Brute Forcing Weak Secure Remote Password Implementation

TBC Edinburgh Napier University, Edinburgh, UK.

Abstract

Passwords have been and are the main way to authenticate with a unique identifier such as a username on to the majority if not all IT systems and platforms. These systems can hold sensitive information for both users and organisations. This can make credentials a high valued target for anyone wishing to gain access to that system. This is why we try to protect password databases by restricting access to passwords, as well as using a unique salt with a suitable bit length and slow one-way hashing algorithm. The issue here is, given enough time and resources, the password will be cracked and best practice is not always adhered too. Barely a week goes by where another set of user credentials are leaked with passwords not stored inline with best practice. This paper will examine Secure Remote Password (SRP); a way to prove the user has the correct password without ever having to transmit or store the password on a server, and what can go wrong if not implemented correctly.

Keywords: Zero-knowledge Proof, secure remote password, PAKE,

1. Introduction

PAKE was first introduced in 1992 by Bellovin and Merritt and was mainly developed to mitigate against dictionary attacks [1]. SRP is a challenge response authentication protocol which can use many underlying protocols, such as Authentication & Key Exchange (AKE) developed by Thomas Wu from Stanford University [2]. RFC2945, which describes SRP-3 AKE, was published in September 2000 [3]. This has been the foundation for the most common protocol that has been used with SRP: Password-authentication key exchange (PAKE) or a PAKE variant. At this time browser-to-website encryption on public websites was not as prevalent or as easy and cheap to implement as it is today, so a way to authenticate over insecure network was necessary. It provides perfect forward secrecy, so if the challenge response was captured over an insecure network the password should not be disclosed and past sessions and access to the account would not be easily obtained. In 2018, Joz Lopez published a paper describing how an offline dictionary attack could be carried out on zkPAKE if the initial setup communications were intercepted by an attacker [4]. As we are focusing on addressing the issue of passwords being stored in databases and as authentication today is increasing happening in a secure encrypted tunnel such as TLS, this is considered to be out of scope for this paper. Although this may cause an issue when quantum computing is utilised to break the encryption of these tunnels, a new two lattice-based protocol has been proposed by Jintai ding, et al [1] and a Isogeny-based PAKE protocol by Oleg Taraskin, et al that may quantum proof this process. SRP has had many revisions over the the last 19 years. This paper will look at the current version of SRP 6a, how it works and the pitfalls if it is not adhered to. The protocols main strength is that it generates a strong cryptographic session key between the client and server to provide secure authentication without ever having to transmit the password.

N	A large safe prime (N = 2q+1, where q is prime)
	All arithmetic is done modulo N.
g	A generator modulo N
k	Multiplier parameter (k = H(N, g) in SRP–6a
s	User's salt
I	Username / identitfier
P	Cleartext Password
H()	One–way hash function
u	Random scrambling parameter
a,b	Secret ephemeral values
A,B	Public ephemeral values
x	Private key (derived from p and s)
v	Password verifier
I	Username / identifier

[5]

When a user creates an account they enter a password P, on the client side a random salt s is generated and added to the password where it is hashed, which gives us x. The verifier and salt are passed to the server along with the username where they are stored[5] (figure 1):

$$x = H(s, P)$$
$$v = g^x$$

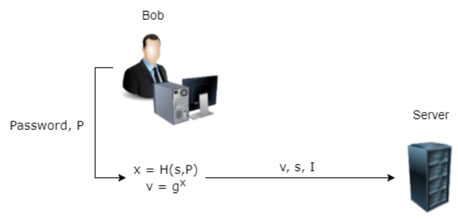

Figure 1: Account creation

The login process to get the strong shared cryptographic session key is as follows (figure 2):

1. Client sends username and calculates A and sends them to the server.

$$I, A = g^a$$

2. Server retrieves the users salt and calculates B and sends this back to the client.

$$s, B = kv + g^b$$

3. Now both the client and server calculate u by hashing the shared values A and B.

$$u = H(A, B)$$

4. User password is now entered in the client and x is calculated the same as in account creation.

$$x = H(s, p)$$

5. Client now has all the values required to calculate the strong session key K.

$$S = (B - kg^x)^{(a+ux)}$$
$$K = H(S)$$

6. Server can also calculate the strong session key K.

$$S = (Av^u)^b$$
$$K = H(S)$$

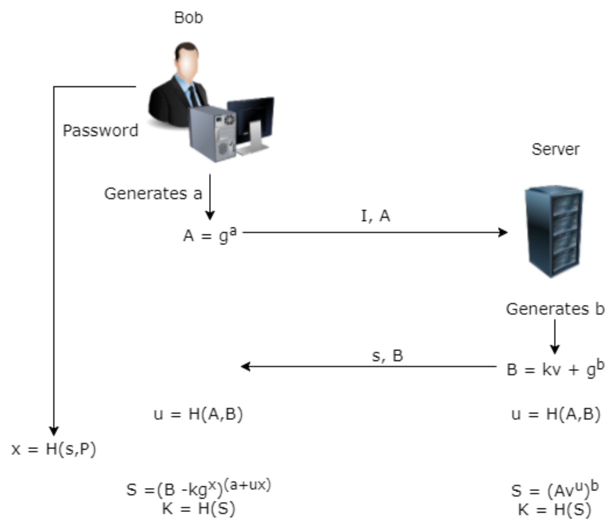

Figure 2: Authentication

The client and server must now prove their session keys match to complete the authentication process. The proposed way by Stanford University is[5]:

1. Client calculates M and sends this to the server.

$$M = H(H(N)\ xor\ H(g), H(I), s, A, B, K)$$

2. Server calculates the hash of the sum of A, M, and K and sends this to the client.

$$H(A, M, K)$$

There are three safeguards that the client and server must use[5]:

1. If the client receives B or u with the value of 0 (zero) from the server.
2. If the server receives A with a value of 0 (Zero) from the client.
3. The client must show its value of K first. If the clients K is incorrect the server must abort without disclosing its value of K.

You may have noticed that if two or more users choose the same password and the salt s is not randomised then they would have the same secret value of x. As x is used to derive the verifier v, this would mean that v would be the same. As both v and s are stored on the server and not the client, a compromised server could send other users salts to different users; if it still authenticates this could tell the malicious actor which users have the same password. We would be able to assume that if the passwords are the same they are likely to be a common and easy to guess password, which would likely be in the top 100k password list released by the NCSC [6]. In RFC2945 [3] it documents adding the username I into the equation that calculates x. This should prevent identical passwords generating the same verifier but it is not clear why this method has been missed out of the SRP 6a update [5]. It is important to add I to the hash to prevent the attacker gaining the knowledge that there is a potential weak password used, as if the attacker could extract the database from the server which stores I, v, and s, (g and N are public knowledge) this means the only unknown value from $v = g^x$ is x (remembering there is an implied mod N of x). This is why it is crucial that the client never shares or stores x. The attacker must find x which could be done by putting a list of commonly used passwords through $x = H(s, P)$. The newly generated x values can now be used to see if any of them match the v, if they do they have successfully cracked the password P. To slow what should already be a long process down, a strong and slow hashing algorithm should be used; PBKDF2 being a prime example. However, RFC5054[7] for which pysrp [8] has aligned its code to only use SHA1-512 and defaults at SHA1 for performance purposes.

2. Related Work

1Password (a password manager) uses SRP at the core of application to authenticate the client side application and the user [9]. Instead of just a user password, 1Password uses two-secret key derivation (2SKD). The first is the "Master Password" which is the password the user enters. The second is the "Secret Key" which includes a version setting the client ID and most importantly a 26 sudo random set of characters which are generated on the client side; it is recommended the "Secret Key" is written down and kept in a safe place. With this approach even if an attacker was able to crack the "Master Password", they would not have access to the vault without the "Secret Key". This paper is interested in how 1Password calculates x to derive v in-order to protect P. 1Password trims any leading white spaces and some normalisation to the user password; this can be ignored. What does apply is the way the s is derived. The initial s is generated by the client application. A new salt is then generated with a lowercase version of the users email address e and is stretched to 32 bits long $s = HKDF(s, version, e, 32)$. The hashing algorithm that is used to get x is PBKDF2-HMAC-SHA256 with 100,000 iterations $x = PBKDF2(P, s, 100000)$. The unique salt per user combined with the slow hashing algorithm of PBKDF2 means that the probability of two users with the same verifier also having the same password is infeasible. To add, with the employment of a slow hashing algorithm it would make brute forcing the password too computationally expensive. Finally, you still need to combine x with the "Secret Key" (another unknown 128 bits) and hashed again $k = HKDF(SecretKey, version, I, x)$.

Tom Cocagne's pysrp is a "Python implementation of SRP" [8]. pysrp comes with a validity and performance testing script which test the SHA variants with four prime numbers at 1024, 2048, 4096, and 8192 bits long. This will allow us to get a baseline of how long in seconds it takes to establish a session; the results are in Table 1 [8]:

Table 1: Hash Algorithm vs Prime Number performance table Baseline

	1024	2048	4096	8192
SHA1	0.000759	0.002135	0.005553	0.020130
SHA224	0.000752	0.002207	0.007225	0.024023
SHA256	0.000779	0.001991	0.007167	0.026835
SHA384	0.001002	0.003025	0.009069	0.033079
SHA512	0.001157	0.003367	0.010958	0.039313

3. Implementation

To obtain the values required in order to brute force SRP passwords, a code based on Stanford University design [5] will be used [10]. Only two changes were made to the code when testing: the hashing algorithm and the prime number. These were changed to test each hash with each prime number bit length. The 1024, 2048, 4096, and 8192 bit length prime numbers were taken from the pysrp _prsrp.py file [8]. The following gives an outline of the Python code created to provide a proof-of-concept to test how long it would take for an attacker to brute force a user password. This is based on the scenario that a servers SRP database has been compromised and the SRP implementation being flawed as highlighted earlier, resulting in two or more users having the same verifier with the same salt. The five values we therefore need are: v, s, and I from the server and the publicly available N and g:

```
import hashlib
import random

# based on http://srp.stanford.edu/design.html extracted from https
    ://asecuritysite.com/encryption/srp

# Between the "----" Lines is coded added by 40414690
#-------------------------------------------------------------
import sys
import time

#import all command line inputs
if (len(sys.argv)>1):
        verifier=int((sys.argv[1]), 16) #Converts hex input to
    interger
if (len(sys.argv)>2):
        passwordtxt=(sys.argv[2]) #password file
if (len(sys.argv)>3):
        s=int((sys.argv[3]), 16) #Converts hex input to interger
if (len(sys.argv)>4):
        I=(sys.argv[4]) #User ID
if (len(sys.argv)>5):
        N=int((sys.argv[5]), 16) #Converts hex input to interger

#-------------------------------------------------------------

# based on http://srp.stanford.edu/design.html


# note: str converts as is, str( [1,2,3,4] ) will convert to
    "[1,2,3,4]"
def H(*args):  # a one-way hash function
    a = ':'.join(str(a) for a in args)
    return int(hashlib.sha1(a.encode('utf-8')).hexdigest(), 16)

g = 5        # A generator modulo N

k = H(N, g)  # Multiplier parameter (k=3 in legacy SRP-6)

# All Code below has been added by Student 40414690

v = 0 #set v to 0

#records time at this point.
start = time.time()

#Counts how many passwords are in file to stop an endless while
    loop.
num_lines = sum(1 for line in open(passwordtxt))
passwordList = open(passwordtxt, "r") #Opens password file

#Loop contiues whilst there are still passwords unless password is
    found.
while num_lines > 0:

    for p in passwordList: #Loops though each password in password
    list.
        p = p.strip() #Strips of the newline (\n) charater at the
    end of each passsword
        x = H(s, I, p)       # Private key from https://
    asecuritysite.com/encryption/srp
        v = pow(g, x, N)     # Password verifier from https://
    asecuritysite.com/encryption/srp
        num_lines = num_lines -1 # decraments line number by 1 each
     for loop to prevent endless while loop.

        if v==verifier: #Check to see if verifier calculated
    matches inputted verifier.
            print("Password is " + p) #Prints out password that was
     used to match the verifier.

            break #breaks out of for loop

    break #Breaks out of while loop

if v !=verifier: #If passwords do not create a matching verifier do
     this
    print("Password not found")
    end = time.time()  #Calcuates how long loop has taken
    print("Time takes in seconds " + str(end - start))

else: #Prints time taken to find the correct password.
    end = time.time()
    print("Time takes in seconds " + str(end - start))
```
Listing 1: Code extract

This code uses values v, s, I, and N along with a password file, converting the hex values back to integers. The value g is hard coded and has to be changed manually. The code then loops through each password in the file until it matches the given v which will print the password to the console, or it has exhausted the password list. For both outcomes it will also print out in seconds how long it took to complete. A caveat needs to be applied in the scenario that the I has not been added to the generation of x as highlighted at the end of section 1, therefore providing the weakness. This code uses I and therefore brute forcing times are likely to be longer. Unfortunately there was not enough time to go back and redo the testing, which also highlights how long it takes to brute force the password if implemented as per SRP 6a. The Python script is called crackSRPpassword.py and accepts arguments only in the following syntax, remembering v, s, and N are in the 0x hex format:

```
python crackSRPpassword.py [v] [PasswordFilePath] [s] [I] [N]
```

Listing 2: crackSRPpassword.py Syntax

4. Testing and Evaluation

These tests were conducted using:

- Host machine is a Dell Inspiron 7577 upgraded to 32GB of memory running Windows 10 Home x64
- Testing was conducted on a Ubuntu 18.04 virtual machine allocated two CPUs, two cores, and 8GB of memory
- Python version is 2.7.16

The test for vectors used were:

- Time in seconds to brute force an SRP password hash algorithm against prime number
- Hashes used were SHA1, SHA224, SHA256, SHA384, and SHA512 using Python module Hashlib [11]
- Prime numbers were 1024, 2048, 4096, and 8192 bits long taken from pysrp [8]
- Password used was "290387" taken from the 50000 line of PwnedPasswordTop100k.txt [6]. This was due to 240 passwords not in UTF-8 format which is required, password list can be found in the github repositories [12]. Any times in the result table should be doubled to estimate testing 100,000 passwords.

The results from this testing can be found in Table 2:

Table 2: Hash Algorithm vs Prime Number time to brute force

	1024	2048	4096	8192
SHA1	23.848565	81.0186735	294.32315	1079.425724
SHA224	36.128835	113.699608	421.862764	1547.362649
SHA256	42.089039	136.957380	517.235304	1754.455684
SHA384	63.37954	207.148031	771.110963	2725.173377
SHA512	84.608909	279.409642	1055.104418	3659.804495

Notes. Time is in seconds and rounded to six decimal places

As you can see from the data (Table 2), when you suspect a user may have a weak password it won't take an attacker long to brute force the password if they are in a known password list, even with a slow Python Script when compared to hashcat [13] which computes millions of hashes per second (H/s). Table 3 displays the H/s the Python code achieved during the testing.

Table 3: Hash Algorithm vs Prime Number Hashes per Second

	1024	2048	4096	8192
SHA1	2161	617	170	46
SHA224	1384	440	119	32
SHA256	1188	365	97	28
SHA384	789	241	65	18
SHA512	591	179	47	14

Notes. Value is hashes per second.

Even though it is much slower than hashcat, hashcat does not have an SRP specific algorithm. It does however have a 1Password algorithm, which may be used for future testing. The Python code could be made more efficient by adding multi-threading and segmenting the password file into batches to be served to each thread as it becomes available.

4.1. Hash Vs Prime Number

Looking away from the code the results show that if you have to make a choice between a stronger hashing SHA algorithm or a Prime number with a longer bit length, you should go with the larger prime number as proven in figure 3. The left side of Figure 3 shows the time percentage difference in time it took to authenticate credentials. The right side shows the percentage difference in time it took to brute force the password. The orange segment represents percentage difference between a 1024 bit length and a 8192 bit length prime number (Figure3).

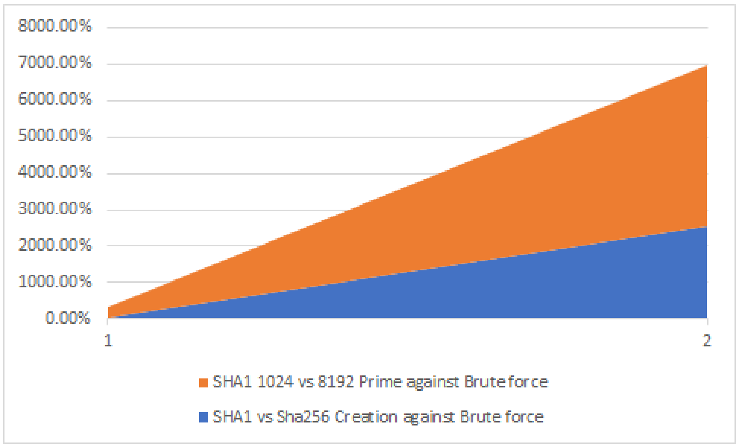

Figure 3: Hash Vs Prime number time to Brute force

There is one exception. Comparing the authentication compute times from figure 1 for SHA512 with a N of 4096 and SHA1 with an N of 8192; SHA1 takes 80% longer. Figure 3 shows that for and 80% compute saving you can use SHA512 with an N of 4096 and the attacker will only gain an extra H/s.

5. Conclusions

This paper has shown the importance of implementing the SRP 6a correctly and has highlighted the additional steps to be taken, such as 1Password [9]. Every user should have a unique s which is combined with their I as well as their P. The strongest hash available should then be used to give $x = H(s, I, P)$. A large prime number N should then be used to give $v = g^x$. This should prevent even the most common passwords from producing the same v which, in turn, will not highlight weak passwords to an attacker.

5.1. Further work

Testing still needs to be completed with the 8192 bit prime number to see how it compares against the other hashing algorithms. Testing also needs to be carried out using the PBKDF2-HMAC-SHA256 hashing algorithm with the 100,000 iterations to see what the time to create vs time to brute force comparison would be. Finally, faster testing could also be achieved by adding multi-threading to the code to increase H/s.

6. Bibliography

References

[1] J. Ding, S. Alsayigh, J. Lancrenon, R. Saraswathy, and M. Snook, “Provably secure password authenticated key exchange based on rlwe for the

post-quantum world," in *Cryptographers' Track at the RSA Conference*. Springer, 2017, pp. 183–204.

[2] T. D. Wu *et al.*, "The secure remote password protocol." in *NDSS*, vol. 98. Citeseer, 1998, pp. 97–111.

[3] T. Wu, "The srp authentication and key exchange system," Internet Requests for Comments, RFC Editor, RFC 2945, September 2000.

[4] J. M. Lopez Becerra, P. Ryan, P. Sala, and M. Skrobot, "An offline dictionary attack against zkpake protocol," 2018.

[5] S. University, "Srp protocol design." [Online]. Available: http://srp.stanford.edu/design.html

[6] NCSC, "Pwnedpasswordtop100k.txt," April 2019. [Online]. Available: https://www.ncsc.gov.uk/static-assets/documents/PwnedPasswordTop100k.txt

[7] D. Taylor, T. Wu, Cisco, N. Mavrogiannopoulos, and T. Perrin, "Using the secure remote password (srp) protocol for tls authentication," Nov 2007. [Online]. Available: https://tools.ietf.org/html/rfc5054#section-2.7

[8] T. Cocagne, "Python implementation of the secure remote password protocol (srp)," May 2018. [Online]. Available: https://github.com/cocagne/pysrp

[9] 1Password, "1password security design," January 2019. [Online]. Available: https://1password.com/files/1Password%20for%20Teams%20White%20Paper.pdf

[10] B. Buchanan, "Secure remote password protocol," May 2016. [Online]. Available: https://asecuritysite.com/encryption/srp

[11] ——, "hashlib — secure hashes and message digests," April 2019. [Online]. Available: https://docs.python.org/2/library/hashlib.html

[12] T. Mee, "Uni/esecuritycw," May 2019. [Online]. Available: https://github.com/Terrizmo/Uni.git

[13] P. Recovery, "Sha256 hash cracking," 2016. [Online]. Available: https://passwordrecovery.io/sha256/