Mikołaj Siedlecki, Jan Walczak

Politechnika Warszawska, Wydział Elektroniki i Technik Informacyjnych

Profesjonalny algorytm kryptograficzny - Makwa, grupa: 102

11 czerwca 2022

Spis treści

Opracowanie w ramach laboratoriów BDAN, 2022. Mikołaj Siedlecki, Jan Walczak, PW.

1. Wstęp

Makwa jest funkcją hashujacą hasła, stworzoną przez Kanadyjczyka Thomasa Pornina w lutym 2014 roku. Funkcja Makwa została wyłoniona jako finalista spośród 24 uczestników konkursu PHC w latach 2013-2015. Dodatkowo została wyróżniona, otrzymała "special recogniction" jako jedyna funkcja ze wszystkich uczestniczących w zawodach oferowała delegację obliczeń(Delegation).

2. Właściwości Makwa

Główną funkcjonalnościa wyróżniających Makwa spośród kandydatów jest w.w. **delegacja obliczeń**. Pozwala ona na przeniesienie głównych najbardziej wymagających pamięciowo i obliczeniowo operacji poza platforme na której wykonywana jest funkcja. Platforma ta może mieć formę zewnętrznego modułu przeznaczonego właśnie do tego typu obliczeń lub chmury obliczeniowej. Innymi właściwościami Makwa są m.in.:

— Obsługa "szybkiej ścieżki" - fast path, pozwala to skrócić najbardziej wymagające obliczenia wykonywane w funkcji. Jest to możliwe przy znajomości czynników p i q modułu N, owe czynniki w celu zapobiegnięcia możliwości wystąpienia podatności na ataki słownikowe(dictionary attack) powinny one pozostać tajne dla osób trzecich.
— Możliwość pre i post hashowania. Hashowanie "przed" zapewnia zwiększenie bezpieczeństwa naszego hasha. Hashowanie "po" również zwiększa bezpieczeństwo, jednakże dodatkowo pozwala skrócić długość ostatecznego wynikowego do zadanej, wybranej przez użytkownika wartości.
— Możliwość wykorzystania soli, zapewnia dodatkowe bezpieczeństwo(Użyta sól ma również wymaganą wielkość minimalną)
— Opartość na arytmetyce modularnej(Modulo)
— Wykorzystanie innych funkcji hashujących - korzystanie z funkcji HMAC, pozwala na dobranie jednej z wielu funkcji hashujący głównego modułu(podstawowa wersja Makwa zakłada wykorzystanie SHA-256)

3. Program

Przygotwana przez nas implementacja Makwa została napisana w języku Java. Nasz program składa się z trzech klas potrzebych do realizacji funkcji, oraz jednej klasy w której używamy Makwa i sprawdzamy poprawność wektorów testowych.

3.1. Utility

Klasa zawierająca w sobie narzędzia potrzebne w różnych etapach realizacji zadania:

— HexStringToByteArray - funkcja pozwalająca na przekształcenie Stringa zawierającego liczbe w postaci hexadecymalnej na tablice byte'ów
— ByteArrayToBigInteger - funkcja pozwalająca na przekształcenie tablicy byte'ów na BigInt
— BigIntegerToByteArray - funkcja pozwalająca na przekształcenie BigInteger na listę byte'ów
— ConcatenateByteArrays - funkcja pozwalająca na połączenie ze sobą w jedną tablice byte'ów

3.2. Generators

Funkcja ta była kilkukrotnie przebudowywana, jej finalna postać jest mocno okrojona w porównaniu z początkowymi zamysłami jej powstania. Klasa zawiera jedną metodę, która służy do utworzenia kluczy przy pomocy RSA, funkcja przyjmuję argument mówiący jak długi ma być klucz, nastąpnie zwraca wartość klucz prywatnego pod postacią BigInt.

3.3. Makwa

Główna klasa naszego algorytmu Makwa, posiada konstruktor i klasę wewnętrzną służącą do wypisywania naszego hasha w postaci Stringa.

Poniżej przedstawione ważniejsze części klasy:

— **Konstruktor** - jako argumenty przyjmuje preHashing i postHashing typu boolean mówiące o włączeniu bądź wyłączeniu funkcji pre i post hashowania. MCost(memory cost) które jest typu int i decyduje o liczbie iteracji głównej części naszej funkcji hashującej(squaring). Następnie postHashLength które ustala długość końcowego hasha jeżeli zdecydowaliśmy się na postHashowanie. Na koniec ustalamy modulo z jakim mają wykonywać się nasze operacje podnoszenia do kwadratu.
— **CreateHash** - szereg funkcji przyjmujących jako argumenty hasło i sól w permutacjach typu String i tablicy byte'ów i następnie przekazujących hasło w poprawnej postaci do funkcji createHashFinal.
— **CreateHashFinal** - funkcja jako argument przyjmuje hasło w postaci tablicy byte'ów, na początku jeżeli użytkownik wybrał tę opcję to przeprowadzane jest pre-hashowanie, następnie zostaje przeprowadzona walidacja danych - długość przekazanej tablicy byte'owej. Po walidacji robimy konkatenacje tablic byte'ów soli, hasła, długości. Na połączonym łańcuchu wykonywana jest funkcja KDF i wynik przypisywany jest do tablicy byte'ów sb. W następnym kroku ponownie wykorzystujemy konkatenacje tablicy hexZero, wcześniej utworzonej sb oraz długości hasła, wynik tej operacji przypisujemy do xb. Następnie ponownie wykorzystując klasę Utility przepisujemy xb z formy tablicy byte'ów na BigInt do zmiennej x. Teraz możemy przejść do głównej pętli programu, wykonywanej mCost(ustalony wcześniej przez użytkownika parametr) razy, podnosimy nasz x do kwadratu modulo(również wcześniej przez nas zdefinowane). Po zakończeniu operacji przepisujemy wartość BigInt z powrotem na tablicę byte'ów i ewentualnie jeżeli było ustawione to wykonujemy postHashowanie. Zwracamy Y.

4. Pseudokod

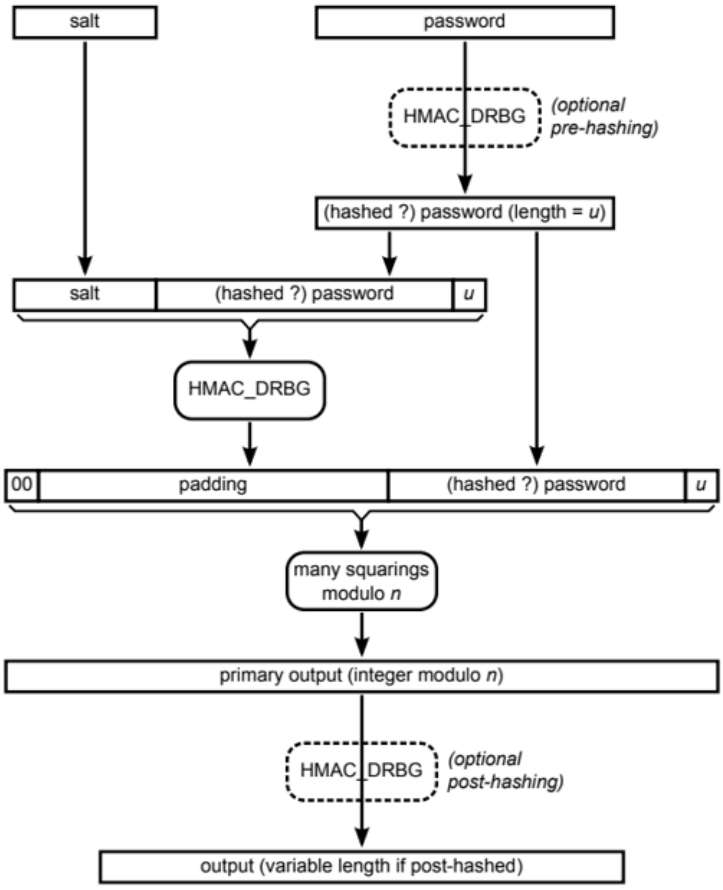
```
# *** Functions/symbols ***
# ||                Concatenate two strings
# len(string)       Byte length of a string
# len(bigint)       Byte length of a big integer (unsigned)
# square_mod        Modular squaring
# STR_TO_BIGINT_BE  Convert a string to a bigint (big-endian, unsigned)
# BIGINT_TO_STR_BE  Encode a bigint into a string (big-endian) with a definite output length
# BYTE(integer)     Encode an integer into a string of exactly one byte
# HMAC(h, k, v)     Compute HMAC with hash function h and key k over value v
# trunc(m, j)       Truncate string m to its first j bytes

# *** Inputs ***
string    password
string    salt
integer   m_cost
boolean   pre_hashing
integer   post_hashing_length

# *** Parameters ***
# These parameters are supposed to be server-wide.
bigint    modulus    # a Blum integer (product p*q, p = 3 mod 4, q = 3 mod 4)
function  h          # a hash function, e.g. SHA-256

# *** Algorithm ***
if m_cost < 0
        return ERROR
k = len(modulus)
if k < 160
        return ERROR

# Pre-hash input password (optional)
if pre_hashing
        password = KDF(password, 64)
```

```
# Salt-derived padding for password
u = len(password)
if u > 255 OR u > (k - 32)
        return ERROR
sb = KDF(salt || password || BYTE(u), k - 2 - u)
xb = BYTE(0x00) || sb || password || BYTE(u)

# Main loop: repeated modular squarings.
x = STR_TO_BIGINT_BE(xb)
for i = 0 to m_cost
        x = square_mod(x, N)
out = BIGINT_TO_STR_BE(x, len(N))

# Post-hashing (optional)
if post_hashing_length > 0
        out = KDF(out, post_hashing_length)

# Finish
return out
```

Funkcja KDF - główna funkcja pomocnicza

```
# *** Helper function ***
KDF(data, out_len)
        r = output length of h() in bytes
        V = BYTE(0x01) || BYTE(0x01) || ... || BYTE(0x01)  # such that len(V) = r
        K = BYTE(0x00) || BYTE(0x00) || ... || BYTE(0x00)  # such that len(K) = r
        K = HMAC(h, K, V || BYTE(0x00) || data)
        V = HMAC(h, K, V)
        K = HMAC(h, K, V || BYTE(0x01) || data)
        V = HMAC(h, K, V)
        T = empty
        while len(T) < out_len
                V = HMAC(h, K, V)
                T = T || V
        return trunc(T, out_len)
```

4.1. Schemat działania algorytmu



Rys. 1. Schemat działania algorytmu

5. Wektory testowe

Wektory testowe łącznie z wynikami są zawarte w klasie Main naszego programu.

Parametry które były wymagane przy wektorze testowym:
— preHashing - false
— postHashing - true
— mCost - 4096
— postHashLength - 12
— mod - C22C40BBD056BB213AAD7C830519101AB926AE18E3E9FC9699C806E0AE5C259414A01AC1D52E873E C08046A68E344C8D74A508952842EF0F03F71A6EDC077FAA14899A79F83C3AE136F774FA6EB88F1D1AEA 5EA02FC0CCAF96E2CE86F3490F4993B4B566C0079641472DEFC14BECCF48984A7946F1441EA144EA4C80 2A457550BA3DF0F14C090A75FE9E6A77CF0BE98B71D56251A86943E719D27865A489566C1DC57FCDEFAC A6AB043F8E13F6C0BE7B39C92DA86E1D87477A189E73CE8E311D3D51361F8B00249FB3D8435607B14A1E 70170F9AF36784110A3F2E67428FC18FB013B30FE6782AECB4428D7C8E354A0FBD061B01917C727ABEE0 FE3FD3CEF761
— sól - 0xC72703C22A96D9992F3DEA876497E392
— hasło tekst - Gego beshwaji'aaken awe makwa; onzaam naniizaanizi.

— hasła hex - 0x4765676F206265736877616A692761616B656E20617765206D616B77613B206F6E7A61616D206E616E69697A61616E697A692E

Oczekiwane wyniki

Finally, the 12-byte output is:

```
out = C9 CE A0 E6 EF 09 39 3A B1 71 0A 08
```

With the string output encoding defined in section A.4, MAKWA produces the following string of 56 characters:

```
+RK3n5jz7gs_s211_xycDwiqW2ZkvPeqHZJfjkg_yc6g5u8JOTqxcQoI
```

Rys. 2. Oczekiwane wyniki

```
"C:\Program Files\Java\jdk-18.0.1\bin\java.exe" ...

TEST VECTORS:

Hash created from PasswordBytes and saltByteArray:
preHashing: OFF
postHashing: ON
Hash text: +RK3n5jz7gs_s211_xycDwiqW2ZkvPeqHZJfjkg_yc6g5u8JOTqxcQoI

Hash created from PasswordText and saltByteArray:
preHashing: OFF
postHashing: ON
Hash hex:
c9-ce-a0-e6-ef-9-39-3a-b1-71-a-8-
```

Rys. 3. Nasze wyniki

6. Przemyślenia

Na plus:

— Przejrzystość i zwięzłość dokumentacji Makwa - 50 stron dokumentacji jest napradę konkretne i zawiera szczegółowe informacje na temat większości zagadnień dotyczących algorytmu. Istnieją szczegółowe odniesienia dotyczące dalszego rozwinięcia zagadnienia w dokumentacji.
— Stosunkowo prosty w implementacji i szczegółowy pseudokod
— Dobrze przygotowany wektor testowy który posiada opis kolejnych wartości w różnych fazach algorytmu, w znaczący sposób ułatwie to debuggowanie programu i śledzenie kolejnych wartości wektora

Na minus:

— Problematyczna obsługa typu byte w Javie, która uznaje każdą z liczb jako signed, jedna z rzeczy która przy realizacji projektu zajęła nam najwięcej czasu. Przy konwersji np. Stringa zawierającego liczbę hexadecymalną na tablice byte'ów na pierwszym miejscu w tablicy dodawane było zero przez co musieliśmy to obsłużyć w wielu miejscach programu.

Ostatecznie uważamy, że realizacja projektu pozwoliła nam na rozwinięcie naszych umiejętności programistycznych, implementacji pseudokodu, oraz czytania i realizacji dokumentacji.