Protokół SRP

Spis treści

[Team: 1](#_Toc43119588)

[Opis protokołu SRP 1](#_Toc43119589)

[Opis działania 1](#_Toc43119590)

[Oznaczenia 2](#_Toc43119591)

[Diagram UML 3](#_Toc43119592)

[Implementacja Python/Sagemath 4](#_Toc43119593)

[Realizacja w Verifpal 6](#_Toc43119594)

## Team:

Weronika Buras

Radosław Motyka

Mateusz Konar

## Opis protokołu SRP

Protokół SRP to tak zwany zero-knowledge proof protocol co w dosłownym tłumaczeniu oznacza “zero wiedzący”. O co więc chodzi? Mianowicie serwer nie przechowuje hasła i klient może pewnie i śmiało się uwierzytelnić. Ataki typu man-in-the-middle czy eavesdropper nie mogą uzyskać istotnych wiadomości do przeprowadzeni ataku. Zalety SRP:

* Uwierzytelniania bez wysyłania hasła do sieci
* Chroni przed atakami man-in-the-middle czy eavesdroppe
* Daje pewność identyfikacji serwera do klienta i na odwrót, daje pewność, że nie grozi nam towarzystwo osoby niepożądanej
* Dodatkowo generuje zaszyfrowany klucz, tak więc klient &serwer mogą przesyłać zaszyfrowane dane na poziomie bezpieczeństwa TLS

Podsumowując SRP jednym zdaniem “Nie wykradniesz hasła, jeśli go nie przechowujesz”

## Opis działania

Rejestracja:

1. Klient wybiera losową wartość soli **s** i oblicza **x=H(s,p), v= gx**
2. Klient->Serwer: **v,s** indeksowane przez nazwę użytkownika **I**

Uwierzytelnienie:

1. Klient->Serwer: generuje losową wartość **a**, wysyła **I** (nazwę użytkownika) oraz **A = ga**
2. Serwer -> Klient: generuje losową wartość **b**, wysyła **s** i **B = kv + gb**
3. Obydwoje: **u=H(A,B)**
4. Klient oblicza **SKlient = (*B* − *kg*x)(*a* + *ux*) = (*kv* + *g*b − *kg*x)(*a* + *ux*) = (*kg*x − *kg*x + *g*b)(a + ux) = (*g*b)(*a* + *ux***)
5. Klient oblicza **KKlient=H(SKlient**)
6. Serwer oblicza ***S*Serwer = (*Av*u)b = (*g*a*v*u)b = [*g*a(*g*x)u]b = (*g*a + ux)b = (*g*b)(a + ux)**
7. Serwer oblicza  **KSerwer= H(*S*Serwer)= KKlient**

Następnie następuje proces udowodnienia pasowania kluczy. Jedną z możliwości jest:

1. Klient→ Serwer: ***M*1 = *H*[*H*(*N*)** [**XOR**](https://en.wikipedia.org/wiki/XOR) ***H*(*g*) | *H*(*I*) | *s* | *A* | *B* | *K*Klient].** Serwer weryfikuje ***M*1.**
2. Serwer → Klient: ***M*2 = *H*(*A* | *M*1 | *K*Serwer)**. Klient weryfikuje ***M*2**.

Alternatywa dla niej:

1. Klient→ Serwer: ***M*1 = *H(* *A* | *B* | SKlient ).** Serwer weryfikuje ***M*1.**
2. Serwer → Klient: ***M*2 = *H*(*A* | *M*1 | SSerwer)**. Klient weryfikuje ***M*2**

Używając SRP do wyznaczenia dzielonego klucza K, który zostanie użyty natychmiast po wyznaczeniu weryfikacja *M*1  oraz *M2* może zostać pominięta.

Zabezpieczenia:

1. Klient przerwie transakcję, jeśli otrzyma **B=0(modN)** lub **u=0**
2. Serwer przerwie transakcję jeśli otrzyma **0=A(modN)**
3. Klient musi okazać **KKlient** lub **SKlient** **najpierw**. Jeśli serwer otrzyma niepoprawne dane przerywa transakcję bez wysyłania jego  **KSerwer** lub ***S*Serwe**

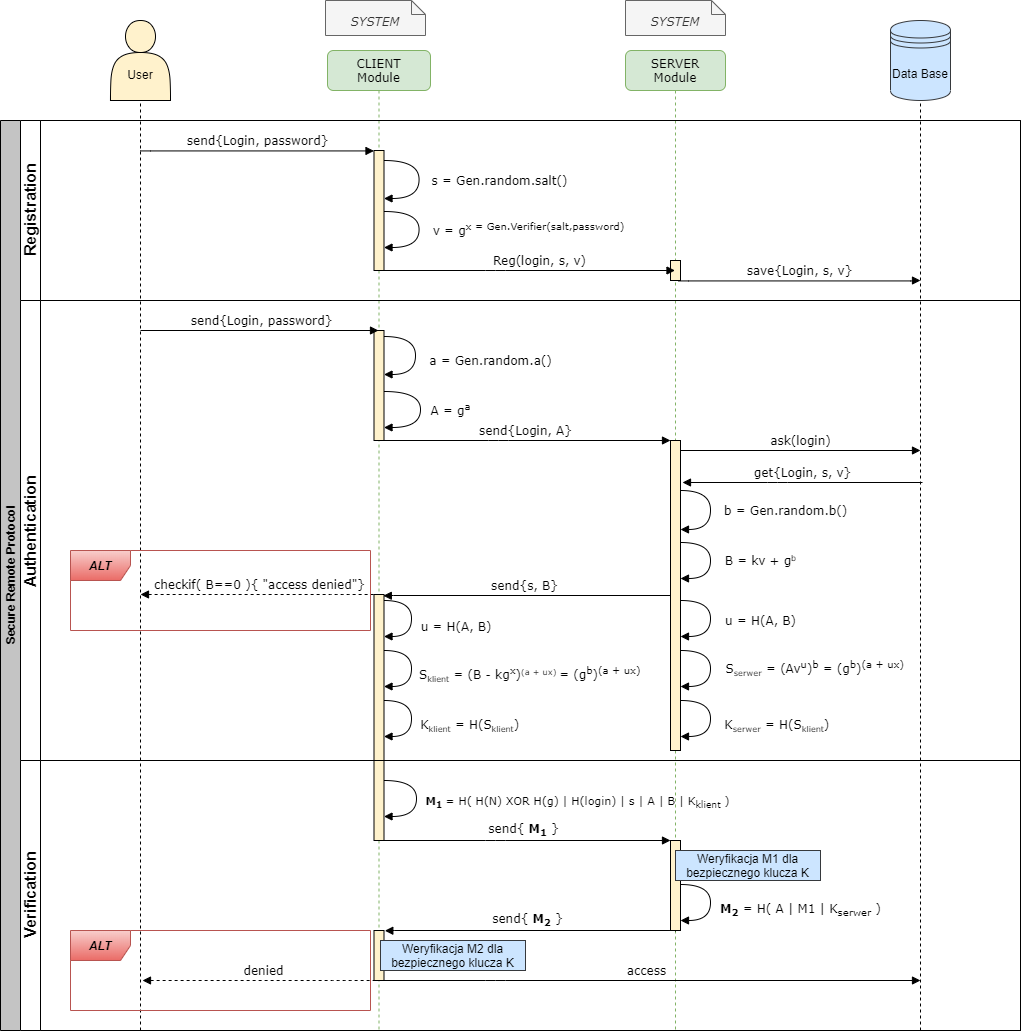
# 

## Oznaczenia

Poniższe oznaczenia i założenia będą realizować podczas tej realizacji protokołu SRP

* q - duża liczba pierwsza
* N = 2q + 1 - duża liczba pierwsza (co najmniej 1000 bajtów)
* Wszystkie obliczenia są wykonywane w obrębie pierścienia całkowitego modulo N
* g - generator grupy
* H() - funkcja haszująca, np. SHA-256
* k – parametr pochodzący z N i g: k = H(N, g). Używany do zapobiegania “2-for-1 guess”, gdy atakujący podaje się za serwer
* s - sól (losowy ciąg znaków)
* I - nazwa użytkownika
* p - hasło użytkownika
* v = gx - weryfikator hasła
* x = H(s, p)
* A i B – losowe, jednorazowe, nietrwałe klucze odpowiednio użytkownika i serwera

## Diagram UML



## 

## Implementacja Python/Sagemath

|  |
| --- |
|  |
|  | import random |
|  | import hashlib |
|  | def Sha(\*args): |
|  | a = ':'.join(str(a) for a in args) |
|  | return int(hashlib.sha256(a.encode('utf-8')).hexdigest(), 16) |
|  |  |
|  | def randnum(n): |
|  | return random.SystemRandom().getrandbits(n) % N |
|  |  |
|  | g = 2 # generator modulo N |
|  | N = 8069496435\*(10\*\*5072)-1 # liczba pierwsza Sophie Germain |
|  | k = Sha(N, g) # funkcja hashująca dla Sha256 i k=3 |
|  |  |
|  | print("H = ", Sha, "\nN = ", N, "\ng = ", g, "\nk = ", k, "\n") |
|  |  |
|  | I = "WAT2020" |
|  | p = "WCY18KY2S1" |
|  |  |
|  | print("REJESTRACJA:\n") |
|  | print("1.Uzytkownik wysyla LOGIN:", I, "HASLO:", p) |
|  | print("\n2.Modol uzytkownika generuje weryfikator:") |
|  |  |
|  | s = randnum(64) # losowa sól |
|  | x = Sha(s, I, p) # Klucz prywatny |
|  | v = pow(g, x, N) |
|  | print(v, "\n\noraz klucz prywatny:", x) |
|  | print("\n3.Modol serwera zapisuje pod indeksem:", I,"salt =", s,"i weryfikator") |
|  |  |
|  | print("\n\nAUTORYZACJA:\n") |
|  |  |
|  | a = randnum(64) |
|  | A = pow(g, a, N) |
|  | print("1. Uzytkownik wysyla login:", I,"oraz krótkotrwaly klucz publiczny A =\n", A,"\n\nna podstawie losowej prywatnej liczby a =", a) |
|  |  |
|  | b = randnum(64) |
|  | B = (k \* v + pow(g, b, N)) % N |
|  | print("\n2. Serwer wysyla salt:", s,"oraz krótkotrwaly klucz publiczny B =\n", B,"\n\nna podstawie losowej prywatnej liczby b =", b) |
|  |  |
|  | u = Sha(A, B) |
|  | print("\n3. Uzytkownik i serwer generuje wspolny klucz SHA(A, B) = U =", u) |
|  |  |
|  |  |
|  | print("\n4. Uzytkownik oblicza klucz sesji, a nastepnie hashuje go") |
|  |  |
|  | S\_u = pow(B - k \* pow(g, x, N), a + u \* x, N) |
|  | K\_u = Sha(S\_u) |
|  | print("Klucz sesji: ", S\_u) |
|  | print("\nHashowany klucz sesji: ", K\_u) |
|  |  |
|  | print("\n5. Serwer oblicza klucz sesji, a nastepnie hashuje go") |
|  | S\_s = pow(A \* pow(v, u, N), b, N) |
|  | K\_s = Sha(S\_s) |
|  | print("Klucz sesji: ", S\_s) |
|  | print("\nHashowany klucz sesji: ", K\_s) |
|  |  |
|  | print("\n\nWERYFIKACJA:\n") |
|  | print("6. Uzytkownik wysyla potwierdzenie klucza sesji W\_u =") |
|  | HN = Sha(N) |
|  | Hg = Sha(g) |
|  | HI = Sha(I) |
|  | W\_u = Sha((HN^Hg), HI, s, A, B, K\_u) |
|  | print(W\_u) |
|  |  |
|  | print("2. Serwer wysyla potwierdzenie klucza sesji W\_s =") |
|  | W\_s = Sha(A, W\_u, K\_s) |
|  | print(W\_s) |

## 

## Realizacja w Verifpal

|  |
| --- |
|  |
|  | attacker[active] |
|  | principal Alice[] //in our description tis is Client |
|  | principal Bob[] // in our description this is Server |
|  |  |
|  | principal Alice |
|  | [ |
|  | generates a, s //generates a and salt |
|  | knows private skA, p, I//knows secret key, password and user name |
|  | ga = G^a //Diffie-Helman key |
|  | x = HASH(s, p) |
|  | v = G^x |
|  | sigA = sign(skA, ga) //siganture key - private key and message |
|  | ] |
|  |  |
|  | Alice -> Bob: ga, sigA, v,s, I |
|  |  |
|  | principal Bob |
|  | [ |
|  | generates b, N //generate b and N- big prime number |
|  | knows private skB, skA //knows secret key Bob's and Alice's |
|  | validateB = SIGNVERIF(G^skA, ga, sigA)? //verifies signature |
|  | gb = G^b //Diffie-Helman key |
|  | k = HASH(N, G) |
|  | sigB = SIGN(skB, gb) //siganture key - private key and message |
|  | u=HASH(ga,gb) //both Bob and Alice calculate this |
|  | ] |
|  |  |
|  | Bob -> Alice: gb, sigB, u |
|  |  |
|  | principal Alice |
|  | [ |
|  | knows private skB |
|  | generates sa, pt1 |
|  | validateA = SIGNVERIF(G^skB, gb, sigB)? //verifies signature |
|  | M1=ENC(HASH(CONCAT(ga,gb,Sa)),pt1) // symetric encyryption of hash, proof keys match |
|  | ] |
|  |  |
|  | Alice -> Bob: M1 |
|  | principal Bob[ |
|  | generates sb |
|  | M2=DEC(HASH(CONCAT(ga,M1,Sb)),M1) //symetric decyryption of hash, proof keys match |
|  | ] |
|  |  |
|  | queries |
|  | [ |
|  | authentication? Alice -> Bob: M1 |
|  | authentication? Bob -> Alice: sigB |
|  | confidentiality? pt1 //poufnosć |
|  | ] |