

Program scard.c zawiera dużo komentarzy dotyczących zawartego w nim kodu. Jednak aby lepiej zrozumieć jego działanie opiszemy w tej sekcji najważniejsze części tego programu.

Na początku znajduje się definicja zmiennych potrzebnych do komunikacji z czytnikiem.

```
// ZMIENNE POTRZEBNE DO KOMUNIKACJI Z CZYTNIKIEM
LONG rv;
SCARDCONTEXT hContext;
SCARDHANDLE hCard;
DWORD dwReaders, dwActiveProtocol, dwRecvLength;
LPTSTR mszReaders;
SCARD_IO_REQUEST pioSendPci;
BYTE cmdDefineCardType[] = { 0xFF, 0xA4, 0x00, 0x00, 0x01, 0x06 };
BYTE cmdWriteCard[] = { 0xFF, 0xD0, 0x00 };
BYTE pbRecvBuffer[266];
BYTE pbSendBuffer[266];
BYTE offset = 32;
unsigned int i, sz, remaining;
unsigned int length = 256;
unsigned int nbytes = 0;
unsigned int msglen = 0;
char *ptr, **readers = NULL;
int fd,nbReaders;
int rid = -1;
```

Są to zmienne, które będziemy przesyłać w argumentach funkcji komunikujących się z kartą i wraz z działaniem programu będą przechowywać wartości takie jak wybrany czytnik czy aktywna karta. Zmienne będą opisane w momencie, w którym jakaś funkcja będzie z nich korzystać.

SCardEstablishContext

Zaczniemy od funkcji SCardEstablishContext, która ustala ogólny kontekst w komunikacji i przyjmuje następujące wartości:

```
LONG SCardEstablishContext(
  DWORD          dwScope,
  LPCVOID        pvReserved1,
  LPCVOID        pvReserved2,
  LPSCARDCONTEXT phContext
);
```

- **dwScope** - określenie czy "menadżerem" kontekstu będzie obecny użytkownik, czy system. Może przyjmować wartość **SCARD_SCOPE_USER** (jeżeli użytkownik) albo **SCARD_SCOPE_SYSTEM** (jeżeli system).
- **pvReserved1, pvReserved2** - pola zarezerwowane, ich wartość musi wynosić NULL
- **phContext** - wskaźnik do zdefiniowanej na początku zmiennej hContext. Po wykonaniu tego polecenia można korzystać z hContext w innych funkcjach.

Funkcja zwraca **SCARD_S_SUCCESS,** jeżeli operacja się powiodła lub kod błędu, jeżeli się nie powiodła.

W naszym programie funkcja wygląda następująco:

```
rv = SCardEstablishContext(SCARD_SCOPE_SYSTEM, NULL, NULL, &hContext);
CHECK("SCardEstablishContext", rv);
```

CHECK jest zdefiniowane na początku pliku i sprawdza, czy funkcja zakończyła się sukcesem, a jeżeli nie to kończy działanie programu. Będzie on występował na przestrzeni całego programu.

SCardListReaders

Ta funkcja ustala jakie są dostępne w systemie czytniki kart inteligentnych, a jej składnia wygląda następująco:

```
LONG SCardListReadersA(
  SCARDCONTEXT hContext,
  LPCSTR       mszGroups,
  LPSTR        mszReaders,
  LPDWORD      pcchReaders
);
```

- **hContext** - kontekst w komunikacji z kartą ustalony podczas wywołania SCardEstablishContext.
- **mszGroups** - ustalenie w zakresie jakiej grupy należy szukać czytników, NULL - wszystkie grupy
- **mszReaders** - wskaźnik do zdefiniowanej wcześniej zmiennej, tutaj zostanie zapisany bufor przechowujący nazwy wszystkich znalezionych czytników
- **pcchReaders** - wskaźnik do definiowanej zmiennej, przechowuje długość bufora z poprzedniego argumentu, czyli sumę długości nazw znalezionych czytników

Zwrócona wartość może być sukcesem, ale również informacją, że grupa nie zawiera żadnych czytników, znaleziony czytnik jest niedostępny lub innym kodem błędu.

W naszym programie funkcja wygląda następująco:

```
rv = SCardListReaders(hContext, NULL, (LPTSTR)&mszReaders, &dwReaders);
CHECK("SCardListReaders", rv)
```

Znając długość bufora i jego zawartość można określić ile czytników znajduje się w systemie i utworzyć na przykład tablicę z dostępnymi czytnikami. W następnych funkcjach będziemy musieli określać jaki czytnik wybieramy, więc trzeba gdzieś przechować jego nazwę. Do czytnika będziemy się odwoływać właśnie po jego nazwie.

SCardConnect

Funkcja pozwalająca na nawiązanie połączenia między menedżerem kontekstu a kartą inteligentną umieszczoną w wybranym czytniku. Składnia wygląda następująco:

```
LONG SCardConnectA(
  SCARDCONTEXT  hContext,
  LPCSTR        szReader,
  DWORD         dwShareMode,
  DWORD         dwPreferredProtocols,
  LPSCARDHANDLE phCard,
  LPDWORD       pdwActiveProtocol
);
```

- **hContext** - kontekst w komunikacji z kartą ustalony podczas wywołania SCardEstablishContext.
- **szReader** - nazwa czytnika, który zawiera kartę, z którą chcemy się połączyć
- **dwShareMode** - informacja czy inne aplikacje mogą się komunikować z kartą po udanym połączeniu się przez nas. Możliwe opcje to:
  - **SCARD_SHARE_SHARED** - dzielenie się kartą dozwolone
  - **SCARD_SHARE_EXCLUSIVE** - dzielenie się dozwolone, ale niepożądane
  - **SCARD_SHARE_DIRECT** - dzielenie się zablokowane, nasza aplikacja ma kartę na wyłączność
- **dwPreferedProtocols** - protokoły komunikacyjne, które są akceptowalne przez nas. Może to być jeden protokół lub więcej z operatorem OR. Dozwolone protokoły to **SCARD_PROTOCOL_T0** oraz **SCARD_PROTOCOL_T1**
- **phCard** - wskaźnik do zdefiniowanej wcześniej zmiennej, tutaj będzie przechowywane połączenie z kartą inteligentną.
- **pdwActiveProtocol** - wskaźnik do zdefiniowanej wcześniej zmiennej. W niej zostanie umieszczony wybrany protokół komunikacyjny.

Funkcja zwraca kod błędu, jeżeli operacja się nie udała lub sukces, jeżeli się udała.

W naszym programie funkcja wygląda następująco:

```
rv = SCardConnect(hContext, readers[rid],
SCARD_SHARE_DIRECT, SCARD_PROTOCOL_T0 | SCARD_PROTOCOL_T1, &hCard, &dwActiveProtocol);
CHECK("SCardConnect", rv)
```

SCardTransmit

Prawdopodobnie najważniejsza funkcja, ponieważ to z jej pomocą możemy faktycznie komunikować się z kartą. Jest składnia jest najbardziej złożone ze wszystkich opisanych funkcji i wygląda następująco:

```
LONG SCardTransmit(
  SCARDHANDLE         hCard,
  LPCSCARD_IO_REQUEST pioSendPci,
  LPCBYTE             pbSendBuffer,
  DWORD               cbSendLength,
  LPSCARD_IO_REQUEST  pioRecvPci,
  LPBYTE              pbRecvBuffer,
  LPDWORD             pcbRecvLength
);
```

- **hCard** - zmienna opisująca połączenie z kartą, której wartość została ustalona podczas wywołania funkcji SCardConnect
- **pioSendPci** - wskaźnik na wybrany protokół w formacie **SCARD_IO_REQUEST.** To, że ta zmienna jest w tym formacie zapewniliśmy umieszczając w kodzie dwie operacje:

```
memset(&pioSendPci,0,sizeof(SCARD_IO_REQUEST));
```

```
switch(dwActiveProtocol){
    case SCARD_PROTOCOL_T0:
        pioSendPci = *SCARD_PCI_T0;
        break;
    case SCARD_PROTOCOL_T1:
        pioSendPci = *SCARD_PCI_T1;
        break;
    case SCARD_PROTOCOL_RAW:
        pioSendPci = *SCARD_PCI_RAW;
        break;
    default:
        break;
}
```

To jaka wartość ma być umieszczona pod tą zmienną trzeba określić na podstawie wartości umieszczonej w zmiennej dwActiveProtocol podczas wywołania SCardConnect

- **pbSendBuffer** - wskaźnik na dane, które mają zostać przesłane do karty. Wskazywana struktura musi tworzyć komendę APDU, która zostanie obsłużona przez czytnik i kartę. Jeżeli protokół komunikacyjny to T0 (a w naszym przypadku taki właśnie jest) to struktura danych musi wyglądać następująco:

```
struct {
    BYTE
        bCla,   // the instruction class
        bIns,   // the instruction code
        bP1,    // parameter to the instruction
        bP2,    // parameter to the instruction
        bP3;    // size of I/O transfer
} CmdBytes;
```

  - **bCla** - klasa instrukcji/polecenia, którego chcemy użyć
  - **bIns** - kod instrukcji/polecenia, którego chcemy użyć
  - **bP1, bP2** - parametry instrukcji - na przykład przesunięcie strumienia danych, który będziemy chcieli przeczytać/ zapisać
  - **bP3** - długość danych, które zostaną przesłane do karty
  - następnie następuje ciąg danych o długość określonej w polu wcześniej
- **cbSendLength** - długość danych w bajtach jakie znajdują się w pbSendBuffer. Jest to oczywiście rozmiar czterech pól w poprzednim polu oraz danych, które chcemy wysłać
- **pioRecvPci** - wskaźnik do zmiennej przechowującej protokół, który jest używany do odpowiedzi przez czytnik. Wartość może być równa NULL i takiej będziemy używać
- **pbRecvBuffer** - wskaźnik do zmiennej wcześniej utworzonej, która będzie przechowywać odpowiedź karty na wysłaną komendę APDU. Oprócz danych, które nas interesują - na przykład czytanie klucza - w tej zmiennej przechowywana jest także dwubajtowa odpowiedź, w której znajdują się informacje o powodzeniu lub porażce, a także wielu innych statusach. To uzasadnia, dlaczego w niektórych fragmentach kodu odejmujemy dwa ostatnie bajty z odpowiedzi.
- **pcbRecvLength** - wskaźnik na zmienną zdefiniowaną na początku pliku, w której przechowana jest długość odpowiedzi, czyli rozmiar faktycznych danych zwróconych przez czytnik.

Funkcja zwraca kod błędu, jeżeli operacja się nie udała albo SCARD_S_SUCCESS, jeżeli operacja się powiodła.

W naszym programie wielokrotnie stosujemy tę funkcję. Poniżej znajdują się trzy przykłady - zapis, odczyt i sprawdzenie kodu do zapisu na karcie.

Sprawdzenie kodu do zapisu:

```
// SPRAWDZANIE CZY KOD DO ZAPISU NA KARTĘ JEST POPRAWNY
pbSendBuffer[0] = 0xFF;
pbSendBuffer[1] = 0x20;
pbSendBuffer[2] = 0x00;
pbSendBuffer[3] = 0x00;
pbSendBuffer[4] = 0x03;
msglen = 8;
memcpy(&pbSendBuffer[5],&writeCode,3);
dwRecvLength = sizeof(pbRecvBuffer);
rv = SCardTransmit(hCard, &pioSendPci, pbSendBuffer, msglen, NULL, pbRecvBuffer, &dwRecvLength);
CHECK("SCardTransmit (unlock)", rv);
CHECK_RESPONSE(pbRecvBuffer, dwRecvLength, 0x07);
```

Widać, że pole pbSendBuffer[4], czyli bP3, zawiera rozmiar danych - 3 bajtowy kod, a sam kod znajduje się w pbSendBuffer[5]. Ta komenda niejako uwierzytelnia nas przed kartą i po jej wykonaniu możemy już zapisywać na karcie informacje, ponieważ mamy takie uprawnienia. Jeżeli nie wykonałoby się tej komendy, można by było jedynie czytać dane z karty.

Zapis danych na karcie:

```
// ZAPIS 128 bitowego KLUCZA [32-48]
memset(pbSendBuffer,0,265);
memset(pbRecvBuffer,0,265);
nbytes = 16; // rozmiar pola do zapisu
offset = 32; // przesunięcie w pamięci karty
pbSendBuffer[0] = 0xFF;
pbSendBuffer[1] = 0xD0;
pbSendBuffer[2] = 0x00;
pbSendBuffer[3] = offset;
pbSendBuffer[4] = nbytes;
msglen = 5 + nbytes; // łączna długość wiadomości 5-potrzebne dane w protokole komunikacji z kartą
key_gen(key);
memcpy(&pbSendBuffer[5], key, 16);

dwRecvLength = sizeof(pbRecvBuffer);
rv = SCardTransmit(hCard, &pioSendPci, pbSendBuffer, msglen, NULL, pbRecvBuffer, &dwRecvLength);
CHECK("SCardTransmit", rv);
CHECK_RESPONSE(pbRecvBuffer, dwRecvLength, 0x00);
```

Znając działanie funkcji SCardTransmit oraz strukturę pamięci karty pamięciowej można zapisywać dane o odpowiedniej długości i odpowiednim przesunięciu na karcie.

Odczyt danych z karty:

```
// ODCZYTANIE KLUCZA
offset = 32;
length = 16;
pbSendBuffer[0] = 0xFF;
pbSendBuffer[1] = 0xB0;
pbSendBuffer[2] = 0x00;
pbSendBuffer[3] = offset;
pbSendBuffer[4] = length;
msglen = 5;

dwRecvLength = sizeof(pbRecvBuffer);
rv = SCardTransmit(hCard, &pioSendPci, pbSendBuffer, msglen, NULL, pbRecvBuffer, &dwRecvLength);
CHECK("SCardTransmit", rv)
CHECK_RESPONSE(pbRecvBuffer, dwRecvLength, 0x00);
sor = dwRecvLength - 2;
response = malloc(sor * sizeof(char));
aes_key = response;
memcpy(response, pbRecvBuffer, sor * sizeof(char));
fprintf(stdout, "KLUCZ: %s\n", response);
```

W przypadku czytania danych z karty, również określa się długość i przesunięcie, od którego mamy zacząć czytanie, a dane umieszczone w pbRecvBuffer można zapisać do innej zmiennej, a następnie z nich korzystać.

SCardDisconnect

Podobnie jak przy postępowaniu z plikami, czy dynamicznie alokowaną pamięcią, połączenie z kartą trzeba zakończyć, aby ktoś inny mógł je rozpocząć. Ta funkcja jest jedną z trzech, których używa się do zakończenia pracy z czytnikiem. Jej składnia wygląda następująco:

```
LONG SCardDisconnect(
  SCARDHANDLE hCard,
  DWORD       dwDisposition
);
```

- **hCard** - zmienna przechowująca połączenie z kartą
- **dwDisposition** - akcja jaka ma zostać podjęta podczas kończenia połączenia. Dostępne opcje to:
  - SCARD_LEAVE_CARD - nie podejmuj żadnej akcji
  - SCARD_RESET_CARD - zresetuj kartę
  - SCARD_UNPOWER_CARD - odetnij zasilanie karty
  - SCARD_EJECT_CARD - wysuń kartę (oczywiście logicznie, nie fizycznie)

Funkcja zwraca kod błędu lub sukces, zależnie od wyniku.

W naszym programie wygląda ona następująco:

```
rv = SCardDisconnect(hCard, SCARD_LEAVE_CARD);
CHECK("SCardDisconnect", rv);
```

SCardFreeMemory

Kolejna funkcja w procesie zakończenia pracy z czytnikiem. Ta funkcja zwalnia pamięć zarezerwowaną przez menadżera kontekstu. Można użyć jej wielokrotnie, dla każdego zarezerwowanego miejsca w pamięci na potrzeby komunikacji z czytnikiem. Jej składnia wygląda następująco:

```
LONG SCardFreeMemory(
  SCARDCONTEXT hContext,
  LPCVOID      pvMem
);
```

- **hContext** - zmienna przechowująca kontekst
- **pvMem** - pamięć, która ma zostać zwolniona, po tym jak została zarezerwowana w poprzednich funkcjach

Funkcja zwraca kod błędu lub sukces.

SCardReleaseContext

Ostatnia funkcja, która jest klamrą zamykającą całą pracę z czytnikiem. Zaczynaliśmy od ustalenia kontekstu, a teraz musimy go zwolnić. Składnia tej funkcji jest bardzo prosta:

```
LONG SCardReleaseContext(
  SCARDCONTEXT hContext
);
```

- **hContext** - zmienna przechowująca kontekst w komunikacji z kartą.

Funkcja zwraca kod błędu lub sukces.

W naszym programie dwie powyższe funkcje wyglądają następująco:

```
rv = SCardFreeMemory(hContext, mszReaders);
CHECK("SCardFreeMemory", rv);
rv = SCardReleaseContext(hContext);
CHECK("SCardReleaseContext", rv);
```

Inne

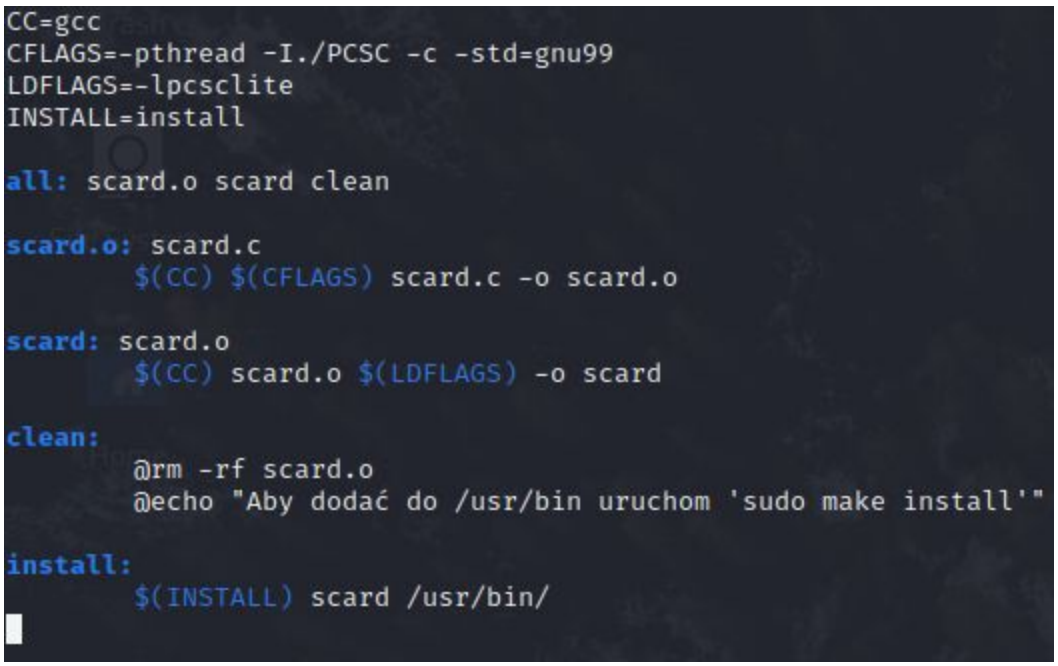
Istnieje dużo więcej funkcji do pracy z kartą i czytnikiem. Omówione powyżej wystarczą do stworzenia działającego programu. Wszystkie funkcje znajdują się w pliku winscard.h, a ich opis znajduje się na tej stronie.

Inne, przykładowe funkcje to na przykład

- **SCardAudit** - zapisuje logi z pracy z kartą i czytnikiem
- **SCardAddReaderToGroup** - pozwala na dodanie czytnika do grupy
- **SCardTransmitCount** - pozwala określić ile operacji Transmit było wykonanych od momentu podpięcia czytnika do systemu
- **SCardIntroduceReader** - określa nową nazwę dla istniejącego czytnika
- **SCardStatus** - zwraca obecny status karty w czytniku

MAKEFILE

Plik makefile znajdujący się w archiwum Scard.tar zawiera polecenia kompilujące program z potrzebnymi opcjami oraz polecenie czyszczące pliki powstałe w procesie kompilacji. Na początku wyspecyfikowany jest kompilator oraz opcje jakich należy użyć. Plik ten zawiera również polecenie umożliwiające dodanie programu scard do /usr/bin. Aby to zrobić trzeba wpis w terminalu ‘sudo make install’. Po tej operacji będzie można używać programu scard z każdego miejsca w systemie plików, bez podawania ścieżki dostępu do pliku.



```
CC=gcc
CFLAGS=-pthread -I./PCSC -c -std=gnu99
LDFLAGS=-lpcsclite
INSTALL=install

all: scard.o scard clean

scard.o: scard.c
	$(CC) $(CFLAGS) scard.c -o scard.o

scard: scard.o
	$(CC) scard.o $(LDFLAGS) -o scard

clean:
	@rm -rf scard.o
	@echo "Aby dodać do /usr/bin uruchom 'sudo make install'"

install:
	$(INSTALL) scard /usr/bin/
```

AES

W tym folderze znajdziemy pliki, które są nam potrzebne do szyfrowania naszych danych algorytmem AES z kluczem o długości 128 bitów zapisanym na karcie inteligentnej. Używamy dopełnienia PKCS#7. Implementację tę zaczerpnęliśmy ze strony i przystosowaliśmy na potrzeby naszego projektu.

Szyfr operuje na blokach 16-bajtowych, w trybie CBC (możliwe jest także użycie trybów ECB i CTR). W związku z tym, aby móc szyfrować dane niebędące wielokrotnością 16 bajtów, wykorzystujemy padding.

aes.h

Jest to plik nagłówkowy, znajdziemy w nim deklaracje funkcji C i definicje makr. Dzięki niemu możemy na przykład wybrać nasz tryb szyfrowania.

```
#ifndef CBC
  #define CBC 1
#endif

#ifndef ECB
  #define ECB 1
#endif

#ifndef CTR
  #define CTR 1
#endif
```

Czy zastąpić domyślny rozmiar klucza 128-bitowego na 192 lub 256-bitowy.

```
#define AES128 1
//#define AES192 1
//#define AES256 1
```

Należy pamiętać o cyfrach 1/0, które oznaczają, że dany tryb działania jest włączony: 1 lub wyłączony: 0.

- Tryb EBC jest uważany za niebezpieczny, dlatego sugerujemy używać dwóch pozostałych trybów szyfrowania
- Mamy padding, więc nie należy przejmować się długością danych.
- UWAGA: nigdy nie używać wektora IV z tym samym kluczem!

aes.c

Ta implementacja jest zweryfikowana na podstawie danych zawartych w: National Institute of Standards and Technology Special Publication 800-38A 2001 ED

Dodatek F: Przykładowe wektory trybów działania AES.

pkcs7_padding.c

Istnieje wiele metod dopełnień, których można użyć Bruce Schneier proponuje dwie raczej proste metody:

- jedna to dopełnienie ostatniego bloku n bajtami, wszystkie z wartością n. Ten tryb dopełnienia jest znany jako padding PKCS # 7 (dla bloków 16-bajtowych), którego użyjemy,
- druga polega na dołączeniu bajtu o wartości 0x80, po którym następuje tyle bajtów zerowych ile potrzeba do wypełnienia ostatniego bloku. Ta metoda jest znana jako dopełnienie ISO, co jest opisane w dokumencie ISO / IEC 9797-1 2.

My w naszym projekcie zastosowaliśmy pierwszą opcję.

Dzięki wykorzystaniu paddingu nie musimy się więcej martwić, aby klucz czy wiadomość były wielokrotnością 16 bajtów (co jest wymagane ze względu na sposób działania algorytmu).

Dzięki poniższej operacji aktualizujemy długość wiadomości tak, aby była odpowiednio wielokrotnością 16 bajtów.

```
// PADDING
int dlenu = dlen;
if (dlen % 16) {
    dlenu += 16 - (dlen % 16);
}
```

Tworzymy dwie tablice znaków wypełnione zerami zgodnie z długością, którą otrzymaliśmy po dopełnieniu, a potem wypełniamy je odpowiednio ciągami znaków.

```
// DEFINIOWANIE TABLIC DO PRZECHOWYWANIA DANYCH WRAZ Z DOPEŁNIENIEM
uint8_t hexarray[dlenu];
uint8_t kexarray[klenu];
memset( hexarray, 0, dlenu );
memset( kexarray, 0, klenu );

// TWORZENIE TABLIC Z WARTOŚCIAMI LICZBOWYMI ZAMIAST ZNAKOWYMI
for (int i=0;i<dlen;i++) {hexarray[i] = (uint8_t)buffer[i];}
for (int i=0;i<klen;i++) {kexarray[i] = (uint8_t)aes_key[i];}
```

Funkcja, której używamy jako pierwszej: **pkcs7_padding_pad_buffer** na wejściu przyjmuje tablicę znaków, wraz z jej początkową długością, następnie zwraca tablicę znaków i liczbę dodanych znaków dopełnienia.

```
// STOSOWANIE PADDINGU
int reportPad = pkcs7_padding_pad_buffer(hexarray, dlen, sizeof(hexarray), 16);
int keyPad = pkcs7_padding_pad_buffer(kexarray, klen, sizeof(kexarray), 16);
```

Zawsze należy pamiętać o inicjalizacji AES za pomocą wektora IV i klucza, zarówno przy szyfrowaniu jak i odszyfrowywaniu:

```
// SZYFROWANIE I ZAPIS DO PLIKU
AES_init_ctx_iv(&ctx, kexarray, iv);
AES_CBC_encrypt_buffer(&ctx, hexarray, dlenu);
fwrite(hexarray, sizeof(hexarray), 1, encrypted);


// DESZYFROWANIE I ZAPIS DO PLIKU
AES_init_ctx_iv(&ctx, kexarray, iv);
AES_CBC_decrypt_buffer(&ctx, hexarray, dlenu);
fwrite(hexarray, dlenu, 1, message);
```

pkcs7_padding.h

W tym pliku nagłówkowym są zdefiniowane makra, załączone biblioteki i funkcje **pkcs7_padding_pad_buffer** oraz **pkcs7_padding_valid**.

PCSC

Folder PCSC zawiera pliki niezbędne w procesie kompilacji programu i służące do utworzenia połączenia z czytnikiem i komunikacji z kartą. Najważniejszym plikiem w tym zestawie jest winscard.h, którego niektóre funkcje omówiliśmy we wcześniejszej sekcji.

Pliki w tym folderze należą do otwartego projektu pcsc-lite, którego autorem jest Ludovic Rousseau. Cały projekt można znaleźć na GitHubie (https://github.com/LudovicRousseau/PCSC) lub na oficjalnej stronie projektu (https://pcsclite.apdu.fr/).

PODSUMOWANIE

Udało nam się zapisać dane na karcie, a następnie je odczytać. Znając działanie funkcji, które w tym celu stosowaliśmy oraz strukturę pamięci karty pamięciowej SLE5542 ustaliliśmy w jakich segmentach pamięci będą przechowywane dane - klucz, dane właściciela oraz czas utworzenia klucza. Następnie zaimplementowaliśmy algorytm AES do programu. Udało nam się zaszyfrować pliki oraz z powodzeniem je odszyfrować.

Program może być stosowany do wielu innych zadań, ponieważ szyfrowanie i deszyfrowanie to jedne z wielu operacji jakie można zaimplementować. Prostota kodu pozwala na jego szybkie zrozumienie i modyfikację w zależności od potrzeb.

BIBLIOGRAFIA

1. https://docs.microsoft.com/en-us/windows/win32/api/winscard/
2. https://github.com/SUNET/memcard-tools
3. https://github.com/LudovicRousseau/PCSC/tree/master/src/PCSC
4. https://github.com/kokke/tiny-AES-c
5. https://erevos.com/blog/tiny-aes-cbc-mode-pkcs7-padding-written-c/