PK projekt: DASS/SPX

Kamila Biernacka, Cezary Bąk

Prowadzący: mgr inż. Tomasz Mazurkiewicz

Marzec 2020

1 Ogólny opis protokołu

1.1 Geneza protokołu

DASS - Distributed Authentication Security Service – usługa uwierzytelniania klient - klient oparta na kluczu publicznym. Jej prototyp został użyty w protokole SPX, którego budowa i cele zbliżone są do protokołu Kerberos. Obsługuje on uwierzytelnianie użytkowników i podmiotów sieciowych, dystrybucję kluczy, ochronę danych w tranzycie danych, pojedyncze logowania, przekazywanie uprawnień na podstawie tożsamości, skalowalność do bardzo dużego środowiska. Pierwsze publikacje DASS/SPX pojawiły się na początku lat 90. XX w. DASS opisuje architekturę, natomiast Sphinx(SPX) odnosi się do implementacji protokołu.

1.2 Opis protokołu

1. Alice i Bob generują klucze prywatne oraz publiczne i wysyłają publiczne do trzeciej zaufanej strony.

2. Trzecia zaufana strona podpisuje otrzymane kopie kluczy publicznych. Podpis składa się z wykonania na wiadomości funkcji skrótu SHA-256, a następnie zaszyfrowania skrótu algorytmem RSA.

3. Alice wysyła do trzeciej zaufanej strony wiadomość zawierającą nazwę Boba.

4. Trzecia zaufana strona przesyła do Alice klucz publiczny Boba podpisany kluczem prywatnym trzeciej zaufanej strony.

5. Alice dokonuje weryfikacji podpisu trzeciej zaufanej strony, sprawdzając, czy klucz, który otrzymała jest aktualnym kluczem publicznym Boba. Dalej generuje losowy klucz tajny i losowe klucz publiczny oraz klucz prywatny, szyfruje czas, używając do tego klucza tajnego, dokonuje podpisu okresu ważności klucza, swojego identyfikatora oraz losowego klucza przy użyciu swojego klucza prywatnego, dokonuje szyfrowania klucza tajnego za pomocą klucza publicznego Boba i podpisuje przy pomocy swojego losowego klucza prywatnego. Całość wysyła do Boba. Szyfrowanie asymetryczne odbywa się przy użyciu algorytmu RSA, do symetrycznego natomiast używany jest AES.

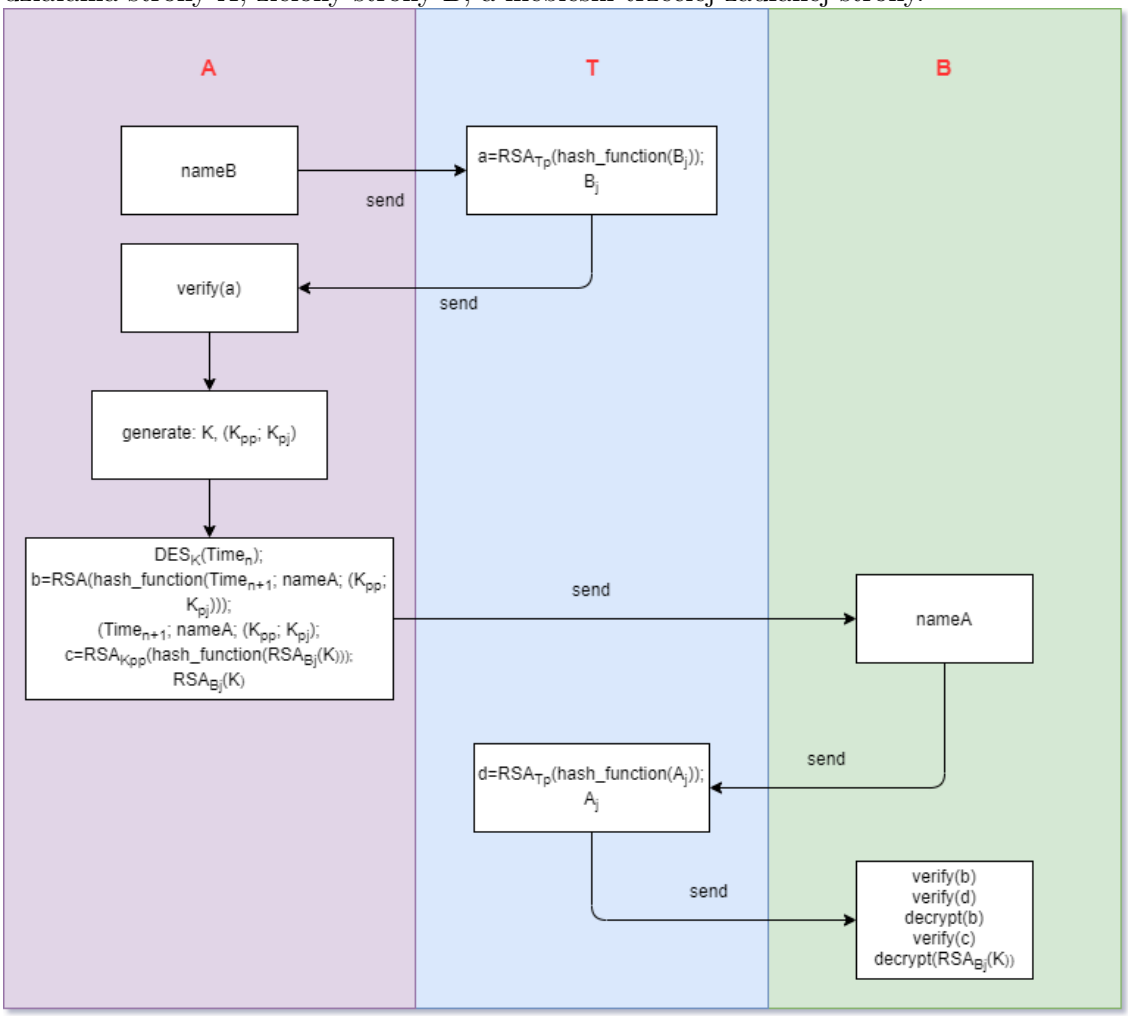
6. Bob przesyła do trzeciej zaufanej strony wiadomość zawierającą nazwę Alice.

7. Trzecia zaufana strona wysyła do Boba klucz publiczny Alice, który podpisała za pomocą swojego klucza prywatnego.

8. Bob sprawdza podpis trzeciej zaufanej strony, by wiedzieć, że otrzymał aktualny klucz publiczny Alice, dokonuje sprawdzenia podpisu Alice i odtwarza jej losowy klucz prywatny, dokonuje sprawdzenia podpisu i odtwarza losowy klucz tajny Alice za pomocą swojego klucza prywatnego.

1.3 Działanie algorytmu

Na rysunku przedstawiono działanie algorytmu DASS/SPX. Kolor różowy reprezentuje działania strony A, zielony strony B, a niebieski trzeciej zaufanej strony.



Algorithm 1 to pseudokod rozpoczynający działanie protokołu. Został opisany w formie synchronicznej (liniowej), poszczególne komunikacje z innymi maszynami (B lub T) wykonują funkcje „*send_to_T*" i „*send_to_B*". Program czeka na odpowiedź pozostałych hostów, następnie kontynuuje wykonanie protokołu.
Algorithm 1 reprezentuje kroki 1, 3 i 5 z opisu protokołu.

Algorithm 1 Uwierzytelnianie użytkownika A względem B

// Użytkownik generuje parę kluczy (publiczny, prywatny)
$\mathbf{p} = random\_prime(2^{512} - 1, False, 2^{511})$
$\mathbf{q} = random\_prime(2^{512} - 1, False, 2^{511})$
$\mathbf{public\_key}, \mathbf{private\_key} = generate\_keypair(\mathbf{p}, \mathbf{q})$
// Teraz wysyłamy nasz klucz do serwera który go podpisze, przechowa oraz odeśle swój klucz publiczny
$\mathbf{T\_public\_key} = send\_to\_T(\mathbf{A\_name}, \mathbf{public\_key})$
// Do T musimy wysłać nazwę użytkownika z którym chcemy się komunikować
$\mathbf{B\_signed\_public\_key}, \mathbf{B\_public\_key} = send\_to\_T(\mathbf{B\_name})$
// Otrzymaliśmy klucz publiczny użytkownika z którym chcemy się komunikować oraz jego podpis wykonany przez T sprawdzamy teraz jego poprawność
if $decrypt(\mathbf{T\_public\_key}, \mathbf{B\_signed\_public\_key})! = hash\_function(\mathbf{B\_public\_key})$ **then**
    // Tutaj obsługujemy błąd niepoprawnego klucza
end if
// Teraz generujemy klucze które posłużą do szyfrowania pakietu danych który przekażemy bezpośrednio do B
$\mathbf{K} = getrandbits(512)$
$\mathbf{p} = random\_prime(2^{512} - 1, False, 2^{511})$
$\mathbf{q} = random\_prime(2^{512} - 1, False, 2^{511})$
$\mathbf{K\_public\_key}, \mathbf{K\_private\_key} = generate\_keypair(\mathbf{p}, \mathbf{q})$
// Tworzymy znacznik czasu oraz okres ważności klucza który będzie określał czy nasza wiadomość jest aktualna
$\mathbf{T\_A} = datetime.now()$
// Jedna godzina
$\mathbf{L} = timedelta(hours = 1)$
// Szyfrujemy teraz czas oraz podpisujemy zestaw danych który wyślemy do B
$\mathbf{a} = aes.encrypt(\mathbf{T\_A}, \mathbf{K})$
$\mathbf{b} = (\mathbf{L}, \mathbf{A\_name}, \mathbf{K\_public\_key}, \mathbf{K\_private\_key})$
$\mathbf{b\_sign} = encrypt(\mathbf{private\_key}, hash\_function(\mathbf{b}))$
$\mathbf{c} = encrypt(\mathbf{B\_public\_key}, \mathbf{K}))$
$\mathbf{c\_sign} = encrypt(\mathbf{K\_private\_key}, hash\_function(\mathbf{c})$
// Teraz wysyłamy wszystko do B i czekamy na jego odpowiedź
$\mathbf{respond} = send\_to\_B((a, b, c))$

Algorithm 2 to pseudokod napisany w sposób funkcyjny. Zawiera metody, które są wywoływane w zależności od podanych przez użytkowników zmiennych.
Algorithm 2 reprezentuje kroki 2, 4 i 7 z opisu protokołu.

Algorithm 2 Metody które zarządzają serwerem

```
1: procedure INIT()
2:     // Serwer też posiada parę kluczy
3:     p = random_prime(2^512 − 1, False, 2^511)
4:     q = random_prime(2^512 − 1, False, 2^511)
5:     public_key, private_key = generate_keypair(p, q)
6: end procedure
7: procedure SET_PUBLIC_KEY(name, key)
8:     // Zapisuje nazwę użytkownika jego klucz publiczny oraz podpis w bazie
9:     users.append((name, key, encrypt(public_key, hash_function(key)))
10:    return public_key
11: end procedure
12: procedure SEND_PUBLIC_KEY_BY_NAME(name)
13:    // znajduje w bazie nazwę użytkownika oraz jego klucz
14:    return users.find(name=name)
15: end procedure
```

Algorithm 3 to część pseudokodu zawierająca program użytkowników A oraz B. Ten fragment służy jedynie do odpowiadania na próbę komunikacji i nie jest wykorzystywany do samodzielnej próby nawiązania połączenia.
Algorithm 3 reprezentuje kroki 1, 6 i 8 z opisu protokołu.

Algorithm 3 Uwierzytelnianie użytkownika A na maszynie B

1: **procedure** INIT()
2: // Zakładamy że ta procedura została już wcześniej wykonana
3: $\mathbf{p} = random\_prime(2^{512} - 1, False, 2^{511})$
4: $\mathbf{q} = random\_prime(2^{512} - 1, False, 2^{511})$
5: $\mathbf{public\_key}, \mathbf{private\_key} = generate\_keypair(\mathbf{p}, \mathbf{q})$
6: $\mathbf{T\_public\_key} = send\_to\_T(\mathbf{A\_name}, \mathbf{public\_key})$
7: **end procedure**
8: **procedure** RECEIVE_CONTACT(a, b, b_sign, c, c_sign)
9: $\mathbf{A\_signed\_public\_key}, \mathbf{A\_public\_key} = send\_public\_key\_by\_name(b[1])$
10: **if** $decrypt(\mathbf{T\_public\_key}, \mathbf{A\_signed\_public\_key})! = hash\_function(\mathbf{A\_public\_key})$ **then**
11: // Tutaj obsługujemy błąd niepoprawnego klucza
12: **return False**
13: **end if**
14: $\mathbf{K} = decrypt(\mathbf{private\_key}, \mathbf{c}))$
15: $\mathbf{T\_A} = decrypt(\mathbf{private\_key}, \mathbf{c}))$
16: **if** $datetime.now() > (T\_A + b[0])$ **then**
17: // Tutaj obsługujemy błąd nieaktywnego klucza
18: **return False**
19: **end if**
20: // Sprawdzamy poprawność podpisów
21: **if** $decrypt(\mathbf{A\_public\_key}, \mathbf{b\_sign})! = hash\_function(\mathbf{b})$ **then**
22: // Tutaj obsługujemy błąd niepoprawnego podpisu
23: **return False**
24: **end if**
25: **if** $decrypt(\mathbf{b}[\mathbf{2}], \mathbf{c\_sign})! = hash\_function(\mathbf{c})$ **then**
26: // Tutaj obsługujemy błąd niepoprawnego podpisu
27: **return False**
28: **end if**
29: **return True**
30: **end procedure**

1.4 Przykład użycia SPX

A, B - Użytkownicy
X - Serwer
Użytkownik A chce się zalogować jako użytkownik B na maszynie X.
B ustawił dostęp do logowania dla A na maszynie X na jego konto (B),
więc A wyśle token weryfikacyjny SPX do X. X rozpatrzy token od A i wykona weryfikację oraz prześle AKCEPTUJ lub ODRZUĆ. Jeżeli podobna weryfikacja miała już miejsce, X w wiadomości AKCEPTUJ załączy tę informacje.
Weryfikacja przez X polega na sprawdzeniu czy B pozwala na logowanie A na swoje konto po zakończeniu procesu weryfikacji.
Możliwe jest otrzymanie wiadomości AKCEPTUJ (bazującej na tokenie), ale także odrzucenie dostępu użytkownika A do konta B.

2 Opis implementacji

Implementacja została wykonana w języku Python z wykorzystaniem biblioteki „*pycryptodome*" do celów kryptograficznych oraz „*PyQt*5" do wizualizacji aplikacji.

Skrypty uruchomieniowe:
- Server.py - uruchamia aplikację serwera(konsolowo) nasłuchującą na porcie 44444;
- Klient.py - uruchamia aplikację użytkownika(okienkowo) nasłuchującą na porcie 44445;
- Klient2.py - uruchamia aplikację użytkownika(okienkowo) nasłuchującą na porcie 44446.

Plik Server.py:
- procedurę „*INIT*" z algorytmu 2 dokumentacji reprezentuje metoda „*start*" klasy Server, generuje ona klucze i rozpoczyna nasłuch wykonywany przez serwer.

- procedurę „*SET_PUBLIC_KEY*" z algorytmu 2 dokumentacji reprezentuje metoda „*_add_user*" klasy Server, zbiera ona dane przysłane od użytkownika i zapisuje je w pamięci(nazywane będzie to dalej rejestracją). Korzysta z pomocniczej metody „*_is_name_used*"(sprawdza, czy serwer zarejestrował już takiego użytkownika) oraz „*_add_user_to_list*"(dodaje nowego użytkownika do zbioru danych serwera).

- procedurę „*SEND_PUBLIC_KEY_BY_NAME*" z algorytmu 2 dokumentacji reprezentuje metoda „*_send_public_key_by_name*" klasy Server, wyszukuje ona nazwy użytkownika w zbiorze danych zapisanych przez serwer, odsyła klucz oraz jego podpis gdy nazwa zostanie znaleziona. Korzysta z pomocniczej metody „*_is_name_used*" oraz „*_get_user_by_name*".

Plik Client.py:
- nie jest plikiem uruchomieniowym, posiada tylko klasę, na bazie której budowany jest potem program użytkownika(od strony komunikacji). W konstruktorze podawany jest por,t na jakim aplikacja ma zostać uruchomiona, co daje możliwość utworzenia przykładowych skryptów Klient3, Klient4, Klient5 ... i, przy małej modyfikacji klasy Client, uruchomienia implementacji protokołu dla dużej liczby użytkowników(tyle ile starczy portów).

- metoda „*__init__*" wykonuje linie 2,3,4 algorytmu 1 oraz procedurę „*INIT*" algorytmu 3, czyli generację klucza publicznego oraz prywatnego użytkownika.

- metoda „*register_in_server*"(wywołuje metodę „*_register_in_server*" która wykonuje

główną pracę) wykonuje linię 6 algorytmu 1. Wysyła klucz publiczny oraz nazwę użytkownika, którą serwer zapisuje w pamięci, co nazwane jest rejestracją. W odpowiedzi otrzymuje klucz publiczny serwera.

- metoda „*get_public_key_from_sever*" realizuje linię 8-12 algorytmu 1. Wysyła do serwera nazwę użytkownika, z którym planuje się skontaktować i otrzymuje klucz prywatny w wiadomości zwrotnej. Po otrzymaniu kluczy weryfikuje je.

- metoda „*connect_to_user*" realizuje linię 14-29(do końca) algorytmu 1. Rozpoczyna oraz kończy komunikację z użytkownikiem o nazwie podanej w parametrze „*username*".

- metoda „*_receive_contact*" opisana została jako procedura „$RECEIVE\_CONTACT$" w algorytmie 3.

- metody „*read_the_socket*" i „*_read_the_socket*" są odpowiedzialne za utrzymywanie nasłuchu prowadzonego przez aplikację klienta otworzoną na danym porcie

3 Weryfikacja formalna

Weryfikacja protokołu została wykona w Verifpalu. Zawiera opis protokołu w oferowanym przez Verifpal języku oraz zbiór pytań pozwalający znaleźć możliwe luki w protokole.

3.1 Kod źródłowy

3.1.1 Analiza 1

attacker[active]

principal Alice[
knows private nameA, nameB
generates pA
$A = G^{pA}$
]

principal Bob[
knows private nameA, nameB
generates pB
$B = G^{pB}$
]

principal Server[
generates pT
$T = G^{pT}$
]

$Server -> Alice : T$
$Server -> Bob : T$
$Alice -> Server : A$
$Bob -> Server : B$

principal Server[
hpka = SIGN(pT, A) hpkb = SIGN(pT, B)
]

$Alice- > Server : nameB$
$Server- > Alice : B, hpkb$

principal Alice[
_ = SIGNVERIF(T, B, hpkb)?
generates pub_key_a
$priv\_key\_a = G^{pub\_key\_a}$
generates secret_key_a
knows private time
enc_time = ENC(secret_key_a, time)
generates validity
pack = CONCAT(validity, nameA, pub_key_a, priv_key_a)
s_pack = SIGN(pA, CONCAT(validity, nameA, pub_key_a, priv_key_a))
enc_sec_key = PKE_ENC(B, secret_key_a)
sign_enc_sec_key = SIGN(pub_key_a, enc_sec_key)
]

$Alice- > Bob : enc\_time, pack, s\_pack, enc\_sec\_key, sign\_enc\_sec\_key$
$Bob- > Server : nameA$
$Server- > Bob : A, hpka$

principal Bob[
_ = SIGNVERIF(T, A, hpka)?
_ = SIGNVERIF(A, pack, s_pack)?
valid, namA, pu_ke_a, pri_ke_a = SPLIT(pack)
_ = SIGNVERIF(pri_ke_a, enc_sec_key, sign_enc_sec_key)?
sec_key_a = PKE_DEC(pB, enc_sec_key)
]

queries[
//confidentiality? pA
//confidentiality? pB
//confidentiality? pT
confidentiality? pub_key_a
confidentiality? priv_key_a

confidentiality? secret_key_a
$authentication?Server- > Alice : B$
$//authentication?Server- > Bob : A$
$//authentication?Alice- > Bob : pack$
$//authentication?Alice- > Bob : enc\_sec\_key$
freshness? hpka
//freshness? hpkb
//freshness? s_pack
//freshness? sign_enc_sec_key
$authentication?Server- > Alice : B[$
$precondition[Alice- > Bob : pack]$
]
]

3.1.2 Analiza 2

attacker[active]

principal Alice[
knows private nameA, nameB
generates pA
$A = G^{pA}$
]

principal Bob[
knows private nameA, nameB
generates pB
$B = G^{pB}$
]

principal Server[
generates pT
$T = G^{pT}$
]

$Server- > Alice : [T]$
$Server- > Bob : [T]$
$Alice- > Server : [A]$
$Bob- > Server : [B]$

principal Server[
hpka = SIGN(pT, A)
hpkb = SIGN(pT, B)
]

$Alice- > Server : [nameB]$
$Server- > Alice : [B], [hpkb]$

principal Alice[
_ = SIGNVERIF(T, B, hpkb)?
generates pub_key_a
$priv\_key\_a = G^{pub\_key\_a}$
generates secret_key_a
knows private time
enc_time = ENC(secret_key_a, time)
generates validity
pack = CONCAT(validity, nameA, pub_key_a, priv_key_a)
s_pack = SIGN(pA, CONCAT(validity, nameA, pub_key_a, priv_key_a))
enc_sec_key = PKE_ENC(B, secret_key_a)
sign_enc_sec_key = SIGN(pub_key_a, enc_sec_key)
]

$Alice- > Bob : [enc\_time], [pack], [s\_pack], [enc\_sec\_key], [sign\_enc\_sec\_key]$
$Bob- > Server : [nameA]$
$Server- > Bob : [A], [hpka]$

principal Bob[
_ = SIGNVERIF(T, A, hpka)?
_ = SIGNVERIF(A, pack, s_pack)?

valid, namA, pu_ke_a, pri_ke_a = SPLIT(pack)
_ = SIGNVERIF(pri_ke_a, enc_sec_key, sign_enc_sec_key)?
sec_key_a = PKE_DEC(pB, enc_sec_key)
]

queries[
confidentiality? pA
confidentiality? pB
confidentiality? pT
confidentiality? pub_key_a
confidentiality? priv_key_a
confidentiality? secret_key_a
$authentication?Server- > Alice : B$
$authentication?Server- > Bob : A$
$authentication?Alice- > Bob : pack$
$authentication?Alice- > Bob : enc\_sec\_key$
freshness? hpka
freshness? hpkb
freshness? s_pack
freshness? sign_enc_sec_key
$authentication?Server- > Alice : B$[
$precondition[Alice- > Bob : pack]$
]
]

W analizie został wykorzystany atakujący aktywny, co pozwala zlokalizować miejsca czułe na ataki statystyczne i brutalne. Wiemy, że stosowany w algorytmie numer jeden AES jest odporny na ataki kryptoanalizy różnicowej oraz liniowej ze względu na stosowaną w nim funkcję substytucyjną o oryginalnej konstrukcji.

3.2 Analiza protokołu

Pierwszą analizę przeprowadziliśmy na protokole pozwalającym atakującemu na manipulację wszystkimi przesyłanymi danymi. Ze względu na bardzo dużą liczbę prób atakującego, a tym samym zbyt dużą liczbę obliczeń, została zadana zmniejszona pula pytań. Atakującemu udało się uzyskać bez dodatkowych założeń losowy klucz publiczny Alice oraz jej losowy klucz prywatny. Gdyby atakujący zdobył klucz publiczny Boba oraz jego podpis przy użyciu klucza prywatnego Servera, to byłby w stanie uzyskać także losowy klucz tajny Alice, a ona wysłałaby paczkę z okresem ważności klucza, swoim identyfikatorem i losowymi kluczami prywatnym i publicznym do Boba. Przy tym samym założeniu mógłby w imieniu Servera wysłać Alice klucz publiczny Boba, a mimo to weryfikacja przebiegłaby pomyślnie. Kiedy atakujący zna podpis klucza publicznego Alice kluczem prywatnym Servera i podpis losowym kluczem publicznym Alice jej zaszyfrowanego klucza tajnego (szyfrowany kluczem publicznym Boba), to wartość podpisu klucza publicznego Alice nie jest świeża i może zostać użyta do ataku powtórzeniowego.
Po tej analizie odebraliśmy atakującemu możliwość modyfikacji przesyłanych między stronami wartości i mogliśmy sobie pozwolić na rozszerzenie listy pytań. W tej sytuacji atakujący jest w stanie wejść w posiadanie tylko losowego klucza publicznego Alice oraz jej losowego klucza prywatnego.