Protokół SKID3

Dominik Kania WCY18KY1S1

1 Wstęp

Poniższy raport został przygotowany na zaliczenie laboratoriów z przedmiotu Protokoły Kryptograficzne. Przedstawiam w nim opis, implementację i weryfikację formalną protokołu SKID3.

2 Ogólny opis protokołu

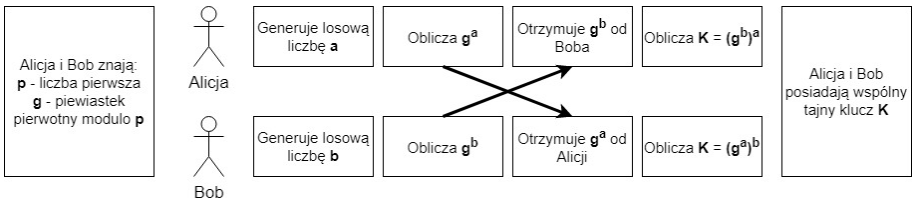
2.1 Opis protokołu

Protokoły z rodziny SKID to symetryczne protokoły identyfikacji utworzone na potrzeby europejskiego projektu RIPE (ang. „RACE Integrity Primitives Evaluation") w latach 1988-1992 jako jedne z prymitywów kryptograficznych wchodzących w skład technik wspierających komunikację IBC (ang. „Integrated Broandband Communication"). Protokoły te wykorzystują jednorazowe funkcje skrótu MAC (ang. „Message authentication code") [1].

Protokół SKID3 umożliwia obustronne uwierzytelnianie użytkowników, podczas gdy SKID2 umożliwia uwierzytelnianie tylko w jedną stronę.

2.2 Działanie protokołu

Załóżmy, że Alicja i Bob chcą się ze sobą skontaktować i chcą mieć pewność, że komunikują się właśnie ze sobą. Warunkiem początkowym prawidłowego działania protokołu jest posiadanie przez Alicję i Boba wspólnego tajnego sekretu K, umożliwiającego wykorzystanie szyfrowanej jednokierunkowej funkcji skrótu. Wspólny sekret można wygenerować np. za pomocą protokołu Diffiego-Hellmana przedstawionego na rys. 1. Wykorzystuje on losową liczbę pierwszą p i pierwiatek pierwotny modulo p[1].



Rysunek 1: Schemat protokołu wymiany klucza Diffiego-Hellmana

Pierwszym etapem protokołu SKID3 jest wygenerowanie losowych liczb przez Alicję i Boba:

[1] $\forall x^k \in factor(p-1) : g^x \% p! = 1$

(I) Alicja generuje 64 bitową liczbę losową R_A. Następnie wysyła tę liczbę do Boba.

(II) Bob generuje 64 bitową liczbę losową R_B. Następnie wysyła Alicji:

$R_B, H_K(R_A, R_B, B)$,

gdzie H_K to jednokierunkowa funkcja skrótu z kluczem K, zaś B to nazwa użytkownika Boba. RIPE sugeruje użycie funkcji skrótu RIPEMD-128 [1].

(III) Alicja oblicza swoją wartość $H_K(R_A, R_B, B)$ i porównuje ją z hashem otrzymanym od Boba. Jeżeli te wartości są identyczne, Alicja ma pewność, że komunikuje się z Bobem.

Punkty (I) do (III) są identyczne w protokole SKID2. Aby dodatkowo Bob miał pewność, że komunikuje się z Alicją, kolejne kroki są następujące:

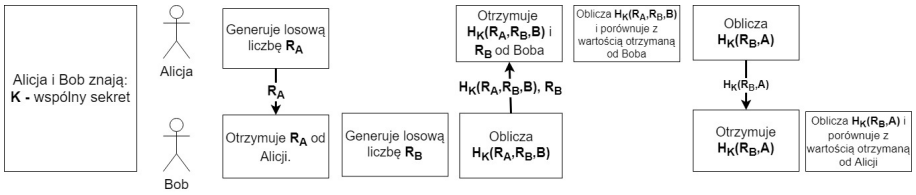
(IV) Alicja wysyła Bobowi:

$H_K(R_B, A)$,

gdzie A to nazwa użytkownika Alicji.

(V) Bob oblicza swoją wartość $H_K(R_B, A)$ i porównuje ją z hashem uzyskanym od Alicji. Jeżeli te wartości są identyczne, Bob ma pewność, że komunikuje się z Alicją.

Reprezentacja graficzna działania protokołu została zaprezentowana na rys. 2.



Rysunek 2: Schemat protokołu SKID3

2.3 Bezpieczeństwo i modyfikacje

Bezpieczeństwo protokołu oparte jest na bezpieczeństwie użytego MAC-a. Kluczowe jest powiązanie losowych liczb Alicji i Boba w funkcji $H_K(R_A, R_B, B)$ w punkcie (III). Neguje to możliwość ataku poprzez sesję równoległą (ang. parallel session attack).

Odgórnie ustalona kolejność parametrów funkcji $H_K(R_A, R_B, B)$ neguje możliwość ataku przez odbicie (ang. reflection attack).

Protokół jest odporny na atak man-in-the-middle pod warunkiem, że atakujący nie zna wspólnego sekretu K. Ważnym jest zatem jak najbezpieczniejsze nabycie tajnego sekretu przez osoby kontaktujące się i jego jednorazowe użycie tylko do autoryzacji.

Zgodnie z wytycznymi twórców, protokół SKID powinien wykorzystywać funkcję skrótu RIPEMD-128. Nie jest to jednak algorytm popularny. Szybko zaczęto używać nowszych algorytmów z rodziny SHA i MD5. Niska popularność dodatkowo sprawiła, że RIPEMD nie jest zbadany tak dogłębnie jak częściej używane algorytmy.

W 2004 roku chińscy kryptolodzy wykryli kolizje w funkcjach RIPEMD oraz MD5 [2]. Dlatego w celu zwiększenia bezpieczeństwa w implementacji jako funkcję skrótu użyję SHA-256.

2.4 Przykład ataku

Załóżmy, że Maria zna sekret K użyty przez Alicję i Boba w funkcji skrótu oraz chce podszyć się pod Boba. Wówczas może wystąpić następująca sytuacja:

1) $A \rightarrow M(B) : R_A$

2) $E(B) \rightarrow A : R_{M(B)}, H_K(R_A, R_{M(B)}, B)$

Alicja wysyła swoją losową liczbę R_A do Boba. Maria przechwytuje ten sygnał i podając się za Boba odsyła Alicji swoją losową liczbę i MAC wyznaczony przez wykradziony klucz, losową liczbę Alicji i nazwę użytownika Boba.

Alicja otrzymuje tą wiadomość i po obliczeniu swojej funkcji skrótu jest przekonana, że komunikuje się z Bobem.

3 Implementacja

3.1 Kod zródłowy kluczowych funkcji

Pełny kod źródłowy znajduje się w pliku **SKID3.py**.

```
def generateRandom():
    r = sage.misc.prandom.getrandbits(64)
    r = bin(r)[2:]
    return r


def mac1(r_a,r_b, name, k):
    ipad = 0x5c
    opad = 0x36
    nick = ''.join(format(ord(i), 'b') for i in name)
    m = str(r_a)+str(r_b)+str(nick)
    j = hashlib.new('sha256')
    j.update(m)

    x = int(k)^int(ipad)
    y = str(x)+str(j.hexdigest())
```

```
    l = hashlib.new('sha256')
    l.update(str(y))

    w = str(int(k)^int(opad))+str(l.hexdigest())
    h = hashlib.new('sha256')
    h.update(w)
    return h

def mac2(r_a, name, k):
    ipad = 0x5c
    opad = 0x36
    nick = ''.join(format(ord(i), 'b') for i in name)
    m = str(r_a)+str(nick)
    j = hashlib.new('sha256')
    j.update(m)

    x = int(k)^int(ipad)
    y = str(x)+str(j.hexdigest())

    l = hashlib.new('sha256')
    l.update(str(y))

    w = str(int(k)^int(opad))+str(l.hexdigest())+str(j.hexdigest())
    h = hashlib.new('sha256')
    h.update(w)
    return h
```

3.2 Opis użytych funkcji

1. Pierwszym krokiem protokołu podjętym zarówno przez Alicję jak i Boba jest wygenerowanie liczby losowej, do czego stosują funkcję *generateRandom*(k), a jako parametr wejściowy wprowadzają liczbę bitów k losowanej liczby:

   ```
   def generateRandom(k):
       r = sage.misc.prandom.getrandbits(k)
       r = bin(r_a)[2:]
       return r
   ```

   Funkcja tworzy losowy ciąg bitów o długości podanej na wejściu i zwraca go.

2. Funkcja *mac*1() zwraca wartość skrótu $H_K(R_A, R_B, B)$ obliczanego zarówno przez Alicję jak i Boba w celu weryfikacji. Na wejściu przyjmuje liczbę losową Alicji, liczbę losową Boba i nazwę użytkownika Boba oraz całość jest zabezpieczona przez wcześniej ustalony wspólny tajny klucz:

   ```
   def mac1(r_a,r_b, name, k):
   ```

```
    ipad = 0x5c
    opad = 0x36
    nick = ''.join(format(ord(i), 'b') for i in name)
    m = str(r_a)+str(r_b)+str(nick)
    j = hashlib.new('sha256')
    j.update(m)

    x = int(k)^int(ipad)
    y = str(x)+str(j.hexdigest())

    l = hashlib.new('sha256')
    l.update(str(y))

    w = str(int(k)^int(opad))+str(l.hexdigest())
    h = hashlib.new('sha256')
    h.update(w)
    return h
```

Funkcja wpierw skraca liczbę losową Alicji, liczbę losową Boba i nazwę użytkownika Alicji. Następnie obliczony zostaje MAC zgodnie z definicją z dokumentu RFC 2104[2]:

$$H(K, m) = H((K \oplus opad)||H((K \oplus ipad)||m)$$

gdzie
H - kryptograficzna funkcja skrótu SHA-256
m - skrót SHA-256 liczby losowej Alicji, liczby losowej Boba i nazwy użytkownika Alicji
K - wspólny klucz tajny Alicji i Boba
$\|$ - konkatenacja
$\oplus$ - działanie XOR
$opad, ipad$ - wartości dopełniające

który zostaje przez funkcję zwrócony.

3. Funkcja $mac2()$ nie różni się znacząco od funkcji $mac1()$. Różni się tym, że zwraca wartość skrótu $H_K(R_B, A)$, czyli uwzględnia tylko wspólny klucz, liczbę losową jednego z użytkowników oraz nazwę drugiego użytkownika.

```
def mac2(r_a, name, k):
    ipad = 0x5c
    opad = 0x36
    nick = ''.join(format(ord(i), 'b') for i in name)
    m = str(r_a)+str(nick)
    j = hashlib.new('sha256')
```

[2] https://tools.ietf.org/html/rfc2104

```
    j.update(m)

    x = int(k)^int(ipad)
    y = str(x)+str(j.hexdigest())

    l = hashlib.new('sha256')
    l.update(str(y))

    w = str(int(k)^int(opad))+str(l.hexdigest())+str(j.hexdigest())
    h = hashlib.new('sha256')
    h.update(w)
    return h
```

3.3 Weryfikacja poprawności

Do testów działania implementacji wykorzystałem środowisko SageMath dostępne na platformie Cocalc[3]. Program zmodyfikowałem celem prostszego wykonania dużej ilości testów. Kod testowanego programu jest następujący (kod źródłowy jest także dostępny w pliku **SKID3_testy.py**):

```
import hashlib
import time
def generateRandom():
    r = sage.misc.prandom.getrandbits(64)
    r = bin(r)[2:]
    return r

def mac1(r_a,r_b, name, k):
    ipad = 0x5c
    opad = 0x36
    nick = ''.join(format(ord(i), 'b') for i in name)
    m = str(r_a)+str(r_b)+str(nick)
    j = hashlib.new('sha256')
    j.update(m)

    x = int(k)^int(ipad)
    l = hashlib.new('sha256')
    l.update(str(int(k)^int(ipad)))

    w = str(int(k)^int(opad))+str(l.hexdigest())+str(j.hexdigest())
    h = hashlib.new('sha256')
    h.update(w)
    return h
```

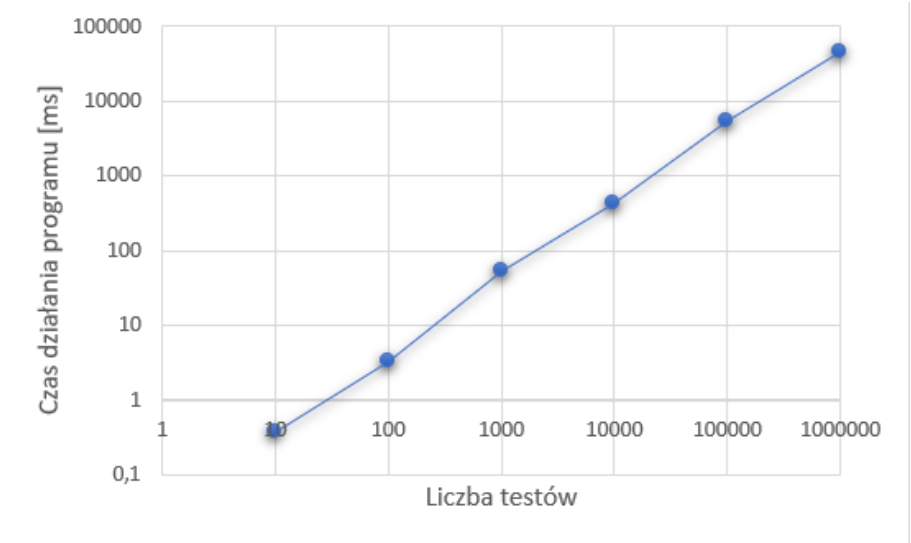
[3] https://cocalc.com

```
def mac2(r_a, name, k):
    ipad = 0x5c
    opad = 0x36
    nick = ''.join(format(ord(i), 'b') for i in name)
    m = str(r_a)+str(nick)
    j = hashlib.new('sha256')
    j.update(m)

    l = hashlib.new('sha256')
    l.update(str(int(k)^int(ipad)))

    w = str(int(k)^int(opad))+str(l.hexdigest())+str(j.hexdigest())
    h = hashlib.new('sha256')
    h.update(w)
    return h

def testy(l):
        s = 0
        p = 0
        for x in xrange(l):
            k = sage.misc.prandom.getrandbits(64)
            k = bin(k)[2:]
            r_a = generateRandom()
            r_b = generateRandom()

            H1 = mac1(r_a,r_b,str(B),k)
            H2 = mac1(r_a,r_b,str(B),k)

            if(H1.hexdigest() == H2.hexdigest()):
                H3 = mac2(r_b, str(A),k)
                H4 = mac2(r_b, str(A),k)

                if(H3.hexdigest() == H4.hexdigest()):
                    s = s + 1

            else:
                p = p + 1

        print("Wykonano " + str(l) + " testow dzialania.")
        print("Udane uwierzytelniania: " + str(s))
        print("Nieudane uwierzytelniania: "+ str(p))


A = raw_input("Podaj nazwe pierwszego uzytkownika:")
B = raw_input("Podaj nazwe drugiego uzytkownika:")
p = raw_input("Ile serii pomiarow chcesz wykonac?")
```

```
l = raw_input("Ile testow chcesz wykonywac w jednej serii?")
p = int(p)
l = int(l)
def pomiar(l,p):
    while(p > 0):
        start_time = time.time()
        testy(int(l))
        print("Czas dzialania programu: %s sekund." % (time.time() - start_time)
        print("\n")
        p = p - 1

pomiar(l,p)
```

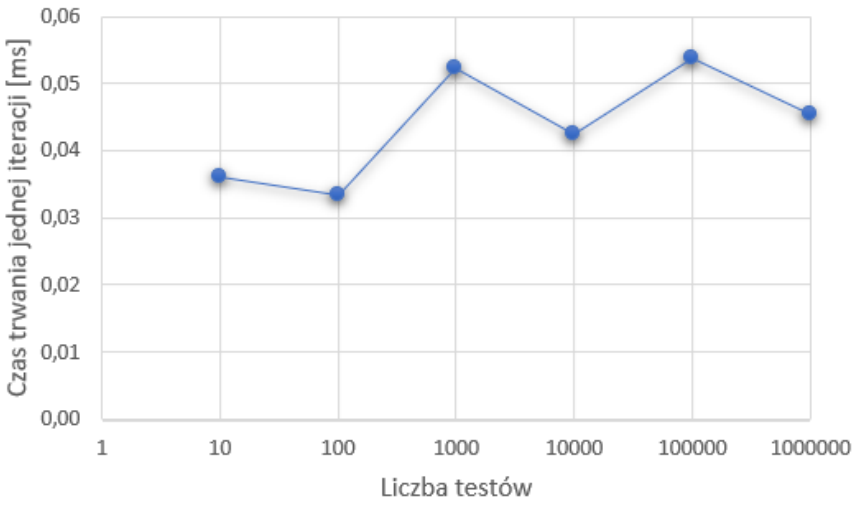
Program w pętli generuje losowe wartości R_A, R_B, K i oblicza skróty celem porównania i sprawdzenia poprawności obliczonych przez Alicję i Boba wartości. Program ten wykorzystuje bibliotekę *time* do pomiaru czasu działania.

Testy były przeprowadzane dla różnych wartości losowych liczb Alicji i Boba oraz wspólnego klucza. Każda seria pomiarów została wykonana 10 razy, a wyniki zostały uśrednione. 100% iteracji protokołu zakończyła się pomyślnie. Poniższy wykres 3 przedstawia czas działania programu w zależności od ilości przeprowadzonych testów. Widać z niego, że czas działania programu wzrasta liniowo względem liczby przeprowadzanych testów protokołu.



Rysunek 3: Zależność między czasem działania a ilością testów

Wykres 4 przedstawia średni czas trwania jednej iteracji protokołu w zależności o liczby przeprowadzanych testów.



Rysunek 4: Średni czas wykonania jednej iteracji protokołu

4 Weryfikacja formalna

Werifykację formalną protokołu wykonałem za pomocą programu Verfipal[4]. Program ten umożliwia zamodelowanie dowolnego protokołu kryptograficznego i poddanie go analizie pod kątem możliwych metod złamania. Program umożliwia dodatkowo wybór metody ataku - atakujący może być pasywny lub aktywny. W mojej weryfikacji sprawdzam, czy aktywny atakujący jest w stanie zmanipulować działanie protokołu i podszyć się pod Alicję lub Boba.

4.1 Kod źródłowy

Kod źródłowy wykorzystany do weryfikacji jest także dostępny w pliku **SKID3.vp**.

```
attacker[active]

principal Alice[
    knows private k
    knows public A
    generates ra
]

Alice -> Bob: ra
```

[4]darmowa wersja jest dostępna na stronie https://www.verifpal.com

```
principal Bob[
    knows private k
    knows public B
    generates rb

    sb = MAC(k,HASH(ra,rb,B))
]

Bob -> Alice: rb, sb

principal Alice[
    _ = ASSERT(MAC(k, HASH(ra,rb,B)), sb)?

    ha = MAC(k, HASH(rb,A))
]

Alice -> Bob: ha

principal Bob[
    db = HASH(rb,A)
    _ = ASSERT(MAC(k, db), ha)?
]


queries[
    authentication? Bob -> Alice: sb
    authentication? Alice -> Bob: ha
        authentication? Alice -> Bob: ra
        authentication? Bob -> Alice: rb
]
```

4.2 Analiza protokołu

Program wykonał 3794 analizy różnych działań atakującego. Program wykrył tylko jedną możliwość ataku w sytuacji, gdy atakujący nie zna tajnego klucza K.

$$M(A) \rightarrow B : R_{M(A)}$$

Atakująca Maria wysyła Bobowi podrobioną losową liczbę $R_{M(A)}$ podając się za Alicję. Skutkuje to obliczeniem przez Boba skrótu $H_K(R_{M(A)}, R_B, B)$ z podrobionej liczby $R_{M(A)}$. Jednak to działanie nie daje żadnych efektów, ponieważ tylko podrabia autentykację Boba względem Marii.

Pomimo braku skutecznych ataków wykrytych przez program nie można stwierdzić, że nie istnieją ataki na protokół. Verifpal jest programem eksperymentalnym i nie gwarantuje sprawdzenia wszystkich możliwości ataku.

5 Podsumowanie

Analiza protokołu SKID3 wskazuje na to, że jest to protokół bezpieczny i skuteczny. Umożliwia obustronne uwierzytelnianie użytków przy użyciu jednego klucza i jednego zbioru danych. Wymaga ona jednak znajomości przez obie strony wspólnego tajnego klucza, co może być trudne. Dodatkowo klucz ten powinien być wykorzystywany jednorazowo, co dodatkowo utrudnia działanie protokołu. Zatem każdym razem trzeba go wygenerować od nowa i obu stronom.

Bezpieczeństwo tego protokołu opiera się na bezpieczeństwie wykorzystanego MAC-a, a nie są one bez wad. Odkrycie kolizji funkcji RIPEMD oraz MD5 [2] poddało pod wątpliwość możliwość wykorzystania tych skrótów. Pomimo możliwości wykorzystywania lepszych funkcji skrótu i ich ciągłego rozwoju[5] zawsze będą one niedoskonałe. Niestety cechą każdej kryptograficznej funkcji skrótu jest to, że istnieje w niej możliwość wystąpienia kolizji.

Praca nad tym raportem umożliwiła mi rozszerzenie wiedzy zarówno o protokołach, jak i sposobach ich badań i analizy. Dodatkowo mogłem rozwinąć swoje umiejętności w wykorzystywaniu środowiska SageMath i języka LaTeX. Jestem przekonany, że te umiejętności przydadzą mi się w moich kolejnych projektach.

Literatura

[1] Bruce Schneier. *Applied Cryptography: Protocols, Algorthms, and Source Code in C.* (Angielski) *[Kryptografia dla praktyków: Protokoły, algorytmy i programy źródłowe w języku C]*, 1996

[2] Wang, Xiaoyun; Feng, Dengguo; Lai, Xuejia; Yu, Hongbo (2004-08-17). "Collisions Hash Functions MD4 MD5 RIPEMD HAVAL"

[3] https://www.infoq.com/news/2020/01/blake3-fast-crypto-hash/

[5] 9 stycznia 2020 roku na sympozjum Real World Crypto ogłoszono nową wersję funkcji z rodziny BLAKE - BLAKE3 [3]