MKOI 17Z - Bezpieczny magazyn plików

Artur M. Brodzki, Kuba Guzek
Prowadzący: Albert Sitek

18 stycznia 2018

1 Temat projektu

Tematem projektu jest implementacja aplikacji chmurowej do przechowywania plików na zdalnym serwerze. Bezpieczeństwo transmisji jest zapewnione przez szyfrowanie danych algorytmem Serpent. Klucz szyfrowania jest uzgadniany asymetrycznie według protokołu Diffiego - Hellmana. Struktura katalogów użytkowników jest płaska - nie ma możliwości tworzenia podfolderów. Lista użytkowników uprawionych do korzystania z systemu jest tworzona przez administratora.

2 Opis wykorzystywanych algorytmów

2.1 Protokół Diffiego - Hellmana

Wykorzystujemy algorytm negocjowania klucza sesyjnego z użyciem dużych liczb pierwszych [1]:

- Klient i serwer współdzielą dużą liczbę pierwszą p. Jest nią kryptograficznie bezpieczna liczba pierwsza RFC3526-6144 o długości - jak wskazuje nazwa - 6144 bitów, czyli 768 bajtów. Współdzieloną liczbą jest też podstawa potęgi $g = 2$.
- Klient losuje sekretną liczbę całkowitą a i wysyła serwerowi liczbę $A = g^a \ mod \ p$. Analogicznie, serwer losuje liczbę całkowitą b i wysyła klientowi liczbę $B = g^b \ mod \ p$.
- Klient otrzymuje od serwera liczbę B i oblicza klucz sesyjny $k = B^a \ mod \ p$. Analogicznie, serwer otrzymuje od klienta liczbę A i oblicza ten sam klucz sesyjny $k = A^b \ mod \ p$.

2.2 Protokoł Diffiego - Hellmana

2.3 Algorytm Serpent

Szczegółowa specyfikacja algorytmu Serpent znajduje się w załącznikach do artykułu dostępnego w zasobach Uniwersytetu Cambridge [2], więc nie będziemy jej tutaj przytaczać w szczegółach. Przedstawimy jednak wszystkie zasadnicze etapy wykonywania tego algorytmu szyfrowania.

Serpent jest szyfrem blokowym operującym na 128-bitowych blokach. 128-bitowy fragment wiadomości jest mapowany na 128-bitowy szyfrogram. Blok podzielony jest na 4 słowa 32-bitowe w konwencji little-endian. Szyfrowanie wykonuje kolejno następujące operacje:

- Permutacja początkowa IP, zdefiniowana w załączniku A.5 do [2]
- 32 rundy szyfrowania: wynik działania rundy R_i staje się wejściem rundy R_{i+1}. Każda runda składa się z następujących etapów:
  - Mieszanie kluczy szyfrujących

  - Transformacja z użyciem S-Boxów. Każdy S-Box Serpenta jest permutacją 4-bajtową. Zdefiniowano 8 różnych S-Boxów, z których każdy jest używany w 4 rundach. W ramach jednej rundy używany jest jeden konkretny S-Box. Kolejność wykonywania S-Boxów określona jest ich numerem porządkowym $S_0, \ldots, S_7$. Specyfikacja wszystkich ośmiu S-Boxów znajduje się w załączniku A.5 do [2]
  - Transformacja liniowa LT, zdefiniowana dokładnie w załączniku A.3 do [2], jest stosowana w każdej rundzie z wyjątkiem ostatniej. W ostatniej rundzie wykonywana jest dodatkowa operacja mieszania kluczy.

- Permutacja końcowa FP będąca odwrotnością transformacji IP.

Odszyfrowanie szyfrogramu jest analogicznym procesem, zachodzą jednak następujące zmiany:

- S-Boxy są stosowane w odwrotnym porządku.
- Stosuje się odwrotną transformację liniową LT^{-1}, zdefiniowaną w załączniku A.4 do [2].

Serpent jest szyfrem blokowym i jako taki może funkcjonować w kilku trybach:

- Tryb elektronicznej książki kodowej (ECB) - wiadomość dzielona jest na bloki i każdy blok szyfrowany jest niezależnie. Ten tryb szyfrowania nie zapewnia współcześnie bezpieczeństwa, ponieważ zachowuje powtarzające się wzorce w danych.
- Tryb wiązania bloków zaszyfrowanych (CBC) - wiadomość jawna bloku B_i jest poddawana operacji XOR z szyfrogramem z poprzedniego bloku B_{i-1} i dopiero szyfrowana blokowo. Do pierwszego bloku B_0 dodaje się wektor inicjalizacyjny IV. Ten tryb zapewnia znaczną entropię wynikowego szyfrogramu i gwarantuje znaczne bezpieczeństwo.

W ramach projektu zaimplementowano algorytm Serpent w obu wymienionych powyżej trybach.

3 Architektura aplikacji - protokół komunikacyjny

3.1 Wstęp

Aplikacja zaprojektowana jest w architekturze klient - serwer. Na serwerze znajdują się wprowadzone przez administratora konta użytkowników uprawionych do korzystania z systemu. Serwer nasłuchuje przychodzących połączeń; po nawiązaniu sesji TCP następuje uzgodnienie klucza sesyjnego przy użyciu protokołu Diffiego - Hellmana. Klucz ten wykorzystywany jest do szyfrowania każdego

pakietu przesyłanego pomiędzy klientem a serwerem, w szczególności danych logowania. Po nawiązaniu bezpiecznego połączenia następuje autoryzacja i - o ile cały proces przebiegnie pomyślnie - serwer zezwala na zarządzanie danymi należącymi do klienta.

3.2 Protokół sieciowy programu

Istotną częścią systemu jest protokół sieciowy używany do komunikacji pomiędzy serwerem a klientem. Opiszemy teraz strukturę tego protokołu w szczegółach.

3.2.1 Logowanie użytkownika

Klient najpierw wysyła swoją nazwę użytkownika. Jeżeli serwer posiada takiego użytkownika w systemie, następuje nawiązanie szyfrowanego połączenia poprzez zastosowanie protokołu Diffiego - Hellmana, a następnie potwierdzenie hasła. Jeżeli takiego użytkownika nie ma w systemie, następuje odmowa połączenia. Jeżeli hasło okaże się nieprawidłowe, również następuje odmowa połączenia.

1. Pakiet REQ-0 : wysyłany przez klienta w celu zalogowania się do systemu.
   - REQ-TYPE = 0x00 (1 B)
   - USERNAME (64 B) - 64 znaki ASCII.
2. Pakiet LOGIN-STATUS : serwer wysyła go w celu potwierdzenia poprawności loginu.
   - LOGIN-FLAG: flaga jest równa 0x00 jeśli logowanie przebiegło poprawnie, lub 0xFF jeśli logowanie nie powiodło się. Protokół znajduje się nad TCP, zakładamy więc, że odebrane dane są zawsze poprawne i nie może wystąpić inna wartosć flagi jak 0x00 | 0xFF.
3. Pakiet DH-1 : Jeśli logowanie powiodło się, serwer losuje liczbę a i odpowiada klientowi swoim sekretem $g^a \ mod \ p$.
   - DH-SERVER-SECRET (64 B)
4. Pakiet DH-2 : klient losuje liczbę b i odpowiada serwerowi swoim sekretem $g^b \ mod \ p$.
   - DH-CLIENT-SECRET (64B)

   W tym momencie serwer i klient posiadają wspólny klucz szyfrowania symetrycznego, równy $g^{ab} = g^{ba}$. Pozostałe komunikaty w ramach sesji są szyfrowane serpentem z użyciem tego klucza.
5. Pakiet PASWD-1 : klient wysyła serwerowi wartość funkcji SHA-512 z hasła.
   - PASSWD (64 B)

6. Pakiet LOGIN-STATUS: serwer wysyła go w celu potwierdzenia poprawności hasła. Struktura identyczna jak w 2.

3.2.2 Listowanie zawartości katalogu użytkownika

Klient wysyła zapytanie o listę plików w swoim katalogu. Serwer wysyła odpowiedź.

1. Pakiet REQ-1 : klient wysyła zapytanie o listę plików w swoim katalogu oraz port, na którym jest gotów odebrać listę.
   - REQ-TYPE = 0x01 (1 B)
   - PORT (2 B)
2. Pakiet LEN-1 : serwer wysyła długość listy plików w bajtach.
   - LENGTH (8 B)

   W tym momencie serwer nawiązuje nową sesję na porcie *PORT* klienta i wysyła tam *LENGTH* bajtów danych, zawierających JSON-a z listą plików. Każdy plik to obiekt JSON-a zawierający pola:
   - NAME
   - SIZE
   - LAST-MODIFICATION

3.2.3 Dodawanie pliku na serwer

Klient wysyła prośbę o wysłanie pliku na serwer. Serwer sprawdza, czy plik o takiej nazwie znajduje się już w katalogu użytkownika. Jeśli nie, wysyła klientowi pozwolenie na wysyłanie pliku wraz z numerem portu, na którym będzie przebiegać wysyłanie. Po otrzymaniu pozwolenia od serwera, klient rozpoczyna wysyłanie pliku. Jeśli plik o takiej nazwie istnieje już na serwerze, serwer odmawia przyjęcia pliku.

1. Pakiet REQ-2 : klient wysyła prośbę o pozwolenie na dodanie pliku do serwera.
   - REQ-TYPE = 0x02 (1 B)
   - NAME (64 B) - 64 znaki ASCII
2. Pakiet RES-2 : serwer wysyła zgodę wraz z numerem portu, na którym serwer jest gotów odebrać plik, lub brak zgody.
   - PERM-FLAG (1 B) - równe 0x00, jeśli plik może zostać wysłany na serwer, lub 0xFF jeśli plik nie może zostać wysłany na serwer.
   - PORT (2 B)

3. Pakiet LEN-1: klient wysyła rozmiar wysyłanego pliku. Struktura identyczna jak w 2.

   W tym momencie klient nawiązuje nową sesję TCP na porcie $PORT$ serwera i wysyła tam $LENGTH$ bajtów danych zawierających dodawany plik.

3.2.4 Pobieranie skrótu pliku z serwera

Klient wysyła prośbę o wysłanie skrótu (SHA-256) pliku o zadanej nazwie. Serwer odsyła żądany skrót, lub 0, gdy takiego pliku nie ma na serwerze.

1. Pakiet REQ-3 : klient wysyła prośbę o skrót zadanego pliku.
   - REQ-TYPE = 0x03 (1 B)
   - NAME (64 B)
2. Pakiet RES-3 : serwer odpowiada skrótem pliku, o ile plik istnieje.
   - EXISTS-FLAG (1 B) - równe 0x00, jeśli plik znajduje się na serwerze, lub 0xFF jeśli pliku brak
   - HASH (64 B) - równe skrótowi pliku, jeśli plik znajduje się na serwerze, lub 0x0...0 jeśli pliku brak.

3.2.5 Pobieranie pliku z serwera

Klient wysyła prośbę o pobranie pliku z serwera. Serwer sprawdza, czy plik o takiej nazwie znajduje się w katalogu użytkownika. Jeśli tak, wysyła klientowi pozwolenie na pobranie pliku wraz z numerem portu, na którym będzie przebiegać transmisja. Po otrzymaniu pozwolenia od serwera, rozpoczyna się pobieranie pliku.

1. Pakiet REQ-4 : klient prosi o możliwość pobrania pliku z serwera.
   - REQ-TYPE = 0x04 (1 B)
   - NAME (64 B)
2. Pakiet RES-4 : serwer odpowiada zgodą, o ile plik istnieje oraz długością przesyłanego pliku.
   - EXISTS-FLAG (1 B) - równe 0x00, jeśli plik istnieje, lub 0xFF jeśli pliku brak.
   - LENGTH (8 B)
3. Pakiet PORT-1 : klient przesyła serwerowi port, na którym jest gotów odebrać plik.
   - PORT (2 B)

   W tym momencie serwer nawiązuje nową sesję TCP na porcie $PORT$ klienta i wysyła tam $LENGTH$ bajtów danych zawierających pobierany plik.

3.2.6 Usuwanie pliku

Klient wysyła prośbę o wysłanie wskazanego pliku z serwera. Serwer odpowiada potwierdzeniem, jeśli plik istnieje i został usunięty.

1. Pakiet REQ-5: klient wysyła prośbę o usunięcie wskazanego pliku.
   - REQ-TYPE = 0x05 (1 B)
   - NAME (64 B)
2. Pakiet RES-5: serwer wysyła potwierdzenie usunięcia pliku lub stwierdza, że pliku nie było na serwerze.
   - DELETE-FLAG (1 B) - równe 0x00, jeśli plik został poprawnie usunięty, lub 0xFF jeśli pliku o zadanej nazwie nie było na serwerze.

4 Stworzona aplikacja

4.1 Serwer

Serwer jest napisany w języku Python 3.6. Udostępnia on wszystkie funkcje, które zostały wymienione w rozdziale o protokołach sieciowych. Korzysta on ze specjalnej odmiany socketservera, która dla każdego nowego połączenia (tj. połączenia z nowego gniazda) tworzy nowy wątek, w którym obsługuje wszystkie zapytania, aż do wiadomości LOG-OUT (lub ewentualnie zamknięcia gniazda). To oznacza, że dla każdego nowego klienta tworzy się nowy wątek do jego obsługi. Na samym początku serwer wykonuje procedurę *auth*. Wykorzystuje on na początku moduł *DiffieHellman.py*, który zawiera klasę *DiffieHellman*. Działanie Diffie Hellmana oraz protokół komunikacyjny zostały wcześniej opisane. Wynikiem komunikacji klienta z serwerem jest nowy klucz do komunikacji. Klucz ten zasila koleny moduł *SerpentCipher.py*. Jest on wymienny z innymi modułami szyfrującymi. Sposób działania serpenta został omówiony w poprzenich rozdziałach. Moduł serpenta w serwerze korzysta z implementacji zamieszczonej na oficjalnej stronie serpenta. Jest ona napisana w języku C. Jako, że serwer działa pod systemem operacyjnym Windows 10 x64, została ona (implementacja) przerobiona do postaci kompilowalnej do pliku *Serpent.dll*. Następnie moduł wczytuje bibliotekę *Serpent.dll* dzięki bibliotece *CTypes*. Biblioteka ta wymagała, aby wszystkie funkcje, które będą używane, były zdefiniowane na poziomie języka Python. Po przeportowaniu można było używać funkcji z *Serpent.dll* jak natywnych funkcji z Pythona, które oczywiście zwracają typy CTypes (które także trzeba zrzutować). Została zaimplementowana minimalna opcja szyfrowania / deszyfrowania, czyli wszystko wykonuje się w wątku klienta. Kolejną czynnością w funkcji *auth* jest sprawdzenie, czy przesłany login oraz hasło są prawidłowe. Używa do tego modułu *Users.py*. Moduł ten dodatkowo udostępnia możliwość manipulowania użytkownikami. Hasła użytkowników są przechowywane w SHA3-512. Po poprawnym zalogowaniu się, wątek serwera, do którego poprawnie zalogował się klient, czeka na polecenia:

- pobranie listy plików,
- upload pliku,
- download pliku,
- usunięcie pliku,
- wylogowanie się.

Oprócz ostatiej opcji wszystkie dotyczą manipulacji na plikach użytkownikach. Wszystkie manipulacje zostały zawarte w module *UserFS.py*. Zarządza ona wirtualnym systemem plików użytkowników. Każdy użytkownik posiada własny folder o nazwie odpowiadającej swojej nazwie użytkownika, który znajduje się w nadfolderze *virtual-fs*.

Kolejnym ważnym modułem w serwerze jest moduł komunikacji *Protocol.py*. Zawiera on klasę *Protocol*, która jest inicjalizowana z argumentem modułu szyfrującego (tu: *SerpentCipher* albo *SerpentCipherClassicalString*). Jest on ważny, gdyż każda wiadomość jest szyfrowana i odszyfrowana w tym module. Protokół opakowuje całą komunikację między klientem a serwerem - w obie strony. Nie będzie w tej dokumentacji opisana każda metoda, gdyż są one opisane w protokole komunikacyjnym. Zamiast tego zostanie tutaj opisana metoda dodawania wszystkich typów wiadomości. Przede wszystkim każda wiadomość z protokołu jest zaimplementowana jako funkcja w stronę: zmienne w pythonie -¿ postać binarna (przyrostek *encode*) oraz w odwrotną stronę: postać binarna -¿ zmienne w pythonie (przyrostek *decode*). Każda z takich metod jest statyczna, gdyż nie potrzebuje nic z klasy (modułu szyfrującego). Oprócz pierwszych paru wiadomości z *auth* każda wiadomość może być następnie zaszyfrowana oraz odszyfrowana. Jest to zaimplementowane w postaci obudowy istniejących funkcji (*encode* i *decode*) za pomocą nowych funkcji klasy *Protocol* (tym razem już nie statycznych, bo używają modułu szyfrującego) o nazwie takiej, jak funkcja, którą obudowuje odpowiednio z przedrostkiem *encrypt* lub *decrypt*.

4.2 Klient

Aplikacja kliencka pozwala na zarządzanie plikami użytkownika zamieszczonymi na serwerze. Zgodnie z założeniami projektu, powinna działać w systemie Android. W celu zbudowania projektu wykorzystaliśmy jednak istniejącą implementację algorytmu Serpent w języku C, kompilowaną do biblioteki dynamicznej. Niestety nie udało się nam przeportować tej implementacji na bibliotekę dynamiczną poprawnie ładującą się na Androidzie z poziomu języka Java, ze względu na problemy z architekturą procesora (urządzenia mobilne, w przeciwieństwie do komputerów tradycyjnych, używają w znakomitej większości procesorów opartych o różne wersje architektury ARM). Sytuacja ta zmusiła nas do wykonania graficznego interfejsu użytkownika dla systemu Windows.

Kod aplikacji znajduje się na serwerze GitHub pod adresem `https://github.com/5james/Simple-Cloud` . Na tej samej stronie możliwe jest pobranie insta-

latora aplikacji w formie skompresowanego archiwum ZIP. Archiwum należy wypakować, a aplikację można uruchomić poprzez plik mkoi.exe.

Aplikacja kliencka łączy się zdalnym serwerem. Dane serwera, do którego będziemy się łączyć - adres IP oraz port - zapisane są w pliku server.conf.

Aplikacja posiada dwa główne okna: okno logowania i okno zarządzania serwerem. W oknie logowania wprowadza się nazwę użytkownika i hasło. Jeżeli logowanie przebiegnie poprawnie, aplikacja przełącza się w tryb zarządzania plikami. W tym trybie widoczna jest aktualna lista przechowywanych na serwerze plików (potencjalnie pusta) oraz przyciski umożliwiające wykonanie różnych akcji:

- Upload - umożliwia dodanie nowego pliku na serwer. Dodawany jest plik znajdujący się lokalnie na komputerze klienta.
- Refresh - umożliwia odświeżenie listy plików.
- Sign out - umożliwia wylogowanie.
- Download - umożliwia pobranie wskazanego pliku z listy na lokalną maszynę. Plik pobierany jest do katalogu z aplikacją.
- Delete - przycisk usuwa wskazany plik z serwera.
- Hash - umożliwia wyświetlenie skrótu wskazanego pliku, np. w celu weryfikacji poprawności pobierania.

Aplikacja kliencka cechuje się prostotą i łatwością użytkowania.

4.3 Testy

Testy aplikacji realizowane były w formie bezpośredniej interakcji użytkownika końcowego z aplikacją. Testy przebiegły pomyślnie, a aplikacja wykonuje poprawnie wszystkie przewidziane jej funkcje:

- Logowanie i wylogowanie użytkownika
- Dodawanie i pobieranie pliku z serwera
- Usuwanie pliku
- Generowanie skrótu SHA-256 pliku znajdującego się na serwerze
- Poprawna obsługa wielu użytkowników

W razie niepowodzenia, aplikacja wyświetla podstawowe komunikaty o błędzie i jego przyczynie. Godny uwagi jest jednak fakt, że działanie aplikacji nie zawsze jest szybkie - zwłaszcza szyfrowanie i deszyfrowanie większych plików zajmuje pewien czas, co spowalnia działanie aplikacji. W tyj sprawie z pewnością jest jeszcze pole do dalszych optymalizacji.

Literatura

[1] Schneier Bruce. *Kryptografia dla praktyków: protokoły, algorytmy i programy źródłowe w języku C.* Wydawnictwa Naukowo-Techniczne, Warszawa; 2002.

[2] Anderson Eli; Knudsen Lars. A proposal for the advanced encryption standard.. *University of Cambridge.* 1998;.