

Password Authentication

by Henrick Hellström, StreamSec, 2004. All rights reserved.

First Revision with added footnotes, December 28, 2004

Password authentication is used in many systems for authenticating the identity of one or both peers of a connection. Most schemes that are used today are trivially insecure, even if they employ cryptographic algorithms such as hash functions and encryption functions. This paper will help you spot some common weaknesses and design more secure password authentication schemes using symmetric cryptographic techniques.

General Advice

Each password based authentication system is, without exceptions, vulnerable to certain attacks if you use weak passwords. Conversely, using a strong password, or rather pass phrase, will not do you any good if you use it for weak or seriously compromised authentication systems. Some general advice are:

- Use passwords and pass phrases you can remember. Having to write your password down, or store it in a file somewhere, is rarely a good thing.
- Use long pass phrases with several sentences rather than short obscure passwords with non-alphabetic characters. Remembering a secure password with 20 letters including digits and special characters is for most people a lot harder than remembering a 128 letter pass phrase consisting only of intelligible sentences.
- Use the same favorite pass phrase for each service you know employs a secure password authentication scheme. Having to remember a multitude of passwords for different services is bad. You should use your brain for something more productive.
- At the same time, you **MUST** remember that most services unfortunately do not employ secure password authentication schemes. If you use the same pass phrase for both service A and service B, your access to service A might become compromised if service B is or becomes compromised. If both services had used secure password authentication schemes this would not have been a concern.

The Problem

Alice wants to prove to someone she believes to be Bob that she is Alice. Alice wants to do so without disclosing any secret information to Eve (who is eavesdropping on the conversation) or Mallory (who has a tendency to impersonate Bob and Alice in order to trick them into revealing their secrets). Furthermore, Alice wants to use the same method of authentication for proving her identity to Sue, and she don't want to put her own, Bob's or Sue's security at risk in case either Bob or Sue becomes compromised.

Cryptographic Primitives

Hash Functions

A Cryptographically Secure Hash Function will transform input of any length to a fixed size output code, known as the Digest of the input. The hash function has the following properties:

1. It is computationally infeasible to find any data that hashes to a given arbitrary digest. In other words a Cryptographically Secure Hash Function can not easily be inverted but is computationally One-Way. This property is also known as Primary Preimage Resistance.
2. It is computationally infeasible to find two different inputs that hash to the same digest value. A

Cryptographically Secure Hash Function is Collision Free. This property is also known as Secondary Preimage Resistance.¹

As a consequence of these two properties a hash function can be used for the following purpose:

(*) Let T_A and T_B be two inputs and let D_A and D_B be the respective digest values of these two inputs. If $D_A = D_B$ then it is secure to infer that $T_A = T_B$.

StreamSec Tools Usage

The following code calculates the SHA-1 digest of the contents of the AnsiString aText. The SHA-1 digest is a 20 character string possibly consisting of non-printable 8-bit characters.

```
uses
  stSecUtils, stSHA1;
...
lDigest := DigestString(haSHA1, aText);
```

Example 1 (Fair but non-secure usage)

Alice sends the digest D_A of her secret password T_A to Bob. Bob looks up Alice in his user datatable, gets the stored value D_A' and verifies that $D_A = D_A'$. The actual value of T_A is not revealed during this process.

Example 2 (Attack)

Eve apprehends the value D_A sent by Alice in Example 1. Eve connects to Bob and sends D_A . Bob is tricked in to believing that Eve is Alice. This is known as a Replay Attack.

Example 3 (Attack)

Harry registers a large number of personalities with Bob's service. Later Harry is able to steal Bob's user datatable. Harry performs a query on the datatable and finds out that the value D_A for Alice is identical to the value D_B for one of Harry's fake personalities. Harry is able to infer that Alice's secret password T_A is identical to the password T_B for his corresponding fake personality.

Example 4 (Limitation)

It is not uncommon that distributors of Open Source software are publishing the MD5 digest of each file together with the file itself. This scheme provides very little security due to the general properties of hash functions. If you are given a pair $\langle T_A, D_A \rangle$ you are able to verify that D_A is in fact a valid digest of T_A , but there is no way for you to verify that T_A and D_A have not both been replaced by someone other than the distributor. Anyone is able to generate a valid MD5 digest of anything.

Keyed Hash Functions

A keyed hash function is a hash function with the additional property that a valid digest (also known as a Message Authentication Code or MAC) can only be calculated by someone who knows a specific secret key. The most common type of Keyed Hash Function is HMAC. The HMAC function works with all MD4 type hash functions, such as MD5 and the RipeMD and SHA family of hash functions, and operates by hashing the input twice and mixing in the key material with

¹ Collision Resistance entails Secondary Preimage Resistance, but the concepts are not trivially equivalent. More precisely, a hash function is collision free if and only if it has secondary preimage resistance for **any** input message. This distinction should be made, since collisions have been found in e.g. MD5 for **some** very specific inputs, but not a method that will find a secondary preimage for **any** input. Still, I would argue that Secondary Preimage Resistance entails that **no** secondary preimage can be found for **any** primary preimage, which means that the concepts are in effect equivalent. In other words: If you can find a secondary preimage T' for a primary preimage T , then T, T' is a collision. Conversely, if you have found a collision T, T' , then you have found a secondary preimage T' for the primary preimage T .

different paddings each time.

Keyed Hash Functions use symmetric keys, which means that both the sender and the recipient of a message must establish the same key.

(*) Let T_A and T_B be two inputs, K_A and K_B two keys and let M_A and M_B be the respective MAC values of these two inputs. If $M_A = M_B$ then it is secure to infer that both $T_A = T_B$ and $K_A = K_B$.²

(**) Let T_A an input, K a key and let M_A be the MAC value of that input. Given only T_A and M_A it is computationally infeasible to find a second input T_B and the MAC value M_B of that input under the key K .³

StreamSec Tools Usage

The following code calculates the HMAC-SHA-1 value of the contents of the AnsiString aText. The MAC is at most an aLength character string (at most the same length as the digest length of the underlying hash function) possibly consisting of non-printable 8-bit characters.

```
uses
    stSecUtils, stSHA1;
...
lMAC := HMACString(haSHA1, aKey, aLength, aText);
```

Example 5 (Fair but non-secure usage)

Alice sends the MAC M_{K_A} of the string 'Alice' and her secret password T_A as key to Bob. Bob looks up Alice in his user datatable, gets the stored value M_{K_A}' and verifies that $M_{K_A} = M_{K_A}'$. The actual value of T_A is not revealed during this process.

Example 6 (Attack)

Eve apprehends the value M_{K_A} sent by Alice in Example 1. Eve connects to Bob and sends M_{K_A} . Bob is tricked in to believing that Eve is Alice.

Example 7 (Failed Attack)

Harry registers a large number of personalities with Bob's service. Later Harry is able to steal Bob's user datatable. Harry performs a query on the datatable but is unable to find any MAC values that are identical, since the HMAC is collision free and Bob enforces the rule that the usernames must be unique.

Example 8 (Limitation)

Using a Keyed Hash Function for proving integrity and authenticity of public files is rarely an option.

Firstly, a HMAC uses a symmetric key. This means that both the sender and the recipient are able to generate a valid MAC of any given input text. If Alice, Bob and Carol all share the same secret HMAC key, there is no way for Alice to tell if a given MAC value was calculated by Bob or by Carol.

Secondly, a keyed hash function will not provide any security benefits over a regular (unkeyed) hash function unless the key is kept secret. If you publish a pair $\langle T_A, M_{K_A} \rangle$ no one who does not know the

2 This property entails that the HMAC construct can be used as a Pairing Operator for the underlying hash function. Producing a unique digest for a pair of values is a non-trivial problem, in so far that hashing the concatenation of two inputs will not produce a digest that is unique for that pair. For example, given the strings U, V, W , note that the hash of the concatenation of the pair $(U+V), W$ is equal to the hash of the concatenation of the pair $U, (V+W)$.

3 This criterion has been added in the first revision, since it better reflects the way a keyed hash function is typically used, i.e. as a cryptographically secure message authentication code. Note that this property is different from the properties of a cryptographically secure hash function.

key K will be able to verify that M_{KA} is in fact a valid MAC of T_A .

Other Concepts

Salt and Nonce

The distinction between a Salt and a Nonce is floating, but is mentioned here for clarity:

- Both are generated in a way that is designed to make them unique within context, e.g. by a Cryptographically Secure Random Generator such as Yarrow.
- Both are used in such way that they do not have to be kept secret but can be transmitted over unprotected channels.
- Both are typically mixed with secret material during some sort of hash operation.
- A Salt is typically used for generating a Verifier value that consists of the hash of the Salt and the Password. The Salt and the Verifier are stored together at the recipient (server) side for the life time of the password. If not random the Salt could be the concatenation of the recipient username and the sender username.
- A salt is more precisely typically used for making password hashes stored in server side user datatables unique. Confer Example 4 and Example 8 above.
- A Nonce is typically used only once for a single session handshake or for a single message. If it is not generated at random it could be a time stamp or an auto incremented persistent counter. Nonces are typically used in Challenge-Response protocols to thwart attacks of the kind described in Example 2 and Example 6 above.

Challenge-Response Protocols

A common way of avoiding some of the attacks outlined in the previous sections is to use Challenge-Response Protocols. These protocols mix in random material with the passwords. Done right this makes it impossible to generate valid protocol messages without the secret information that only the legitimate parties are assumed to possess.

These protocols will only output a binary Yes or No to the question: Were the incoming protocol messages authentic? More elaborate protocols will in addition output a Session Key that is used for protecting the confidentiality and message integrity of the following bulk messages. The security benefits of using such protocols should not be underestimated, since they entail that the confidentiality and integrity of the bulk contents is closely linked to the Entity Authentication established during the authentication phase of the protocol.

Challenge-Response protocols are traditionally designed to be either Two-Pass or Three-Pass. Two-Pass Challenge-Response protocols are used for authenticating the client to the server but not the server to the client. This section outlines two Three-Pass protocols that will authenticate both the client and the server to each other.

Protocol #1

This protocol is designed for situations where the speed of the network is the biggest bottle neck.

Preparations:

Each client entity Carol registers herself with the server entity Sue. Either Carol or Sue generates the following values:

$\text{Salt}[\text{Carol}] := \text{HMACString}(\text{haSHA1}, \text{Realname}[\text{Carol}], 20, \text{Realname}[\text{Sue}]);$

Verifier[Carol] := HMACString(haSHA1,Salt[Carol],20>Password[Carol]);⁴

Sue registers Realname[Carol] and Verifier[Carol].

- It is not important for the operation of the actual protocol if Carol or Sue generates the Verifier [Carol] value, since Sue will not use Password[Carol] at any point during the actual protocol. It is however important that the values that are sent between Carol and Sue during the registration phase are guaranteed to be both confidential and authentic. Furthermore, Carol should send Verifier[Carol] rather than Password[Carol] if she intends to reuse the same password for other purposes. It is computationally infeasible to derive Password[Carol] from Verifier[Carol].
- The value of Salt[Carol] must be unique. In particular, Realname[Sue] should be set to any information that will fully identify the particular service denoted “Sue” here, such as a full URI including protocol, hostname and path, possibly mixed with static information from Sue's Server Hello message. Alternatively, Salt[Carol] could be generated at random, but that would require that Salt[Carol] is stored by Sue and transmitted back to Carol during the protocol. It is assumed that if Salt[Carol] is generated according to the definition above, both Carol and Sue can independently regenerate the value at any later time.
- Sue **must** keep the Verifier[Carol] value confidential. It will typically be stored in a user datatable, and if this datatable is compromised each user will have to generate new Verifier values. It is **important** to note that it is impossible to generate new Verifier values if neither of the values Realname[Sue], Realname[Carol] nor Password[Carol] is changed, since Verifier [Carol] is a deterministic function of these values. Sue **must** consequently change Realname [Sue] if the functions described above are used and the user datatable is compromised. An alternative is to use randomly generated Salt values.

Steps and Messages:

1. Sue generates a random nonce Nonce_S.
2. **Sue -> Carol:** Nonce_S
3. Carol generates a random Nonce_C.
4. Carol calculates Verifier[Carol] as described above.
5. Carol calculates Temp1 := HMACString(haSHA1,Verifier[Carol],20,Nonce_S);
6. Carol calculates Response_C := HMACString(haSHA1,Temp1,20,Nonce_C);
7. **Carol -> Sue:** Realname[Carol], Nonce_C, Response_C
8. Sue looks up Realname[Carol] in the user datatable and gets Verifier[Carol].
9. Sue calculates Response_C as described above and compares it with the value received from Carol. If the values do not match Sue aborts, resulting in the protocol output “No”.
10. Sue calculates Temp2 := HMACString(haSHA1,Verifier[Carol],20,Nonce_C);
11. Sue calculates Response_S := HMACString(haSHA1,Temp2,20,Nonce_S);

4 These lines turned to not be completely uncontroversial. It should be noted that a public value Salt[Carol] is passed as the key input to the HMAC function, while the secret value Password[Carol] is passed as the text input, which is not coherent with the way a keyed hash function is typically used. The reason this line looks the way it does, is that the HMAC function in this case is really only used as a secure pairing operator for the underlying hash function. Since the first element Salt[Carol] of the input pair has a fixed length, it would be equally secure to simply use:

Verifier[Carol] := DigestString(haSHA1,Salt[Carol] + Password[Carol]);

Please note that DigestString can be securely used instead of HMACString because the Salt[Carol] value is computed internally **and** has a fixed length. There is nothing wrong with using HMACString instead, except that it might appear confusing for someone who would expect the key input to a HMAC function to be secret.

12. **Sue -> Carol:** Response_S

13. Carol calculates Response_S as described above and compares it with the value received from Sue. If the values do not match Carol aborts, resulting in the protocol output “No”.

14. Output “Yes”.

Protocol #2

This protocol is a variation of Protocol #1 with the difference that the Salt[Carol] value is generated randomly and is stored in Sue's user datatable. The implications of this have already been discussed.

Steps and Messages:

1. Carol generates a random Nonce_C.

2. **Carol -> Sue:** Realname[Carol], Nonce_C

3. Sue looks up Realname[Carol] in the user datatable and gets Salt[Carol], Verifier[Carol].

4. Sue generates a random value Challenge

5. Sue calculates Nonce_S := HMACString(haSHA1,Challenge,20,Realname[Sue]);⁵

6. Sue calculates Temp1 := HMACString(haSHA1,Verifier[Carol],20,Nonce_C);

7. Sue calculates Response_S := HMACString(haSHA1,Temp1,Nonce_S);

8. **Sue -> Carol:** Challenge, Salt[Carol], Response_S

9. Carol calculates Verifier[Carol] := HMACString(haSHA1,Salt[Carol],20>Password[Carol]);

10. Carol calculates Nonce_S as described above.

11. Carol calculates Response_S as described above and compares it with the value received from Sue. If the values do not match Carol aborts, resulting in the protocol output “No”.

12. Carol calculates Temp2 := HMACString(haSHA1,Verifier[Carol],20,Nonce_S);

13. Carol calculates Response_C := HMACString(haSHA1,Temp2,20,Nonce_C);

14. **Carol -> Sue:** Response_C

15. Sue calculates Response_C as described above and compares it with the value received from Carol. If the values do not match Sue aborts, resulting in the protocol output “No”.

16. Output “Yes”.

Other Considerations

A common characteristic of both Protocol #1 and Protocol #2 is that the exclusive usage of symmetric cryptographic primitives (HMAC) makes the user datatable extremely sensitive to compromise. The protocols will **only** provide entity authentication if Sue manages to keep the user datatable both confidential and authentic. Using a Salt in the calculation of the Verifier values **will** protect the confidentiality of the passwords, but it will **not** prevent an attacker who obtains the Verifier values from simulating the steps and messages of Carol in either of Protocol #1 or Protocol #2. If such protection is required a different protocol that employs Asymmetric Cryptographic Primitives, such as SRP, should be used.

Another consideration is that both Protocol #1 and Protocol #2 will only output a binary Yes or No, which means that the output of the protocol is not necessarily linked to the confidentiality or the

⁵ Confer the prior footnotes. In this case HMACString is used only as a secure pairing operator. It is equally secure to use DigestString on the concatenation of Challenge + Realname[Sue], given that the length of Challenge is a constant and that Carol validates that Challenge has the predefined length before computing Nonce_S.

integrity of the bulk contents of the session. In particular, it should be noted that the Nonces that are used as part of the protocols will protect against Replay Attacks of the kind described in Example 2 and Example 6, but they will not protect against Man-In-The-Middle attacks:

Example 9 (Attack)

This is an example of how a Man-In-The-Middle (Mallory) can attack Protocol #2. There exists a corresponding attack against Protocol #1.

1. Mallory tricks Carol into believing that Mallory is Sue. The difficulty of this deceptions depends on a vast number of factors, such as the integrity of the network as well as possible implementation flaws in the mechanism Carol uses for associating Realname[Sue] with a particular service.
2. **Carol -> Mallory:** Realname[Carol], Nonce_C
3. **Mallory -> Sue:** Realname[Carol], Nonce_C
4. **Sue -> Mallory:** Challenge, Salt[Carol], Response_S
5. **Mallory -> Carol:** Challenge, Salt[Carol], Response_S
6. **Carol -> Mallory:** Response_C
7. **Mallory -> Sue:** Response_C
8. Sue outputs "True". Carol outputs "True".

The last step might entail that Carol thinks she has verified that Mallory is Sue, or that Sue thinks she has verified that Mallory is Carol. It is important to note that neither Protocol #1 nor Protocol #2 will give such guarantees. They will **only** prove that the messages originated from the authenticated entity, but **not** that this entity is necessarily identical to the other peer. Also note that this is not necessarily a bad thing (think Proxy, Middle-Tier and Load Balancing), but that the security implications often make it extremely hard to use these protocols appropriately.

Asymmetric Challenge-Response Protocols

This section outlines a single protocol that is outlined after the Secure Remote Password authentication protocol (SRP) by Thomas Wu of Stanford University. The protocol is relatively complex but provides a range of security services that makes it interesting to invest the time in implementing it. Quoted from the original paper:

- An attacker with neither the user's password nor the host's password file cannot mount a dictionary attack on the password. Mutual authentication is achieved in this scenario.
- An attacker who captures the host's password file cannot directly compromise user-to-host authentication and gain access to the host without an expensive dictionary search.
- An attacker who compromises the host does not obtain the the password from a legitimate authentication attempt.
- An attacker who captures the session key cannot use it to mount a dictionary attack on the password.
- An attacker who captures the user's password cannot use it to compromise the session keys of past sessions.

Example

Below is the client side SRP code, taken from the SRP demo in the Variants folder

```
var
  lA, lB, lX, lV, lS: Variant;
  lAPriv: Variant;
  lSalt: string;
```

```

lK: OctetString;
lM1, lM2: string;
begin
Connect;
// a,A is the client ephemeral key, Diffie-Hellman style
lAPriv := RandomPrivateKey;
lA := VarMPIIntegerExpMod(2,lAPriv,P);

// C -> S: user, A
Send('user',edtUsername.Text);
Send('A',lA);

// S - C: salt, B
lSalt := Read('salt');
lB := Read('B');
// B is the server public ephemeral key, Diffie-Hellman style, mixed
// with the client V value which is stored in the server user datatable
// together with salt and user.
// The client MUST abort if B = 0
if (lB mod P) = 0 then begin
    Error('failed authorization');
end;

// X is the client long term private key and is derived from the password
lX := HashPassword(lSalt,edtUsername.Text,edtPassword.Text);
// V is the client 'verifier' or long term public key.
// It is used in the SRP protocol in such way that it cannot be derived by
// an attacker from the protocol messages. This makes a dictionary attack
// against the user password as hard as solving the Discrete Log Problem
lV := VarMPIIntegerExpMod(2,lX,P);
// The 'actual' server public ephemeral key is B-V. It is raised by the client
// to a power (a+u*X) that also hides the value of X behind the hardness of
// the Discrete Log Problem. S is the shared secret output of this operation
lS := VarMPIIntegerExpMod(lB-lV,
                        lAPriv + lX * CalculateU(lB),
                        P);
// K is the shared key that is derived from the shared secret S
lK := SHA_Interleave(lS);
// M1 is the client finished message. It can only be calculated by the client
// and by the server. The purpose of this message is to prove that the client
// derived the same value of K as the server, and hence that the client knows
// the password
lM1 := FinishHashClient(edtUserName.Text,
                        lSalt,
                        lA,
                        lB,
                        lK);

// C -> S: M1
Send('M1',lM1);

// S -> C: M2
lM2 := Read('M2');
// M2 is the server finished message. It can only be calculated by the server
// and by the client. The purpose of this message is to prove that the server
// derived the same value of K as the client, and hence that the server knows
// the V verifier value

if lM2 <> FinishHashServer(lA,lM1,lK) then begin
    Error('failed authorization');
end else
    SetKey(lK);

```