Authentication: A Primer

Burton Rosenberg

Last Update: May 4, 2019

Version 3: April 8, 2011
Version 2: April 21, 2003
Version 1: December 30, 2002

Contents

1 Password Authentication

Authentication is the convincing of another that you are who you say you are. The most usual procedure for authentication is the presentation of a password. The password syllogism is:

```
Only Socretes knows the password
This man knows the password
Therefore this man is Socretes
```

In practice proof of knowledge of the password is provided by revealing it, generally typing it into the password box of the log-on screen. So not only Socretes knows the password, so does the authenticator. Or if the authenticator didn't know the password prior to authentication, it does so afterwards.

In the electronic version of this interchange, two other concerns arise. Since the authentication belongs to the near end of a communication channel, we must assure that the other end of the channel is not diverted after authentication; and we must assure that the channel is confidential, so no third party can eavesdrop and learn the password. Finally, the assurance of authentication is only a probability. Plato can, with a certain probability, guess Socretes' password, and then be accepted as Socretes.

$$\text{Socretes} \overset{\text{password}}{\longrightarrow} \text{Aristotle}$$

Figure 1: Password authentication

Whatever its deficiencies, password presentation remains a popular authentication mechanism. A variant of the password is the PIN, a Personal Identifying Number, which is a short password drawn from the restricted alphabet of numeric characters. In applying either of these mechanisms, one should refer to the Federal Information Processing Standard FIPS 112 which makes recommendations and thereby establishes minimal standards for the use of PINs and passwords.

I am no lawyer, but FIPS 112 can be claimed as a standard of due care in the use of passwords. Security breaks which result in damage might be considered the software writer's negligence if security measures were not equal to accepted industry practice. It might be claimed that FIPS 112 provides a baseline for due care and accepted industry practice.

1.1 Password entropy

The authentication by password is only a probability. It is always possible for Plato to guess Socretes' password. The likelihood of this event should be quantified. If Socretes has n passwords to choose from, and he chooses one without bias, Plato's has a 1 in n chance of authenticating as Socretes purely by chance. The two key elements are the space of possibilities from which Socretes can choose, and his bias in making the choice.

Let $P = \{p_1, p_2, \ldots, p_n\}$ be the space of passwords, and by abuse of notation, p_i also be the probability that password p_i is chosen. The entropy of the space P is,

$$\mathcal{E}(P) = -\sum_{p_i \neq 0} p_i \log p_i.$$

Following Shannon, we take the base 2 logarithm so that the result is in units of "bits". If one of $256 = 2^8$ passwords is chosen uniformly at random, $\mathcal{E}(P) = 8$ bits, was we would hope.

Passwords are typically case sensitive and allow for punctuation characters, to increase the size of the password space, and should not be chosen as a common word, to reduce password bias.

The Shannon entropy of a distribution is the expected number of subset queries to identify an element, when the subsets have been optimally arranged. A Huffman encoding of the sets of passwords provides on example of optimally selected subsets: for each bit position make subsets of those passwords whose code has a one (or a zero) in that position.

This does not really capture what is wanted from a password. For instance, consider a password distribution χ of 16 passwords each chosen with probability 1/32 and a common password chosen with probability 1/2. The entropy is the same as eight passwords with the uniform distribution.

$$\mathcal{E}(\chi) = -1/2 \log 1/2 - 16(1/32 \log 1/32) = 3$$

This distribution, however, is intuitively less secure, as a single password is used for half of the accounts. This example is easily generalized to have arbitrarily high Shannon entropy, while still having a single password chosen with probability 1/2.

The entropy measure is the minimum of the expected number of guesses needed to identify a hidden element from a set. In measuring Shannon entropy, each guess can be a subset of the candidates and the query is whether that hidden element is in that subset. The correct model, however, is that each password is presented one by one, and the only information gained is whether or not that is the correct password. No partial credit for close answers.

It is a flaw in the security system if the password panel alows questions such as "does my password start with the string s". Answering such queries enable procedures for fast password extraction.

If each letter x in s is chosen from a set of k allowable characters, an n word password would be guessed in at most kn guesses, while the probability of any password should be as high as $1/k^n$, for an expected guessing time of $k^n/2$.

While it is obviously wrong to allow such queries, when we have the situation sufficiently abstracted to see clearly the flaw, this sort of attack does happen in real life. I personally have encountered and reported two such situations in important, deployed, code.

A measure of the expected number of guesses given a probability distribution of $p_1 \geq p_2 \geq \ldots p_n$ for the password $\sigma_1, \sigma_2, \ldots, \sigma_n$, respectively, could be,

$$\sum_i i\, p_i \quad \text{with} \quad p_1 \geq p_2 \geq \ldots p_n > 0.$$

This is known as the *guessing entropy*, and a good reference for its first definition is James L. Massey, *Guessing and Entropy*, Proc. IEEE Int. Symp. on Info. Th., 1994, p. 204.

A better measure would be related to the size of a set of passwords sufficient to guess correction some fraction α of the accounts. The *marginal guessing entropy* truncates the sum in the definition of the guessing entropy. Such measure are all the more relevant in judging the security of "naturally occurring passwords", such as security questions used in password recovery. In such cases, it is not possible to simply demand an almost uniform distribution across password choices. (See *What's in a Name? by J. Bonneau, M. Just and G. Matthews, Financial Cryptography '10, 2010*)

Question: What is the entropy of user selected passwords?

1.2 Password crackers

Automated tools exist for password guessing. Authentication systems thwart on-line application of such tools by limiting in some manner the number of logon retries and reporting logon failures to a security log. These automated tools are used for off-line attacks, when a password file has been collected by some means and is in the possession of the attacker. Crack is the classic Unix program for this. L0pht-crack provides the NT system administrator with the same amusement for his machines.

1.3 The Unix password hash

The subject requiring the authentication, the authenticator, must know the password. It is not usually stored on the authenticator without some encryption. If the database of encrypted passwords is stolen, the passwords are still safe until an attack is made against the encryption. Furthermore, the authentication decision is made by comparing encrypted versions of the password, so that passwords entered in the database are never again put clear, and perhaps it is infeasible to do so.

The traditional Unix password scheme [Morris and Thompson, 1979] passes the password through a modified DES encryption. Newer versions of Unix have replaced DES by other functions. FreeBSD uses an MD5 has of the password. Adoption of variants was delayed due to export considerations: strong encryption can be exported from the United States only under special license. FreeBSD solved this problem by distributing the code source with or without MD5 encryption. At this point in time, export controls have been eased and FreeBSD includes MD5 encryption in its base release.

The current FreeBSD implementation uses crypt to hash the password. Crypt has three modes of operation: traditional; extended; and modular. The the format of the password in /etc/passwd (or in the shadow file master.passwd) determines the mode. The following entry specifies modular, with code 1 representing MD5:

```
burt:$1$NtGQFzyQ$kOcs/5.VWulAk.S9euHqC1:542:15::0:0: \\
Burt Rosenberg:/home/burt:/bin/tcsh
```

An alternative is code 3 for blowfish. The code is between the first two dollar sign characters, and the salt is between the second two dollar sign characters. The remainder of the field is the hash.

The traditional unix format is invoked by 13 characters in the password field, not beginning with a dollar sign. The first two characters are salt and the remaining 11 characters are hash. Base-64 encoding is used, where 0-63 are given by the characters:

$$., /, 0, \ldots, 9, A, \ldots, Z, a, \ldots, z$$

(Note that this is not the same Base64 as used by MIME: `[A-Za-z0-9+/=]`.)

The 12 bits of salt are used to modify the DES algorithm, specifically, its E-box. The value zero is passed through the modified DES algorithm for 25 iterations, using the eight character password as the encryption key. The result has the salt prepended and a check is made if the password hash is recovered. If so, authentication succeeds. If not, authentication fails.

I can't and won't even try to describe the MD5 algorithm. The password Base64 encoding is also used. There are 8 characters of salt and 22 characters of hash result (6 bits per character).

Source code for FreeBSD implementations are found:

- `/usr/src/lib/libcrypt/crypt.c`
- `/usr/src/secure/lib/libcrypt/crypt-des.c`
- `/usr/src/lib/libcrypt/crypt-md5.c`

The advantages of the salt are the following,

1. It slows down brute force password search by a factor equal to the size of the salt space.
2. It prevents easy identification of shared passwords by users on the same or different machines.
3. It prevents the use of standard DES hardware from participating in a brute force search for passwords.

1.4 The Windows NT password hash

Windows NT also uses a password hash. In this case it is a simple MD4 of the password, yielding a 128-bit hash. However, NT uses this password hash in certain ways that makes it a complete replacement for the password. Under certain conditions, having the NT hash is as good as having the password, so their scheme does not truly increase security.

1.5 Trusted path, SAS, Trojans and other animals

There is a software and hardware path between the presentation of the password and the returned authentication decision. This path is trusted to be truthful by all components using the authentication decision. This trust is reasonable when the authentication system has a vested interest in providing accurate decisions for use by the other components. For example. the Unix authentication system shares with the user the password, but it would not share this password outside its system since the authentication system is part of the operating system that it is obligated to protect.

It is not easy to establish and maintain a trusted path. Network logon, for instance, requires the use of a network system which is outside the control of the authenticator. Even the integrity of this path for a local logon can be compromised. A Trojan is a program running on the machine which tricks the user into providing it with that user's password. It generally provides a false logon screen which the user believes to be genuine and collects the user's password.

An ultimate Trojan was an ATM placed in a shopping mall only for the purpose of collecting account numbers and corresponding PIN's. Users would commence a transaction, inserting their card and entering their PIN and the machine would then reject the request with some benign message such as "Machine out of order." The machine was collected after a day or two and the information gathered was extracted. [The Risks Digest, Vol 14, Issue 60, May 1993.]

In practice, the authenticator authenticates the user but the user relies on extra-technological clues to authenticate the authenticator. The user also depends upon the true service to support them in the case their judgment fails. Banks have worked hard to maintain user's trust in ATM's and credit cards in order to preserve these valuable businesses.

Windows NT has a Secure Action Sequence, SAS, also known as Control-Alt-Delete, to establish a trusted path between the local keyboard and the Gina, NT's front-end to the authentication subsystem. The NT operating system makes strong guarantees that the host will respond to an SAS by presenting an authentic logon screen. Without such a guarantee, passwords can be gathered either by Trojans or by software keyboard sniffers. A software keyboard sniffer was used by the FBI in the Scarfo case [US v. Scarfo]. Even with the SAS the path can be compromised, specifically between the wire connecting the keyboard to the host. Hardware keyboard sniffers are commercially available which will record all keystrokes as they pass from the keyboard into the host.

If the trusted path can be compromised, it is possible to achieve relay-attacks against even sophisticated authentication protocols. In the next section, challenge-response protocols will be described.

These rely on the proper completion of a multi-step question-and-answer session which reveal no (or little) reusable authentication data. A method to compromise such systems is for two imposters to work together, one pretending to be Socretes and presenting itself to Plato, and the other pretending to be Plato and presenting itself to Socretes. When the real Plato asks the fake Socretes a question, the fake Socretes transmits the question to the fake Plato who asks it of the real Socretes. The response is relayed back.

Anderson reports an actual incident of compromise of "friend-or-foe" systems employed by fighter planes to evaluate whether to attack or not. The attack plane asks "friend-or-foe" a challenge. If the victim cannot respond correctly, the attack plane attacks. Since an enemy victim cannot respond correctly, it relays the signal to another battle field, where it broadcasts it against a "friendly" plane. That response is then relayed back, so that the enemy plane can respond tricking the attack plane into thinking it is a friend.

Interesting counter-measures to relay attacks were given in *Multichannel protocols to prevent relay attacks, by F. Stajano, F–L Wong and B. Christianson, Financial Cryptography '10, 2010.* In this paper, an additional channel is required between the face-to-face partners that is unclonable. They give the example of a challenge being the serial number of a banknote, and the response being the encryption by secret key of the serial number and the presentation of the bank note. While the encrypted serial number can be relayed, the banknote cannot, and no two banknotes have the same serial number.

2 Challenge Response

Rather than present the password as proof of knowledge, the client can be asked to perform an action which implies knowledge of the password but which does not reveal the password. A Challenge-Response protocol asks the client to perform a calculation, a function of a randomly chosen number, the challenge, and a numeric version of the password, and to return the result of the calculation, the response.

The calculation should be such that the pair challenge—response assures knowledge of the password but does not reveal the password. The challenge must always be a new random number, else an attacker monitoring past sessions can present an old response to the old challenge and successfully authenticate.

$$\begin{tikzcd} \text{Socretes} \arrow[r, shift right, "\text{response}"'] & \text{Aristotle} \arrow[l, shift right, "\text{challenge}"'] \end{tikzcd}$$

Figure 2: Challenge response authentication

2.1 APOP

The Post Office Protocol (POP) allows network access to a user's email. Authentication is typically done using password authentication. The problem of password sniffing is particularly severe for POP for two reasons. First, POP is used for remote access while on travel, when the client is certainly unsure of the security of the network. Second, POP authenticates frequently, so the sniffer needn't be lucky to be looking at just the right moment. APOP uses challenge response to address these problems.

The banner of a server supporting APOP includes a unique challenge string. RFC 1939 gives the exact requirements for the challenge, but one acceptible format is `<process-ID.clock@hostname>`. The client appends a secret shared by client and server and takes the MD5 hash of the resulting string. It returns this in hexidecimal to the server as the second argument of the APOP command. The server also computes the MD5 hash using the shared secret and checks for agreement with the client's returned value.

```
S: +OK QPOP (version 2.3) at passaic.cs.miami.edu starting.  \\
   <19481.888520091@passaic.cs.miami.edu>
C: APOP burt 5916139512e4cc9433a24ba7d1e803f4
S: +OK burt has 1 message(s) (3065 octets).
```

Figure 3: APOP Session Example

2.2 MS-CHAPv1

MS-CHAP is the Microsoft Challenge-Response Authentication Protocol. Version 1, MS-CHAPv1 is documented by RFC 2433, and Version 2 y RFC 2759. The protocol is used in Microsoft's remote login protocols, such as PPP and PPTP. It is also similar to Microsoft authentication protocols for SMB, which are not documented. These protocols were reverse engineered by the SAMBA group, an Open Source project to port Windows file sharing and remote access to unix.

We discuss Version 1 in this section, and version 2 in the next.

The server sends the client a 8 byte nonce, to which the client responds with the encryption of the nonce three times, by three keys derived from the MD4 hash of the user's password. The 16-bytes MD4 hash has 5 bytes of zeros appended and then broken into three 7 byte pieces. A challenge C is presented by the server and the client responds with the response R,

$$\begin{aligned} P^h &= \mathcal{MD}4(P_u) \\ P^h_1|P^h_2|P^h_3 &= P^h|Z^5 \\ R &= \mathcal{DES}_{P^h_1}(C)|\mathcal{DES}_{P^h_2}(C)|\mathcal{DES}_{P^h_3}(C) \end{aligned}$$

The pair P^h, R is used as a shared secret for authentication and integrity checking for the remaining messages between client and server. A serial number is set to 0 for the client replay and incremented to 1 for each message sent client to server or server to client. The MAC is calculated,

$$\text{MAC(serial number, message)} = \mathcal{H}(P^h, R,\ \text{serial number},\ \text{message}\ , [U, R])\{8\}$$

and truncated to the hight order 8 bytes. Here, MD5 is the hash function. The final U, R pair is only used on the first message from client to server. All other hashes omit these elements.

2.3 MS-CHAPv2

An improved version of MS-CHAP, called version 2, was introduced. This description is taken from *Cryptanalysis of Microsoft's PPTP Authentication Extensions (MS-CHAPv2)*, by B. Schneier, Mudge and D. Wagner, September 1999.

$$\begin{array}{ccc} \text{Mudge} & \xleftarrow{SC} & \text{Gates} \\ \text{Mudge} & \xrightarrow{R,PC} & \text{Gates} \\ \text{Mudge} & \xleftarrow{A} & \text{Gates} \end{array}$$

Figure 4: MS-CHAPv2 summary

The server sends a 16-byte challenge SC. The client adds its own randomness by generating a random 16-byte Peer Authenticator Challenge PC. The two challenges are combined to an 8-byte challenge C,

$$C = \mathcal{SHA}(PC, SC, \text{client username})\{8\}$$

The first 8 bytes of the SHA-1 hash of the above information forms C. In response to the client receiving the 16-byte SC it sends the 24-byte response R and its 16-byte PC. R is formed as in MS-CHAPv1.

The protocol continues by authenticating the server. The server sends an Authenticator Response based on PC and the hashed password P^h,

$$\begin{aligned} A' &= \mathcal{SHA}(\mathcal{MD}4(P^h), R, \text{“Magic server to client constant”}) \\ A &= \mathcal{SHA}(A', C, \text{“Pad to make it do more than one iteration”}) \end{aligned}$$

The 20-bytes of A are sent as a response.

The improvements from v1 to v2 include:

1. The response is based on client randomness as well as server randomness, preventing a chosen plaintext attack.
2. The authentication is mutual, the server authenticates to the client in the last step.
3. Not shown, the LANMAN challenge response subprotocol was removed.

2.4 NTLM

NTLM is the Microsoft authentication protocol used with the SMB protocol, also known as CIFS, which is the Microsoft file and printer sharing technology. It is the successor of LANMAN, an older

Microsoft authentication protocol, and attempted to be backwards compatible with LANMAN. During protocol negotiation, the internal name is *ntlm 0.12*. The version number 0.12 has not been explained. NTLM was followed by version two, named NTLMv2, at which time the original was renamed NTLMv1.

There seems to be no official documentation of the protocol, however it has been reverse engineered by the SAMBA team and their documentation is definitive, in particular works by Luke Kenneth Casson Leighton. The cryptographic calculations are identical to that of MS-CHAP and are documented by RFC 2433 for v1 and RFC 2759 for v2.

2.5 NTLMv1

NTLMv1 is essentially MS-CHAPv1. The server authenticates the client by sending a 8-byte random number, the challenge. The client performs an operation involving the challenge and a secret shared between client and server, e.g. a password. The client returns the 24-byte result of the computation. In fact, in NTLMv1 two computations are made using two different shared secrets and two 24-byte results are returned. The server verifies that the client has computed the correct result, and from this infers possession of the secret, and hence the identity of the client.

The two secrets are:

- the LANMAN hash of the user's password and
- the NT Hash

Both these hashes produce 16-byte quantities. The NT Hash has been described in the section on MS-CHAP. The LANMAN hash is an old operation from the predecessor protocol, WIndows LAN Manager. Is is calculated by taking the user's password to all upper case and extending (or truncating) it to 14 characters. The each of the first and last 7 characters are taken as a 56-bit DES key to encrypt the 8-byte string "KGS!@#$%". The two resulting encrypted blocks are concatenated to get a 16-byte hash.

The 16-byte hashes are extended to 21 bytes by appending 5 bytes of zeros. The 21 bytes are separated in three 7 bytes quantities. Each of these 56 bit quantities is used as a key to DES encrypt the 64 bit challenge. The three encryptions of the challenge are reunited to form the 24-byte response. Both the response using the lanman hash and the the NT Hash are returned to the server.

```
C = 8-byte server challenge, random
K1 | K2 | K3 = NT-Hash | 5-bytes-0
R1 = DES(K1,C) | DES(K2,C) | DES(K3,C)
K4 | K5 | K6 = LM-Hash | 5-bytes-0
R2 = DES(K4,C) | DES(K5,C) | DES(K6,C)
response = R1 | R2
```

The server checks either of the responses to see if the calculation was done correctly. If so, it considers that the client knows the NT-Hash, or the LM-Hash, and continues the login.

2.6 NTLMv2

NTLMv2 is intended as a cryptographically strengthened replacement for NTLMv1. It consists of two different protocols, one which differs greatly from NTLMv1, and a second which shares much of NTLMv1's structure and is similar to MS-CHAPv2. Since there is no official documentation, it is hard to describe these algorithms since there are no official names. All three subprotocols can be called NTLMv2. However, careful reading of Microsoft documentation shows a tendency to call the first two NTLM2, and the third, differing protocol as NTLM2 Session.

NTLM2 sends two 16-byte responses to an 8-byte server challenge. The response is the HMAC-MD5 hash of the server challenge, a randomly generated client challenge, and a HMAC-MD5 hash of the user's password and other identifying information. The two responses differ in the format of the client challenge. The shorter response uses an 8-byte random value for this challenge. In order to verify the response, the server must receive as part of the response the client challenge. For this shorter reponse, the 8-byte client challenge appended to the 16-byte response makes a 24-byte package which is consistent with the 24-byte response format of the previous NTLMv1 protocol. In certain non-official documentation (e.g. the book DCE/RPC Over SMB by Leighton) this response is termed LMv2.

The second response sent by NTLM2 uses a variable length client challenge which includes (1) the current NT Time (multiple of 100 ns since January 1, 1601, whatever that means), (2) an 8-byte random value, (3) the domain name and (4) some standard format stuff. The response must include a copy of this client challenge, and is therefore variable length. In non-official documentation, this response is termed NTv2.

Both LMv2 and NTv2 hash the client and server challenge with a hash of the user's password and other identifying information. The exact formula is to begin with the NT Hash of NTLMv1, which is stored in the SAM, and continue to hash in, using HMAC-MD5, the username and domain name.

```
SC = 8-byte server challenge, random
CC = 8-byte client challenge, random
CC* = (X, time, CC, domain name)
v2-Hash = HMAC-MD5(NT-Hash, user name, domain name)
LMv2 = HMAC-MD5(v2-Hash, CS, CC)
NTv2 = HMAC-MD5(v2-Hash, CS, CC*)
response = LMv2 | CC | NTv2 | CC*
```

2.7 NTLMv2-Session

The NTLMv2 Session protocol is entirely different, being very similar to MS-CHAPv2. Its description is running around the Internet, and has been ported to SAMBA. Microsoft does not seem to have published any precise information about this protocol. This description is based on Eric Glass' ntlm page.

Briefly, the NTLMv1 algorithm is applied, except that a 8-byte client challenge is appended to the 8-byte server challenge and MD5 hashed. The least 8-byte half of the hash result is the challenge utilized in the NTLMv1 protocol. The client challenge is returned in one 24-byte slot of the response message, the 24-byte calculated response is returned in the other slot.

This is a strengthed form of NTLMv1 which maintains the ability to use existing Domain Controller infrastructure yet avoids a dictionary attack by a rogue server. For a fixed X, the server computes a table where location Y has value K such that $Y = DES_K(X)$. Without the client participating in the choice of challenge, the server can send X, look up response Y in the table and get K. This attack can be made practical using a space-time tradeoff called the *rainbow attack*.

However, existing NTLMv1 infrastructure allows that the challenge/response pair is not verified by the server, but sent to a Domain Controller for verification. Using NTLMv2 Session, this infrastructure continues to work if the server substitutes for the challenge the hash of the server and client challenges.

```
NTLMv1
    Client<-Server:  SC
    Client->Server:  H(P,SC)
    Server->DomCntl: H(P,SC)), SC
    Server<-DomCntl: yes or no

NTLMv2-Session
    Client<-Server:  SC
    Client->Server:  H(P,H'(SC,CC)), CC
    Server->DomCntl: H(P,H'(SC,CC)), H'(SC,CC)
    Server<-DomCntl: yes or no
```

2.8 NT Authentication in context

We have described the cryptography of NT Authentication, either MS-CHAP or NTLM. In this section we describe a bit more about the login process and how the NT-Hash is stored.

NT Authentication uses a module called MSV1_0.dll to process the challenge/response. One half of this module interacts with the user to get the password, the other half interacts with the SAM, the database containing the password hashes, to authenticate the user. Using Microsoft RPC, these two halves may be on different computers. Typically, in a domain situation, the user half is on the client machine and the SAM half is on the PDC, the primary domain controller.

This architecture makes little distinction between a local account and a domain account. For a local account the MSV1_0 uses the local SAM and does not need an RPC, it does an LPC (local procedure calle). For a domain account the MSV1_0 on the PDC is involved using an RPC.

The box appearing for the user's password is called the GINA. It is presented by the SAS. The SAS and GINA are tightly coupled so that it is difficult to circumvent this trusted path. The SAS switches desktops even, and presents the GINA in a secure desktop under Operating System control. The GINA collects authentication information and calls the LSA in the kernel (Local Security Administrator). The LSA then uses some authentication service to verify the password. In current NT, LSA uses MSV1_0, although this is modular.

There are actually two types of interactions between computers. Challenge/Response between client and PDC need not be cryptographically secure. However, authentication could require three computers, with the middle computer acting as a proxy between the client and the PDC. In this case an encrypted connection is established between client and the middle computer using a shared secret. Over this secure channel the password passes clear, and the middle computer completes the Challenge/Repsonse protocol for the client.

The shared secret is generated when a computer joins a domain and is automatically refereshed each week. The initial shared secret is derived in a completly public manner using the computer's name. The shared secret is stored in the local SAM. But manipulation the protections on the SAM it can be viewed. You usually need to use both regedt32 to widen permissions to see the SAM and the regedit to actually open the folders. Look for HKLM/Security/Policy/Secrets then something like Machine. There will be subkeys for CurrVal, OldVal, CupdTime, and so on, for the current secret value, old secret value and the update time.

3 One-time passwords

Note: This section draws heavily on Smith's book, see references.

A perfectly secure method of authentication is for the client to make use of a random list of passwords, shared with the server, and both client and server agree that each password will be used only once. A password's use renders it useless. Therefore an eavesdropper learns only useless information.

It is impractical to implement this scheme strictly. Rather than a list of truly random passwords, a pseudo-random generator replaces the list to generate new passwords as required. Breaking the pseudo-random generator breaks the one-time password scheme, so the pseudo-random generators are chosen with great care.

3.1 Lamport Hash and S/Key

The Lamport hash [Lamport 1981] uses a strong hash function to generate the list of passwords. Starting from a client secret p, the passwords are,

$$\begin{aligned} p_0 &= \mathcal{H}(p) \\ p_i &= \mathcal{H}(p_{i-1}), \;\; i > 0 \end{aligned}$$

where $\mathcal{H}$ is a suitably strong hash function. The passwords are used in order of descending i. Presented with an i, the client can quickly reconstruct p_i. The server, having in its database a p_j with $j > i$ can quickly hash forward to verify the password.

S/Key is an implementation of Lamport's scheme [Haller 94]. It has been ported to FreeBSD in the original form and as OPIE (Onetime Passwrods In Everything). Each prompt the user with a counter number and a seed. The seed as appended to a memorized password and hashed counter number of times. The result is a 64 bit value which is transformed into a sequence of six short English words.

Hitting return at the password prompt will make the password typing visible — to avoid typing the password wrong. Since it is one time, there is no security lost in this. The seed is used to make possible the use of the same user secret on various machines. The seed will be different for each machine so that the hash chain will be different on each machine.

Here is an S/Key example from FreeBSD, taken from the FreeBSD handbook.

```
% telnet example.com
Trying 10.0.0.1...
Connected to example.com
Escape character is '^]'.

FreeBSD/i386 (example.com) (ttypa)

login: <username>
s/key 97 fw13894
Password:
```

The cournter and seed are given. On the local machine, the onetime password is calculated:

```
% key 97 fw13894
Reminder - Do not use this program while logged in via telnet or rlogin.
Enter secret password:
WELD LIP ACTS ENDS ME HAAG
```

The password is transferred to the login prompt. In this example, we turn echo on:

```
login: <username>
s/key 97 fw13894
Password: <return to enable echo>
s/key 97 fw13894
Password [echo on]: WELD LIP ACTS ENDS ME HAAG
Last login: Tue Mar 21 11:56:41 from 10.0.0.2 ...
```

3.2 Counter, clock and PIN based One-time schemes

Another class of one-time password schemes are based on the hash of a shared secret combined with either a counter value or the time, or both. These implementations usually are small devices which produce the one-time password on a display when a button is pushed. Some companion machine loads the device with a random secret. To prevent unauthorized use of the device, a PIN might be included. The PIN works either by unlocking the device, or by becoming itself a part of the hash.

Device	*Company*	*Hash*	*Synchronization*
SecureID	RSA Security	secret, time, PIN	time window, drift following
SafeWord	Secure Computing	secret, counter	counter window
ActivCard	PC Dynamics	secret, counter, timer, PIN	partial reveal

Figure 5: Onetime password devices

The use of one-time password schemes has the problem of keeping client and server synchronized as to the next password to use. Counter based tokens might become unsynchronized by the user pushing the generation button several times. The system lets the user log in with with a counter value beyond the current with a fixed tolerence. The server can search forward over counter values for a password match. If synchronization is still unattainable (the client counter is too far ahead of the server counter) a sequence of two good passwords will be required.

Timer based schemes are implicitly synchronized except for timer drift. The server will attempt to track drift per device and compensate. A two consecutive good password sequence could be used to recover if the drift value is too great for security purposes.

ActivCard sends the least significant digit of the counter and timer as part of the password. In this way, synchronization is less of a problem.

3.3 OTP versus C/R

How do one-time passwords and Challenge-Response differ?

In a sense, the devices above are challenge-response with an implicit challenge: the time and counter value. The important similarity is that both client and server share a secret value, the partial base of the hash. However, these devices are "air-gapped" from the system, that is, they have no direct connection to the client computer. Certain forms of attack are therefore impossible. The challenge being implicit means that only half the amount of information has to be hand carried between device and client computer.

The Lamport scheme has a major difference in that the server does not have a compromisable secret. It has the disadvantage of requiring refresh after the hash chain is exhausted.

A true OTP would be a list of random numbers, and it would be information-theoretically impossible to guess the next valid password from previous passwords. The schemes described here are only computationally secure. However, a OTP would require a large shared secret — the entire list of passwords, and this might be hard to secure.

4 Indirect Authentication

Centralized user administration gives rise to authentication servers. Sun introduced NIS (originally known as Yellow Pages) to centralize the password database, as well as many other user and system configuration files using the same mechanism. Windows uses NTLM, the concept of NT Domains, and more recently , Active Directory. In fact, it was the highly unsuccessful Novell that was the master of this sort of thing.

The first indirect authentication scheme provides access to a single machine. Key Distribution Centers solve a larger problem, sometimes called single-logon. With a KDC the user authenticates to a central authentication server and can receive permission on many different network services using the central server.

4.1 RADIUS

Remote Authentication Dial In User Service, also known as RADIUS, see RFC 2865, C. Rigney et. al., consists of a client-server architecture in which the authenticating host contacts the RADIUS server, presenting the client's information in an *Access-Request* message, and waiting for either an *Access-Accept* or *Access-Reject* reply from the RADIUS server.

The protocol is packet-oriented, with three types of messages:

1. *Access-Request*, from client to server to verify authorization information.
2. *Access-Accept/Access-Reject*, from server to client, informing client of authentication decision.
3. *Access-Challenge*, from server to client, to send challenge information to the client, and expecting a subsequent Access-Request message responding to the challenge.

A share secret authenticates the Radius server to its clients. Each Access-Request packet includes a 16-byte Authenticator, chosen at random, and unique of the lifetime of the shared secret. Response packets must contain the correct Response Authenticator,

$$\begin{array}{rcl} \text{ReqAuth} & \in_R & \{\text{16-byte}\} \\ \text{RespAuth} & = & \mathcal{MD}5(\text{response data}, \text{ReqAuth}, \text{secret}) \end{array}$$

If the Access-Request packet includes a password, the password is encrypted by exclusive-or with a 128-bit value derived from the Request Authenticator and the secret,

$$\mathcal{E}(\text{password}) = \mathcal{MD}5(\text{secret}, \text{ReqAuth}) \oplus \text{password}$$

4.2 Needham-Schroeder

The basic network logon scheme is due to Needham and Schroeder [Needham 78]. Once the user is authenticated to the Key Distribution Server, the user can establish privileges for many network services.

The KDC is key server, and is maintained with the strictest security. It shares with each user a secret user key. For two users to authenticate, the KDC will generate a fresh session key and send the key to both parties, encrypted with the user's secret key. The users then mutual authenticate by demonstrating knowledge of the fresh session key. For engineering reasons, the KDC communicates with only one of the parties, the party requesting the connection. That party will forward KDC messages for the second party on behalf of the KDC.

$$\begin{array}{lcl}
\begin{array}{rcl} R_1 & \in_R & Z \\ M_1 & = & R_1, \mathcal{N}, \mathcal{S} \end{array} & \mathcal{N} \xrightarrow{M_1} \mathcal{KDC} & \\
 & \mathcal{N} \xleftarrow{M_2} \mathcal{KDC} & \begin{array}{rcl} K_{NS} & \in_R & Z \\ T & = & \mathcal{E}_S(K_{NS}, \mathcal{N}) \\ M_2 & = & \mathcal{E}_N(R_1, \mathcal{S}, K_{NS}, T) \end{array} \\
\begin{array}{rcl} R_2 & \in_R & Z \\ M_3 & = & \mathcal{E}_{K_{NS}}(R_2) \end{array} & \mathcal{N} \xrightarrow{T, M_3} \mathcal{S} & \\
 & \mathcal{N} \xleftarrow{M_4} \mathcal{S} & \begin{array}{rcl} R_3 & \in_R & Z \\ M_4 & = & \mathcal{E}_{K_{NS}}(R_2 - 1, R_3) \end{array} \\
M_5 = \mathcal{E}_{K_{NS}}(R_3 - 1) & \mathcal{N} \xrightarrow{M_5} \mathcal{S} &
\end{array}$$

Figure 6: Needham-Schroeder

The security reasoning is as follows:

1. M_1 makes no trusted assertions, as it could be sent by anyone to the KDC.
2. N believes that the KDC says $\langle K_{NS}, S \rangle$ because M_2 is encrypted by $\mathcal{E}_N$, which can only be done by N or the KDC.
3. N also believes that KDC said this recently, because R_1 is also included, and N believes R_1 to be recent.

4. N infers that an entity that knows K_{NS} must be N, S or the KDC, because the KDC says K_{NS} is fresh, and the protocol keeps K_{NS} secret to all other entities.
5. Note that T makes no trusted assertions to N, as it is indiscernible from random to N. Also, S makes no trust inferences that N says T. It only believes that KDC says T, because it is encrypted by $\mathcal{E}_S$.
6. S infers that the KDC asserts that an entity that knows K_{NS} must be N, S or the KDC, because N is in T, because T is encrypted by $\mathcal{E}_S$, and that the protocol keeps K_{NS} secret to all other entities.
7. N believes S is the counter party because S demonstrates to N that it knows R_2, and hence must know K_{NS}. And likewise with S, N and R_3.

4.3 BAN Logic and the failure of Needham-Schroeder

There is considerable concern over the ad hoc nature of reasoning about security protocols. BAN logic attempts to formalize this reasoning [Burrows et al, 1990]. This system has itself been criticized (find reference), for instance, in the incorrect assumption that confidentiality (encryption) implies authenticity. However, it has discovered a flaw in Needham-Schroeder.

BAN logic captures to protocol flaw of replay in the formalization of *freshness*. Suppose Alice says X to Bob. Bob then believes that Alice once believed X. If X is new or novel, then Bob will believe that Alice currently believes X. A nonce provides the necessary freshness to transform what Alice once said, for example, that this password was sufficient to authenticate, into what Alice still believes, that this password is sufficient to authenticate.

There is nothing in Needham-Schroeder to guarantee freshness of the ticket generated by the KDC. While the KDC will not create a ticket unless the KDC believes the encrypting key is sufficient for authentication, the ticket receiver has no reason to believe that the ticket was created recently, that is, that the KDC still believes that the encrypting key is good. Needham can bluff Schroeder into a conversation if ever it has been allowed to converse with Schroeder, unless Schroeder's own key is revoked.

4.4 Otway-Rees

Another protocol, similar to Needham-Schroeder is Otway-Rees [Otway 1987].

Question: find the original protocol and apply BAN suggestions to a modified version.

4.5 Kerberos

Kerberos is a network authentication protocol based on Needham-Schroeder. It has three main differences from Needham-Schroeder. First, the mutual authentication in the final steps of N-S is

$$\begin{array}{rcl} N_a, N_o & \in_R & Z \\ M_1 & = & \mathcal{E}_o(N_a, N_o, \mathcal{O}, \mathcal{R}) \end{array} \qquad \mathcal{O} \overset{N_a,\mathcal{O},\mathcal{R},M_1}{\longrightarrow} \mathcal{R}$$

$$\mathcal{KDC} \overset{M_1,M_2}{\longleftarrow} \mathcal{R} \qquad \begin{array}{rcl} N_r & \in_R & Z \\ M_1 & = & \mathcal{E}_r(N_a, N_r, \mathcal{O}, \mathcal{R}) \end{array}$$

$$\begin{array}{rcl} K & \in_R & Z \\ M_3 & = & \mathcal{E}_r(N_r, K) \\ M_4 & = & \mathcal{E}_o(N_o, K) \end{array} \qquad \mathcal{KDC} \overset{N_a,M_3,M_4}{\longrightarrow} \mathcal{R}$$

$$\mathcal{O} \overset{M_4}{\longleftarrow} \mathcal{R}$$

Figure 7: Otway-Rees

done using timestamps, rather than random nonces, and is accomplished in two steps rather than three. Second, the conversations between a client and the KDC begin with the assignment of a session key, which replaces the client key for the remainder of the session. This is done to reduce dictionary attacks on the shared key. It is expected that the shared key will be user derivable and therefore of small entropy. Third, so that the KDC does not itself have to remember the session key, a Ticket-Granting-Ticket is given to the client when the session key is generated. This TGT is for the KDC's benefit. Inside the TGT, encrypted by a private key to the KDC, is the session key. The KDC does not remember the session key, rather it requires the client to present the TGT. The KDC decrypts the TGT and is thereby reminded of the session key.

4.5.1 Authentication Server Request/Reply

The Kerberos protocol can be neatly broken down into three classes of Request-Reply. When the user first logs in, the user's workstation and the KDC authenticate the user, assign a session key, and create a TGT.

The request by Alice to the KDC is simply $\mathcal{A}$, the name of the user, Alice,

$$\text{AS-REQ} = \mathcal{A}$$

The KDC generates a fresh random number S_A to be used to encrypt communication between Alice and itself for a duration of this session, and a Ticket Granting Ticket including S_A for its own future use,

$$\text{TGT} = \mathcal{E}_X(\mathcal{A}, S_A)$$

where X is the KDC key and is only know to the KDC. The KDC send Alice a response with the TGT an the key S_A encrypted on A, the shared key between Alice and the KDC,

$$\text{AS-REP} = \mathcal{E}_A(S_A, \text{TGT})$$

4.5.2 Ticket Granting Service Request/Reply

Alice asks the KDC for a shared key with Bob. It presents the request with its TGT and an authenticator. The authenticator is the time encrypted with the session key. The KDC verifies the information, creates the key K_{AB} for Alice and Bob and returns it to alice as an encrypted (by the session key) packet including K_{AB} and K_{AB} encrypted by Bob's key, the ticket to forward to Bob.

The request is,

$$\text{TGS-REQ} = \mathcal{A}, \mathcal{B}, \text{TGT}, \mathcal{E}_{S_A}(t)$$

The KDC generates a fresh random number K_{AB} that will be the encryption key shared by Alice and Bob, and forms a ticket that only Bob can decrypt, including the ticket the name of Alice, $\mathcal{A}$,

$$T_{AB} = \mathcal{E}_B(\mathcal{A}, K_{AB})$$

The KDC then encrypts this ticket, the key K_{AB} and the Bob's name $\mathcal{B}$ under the session key S_A and sends it to Alice,

$$\text{TGS-REP} = \mathcal{E}_{S_A}(\mathcal{B}, K_{AB}, T_{AB})$$

The point of encrypting Bob's name is that the request to the KDC is somewhat weak, and it possible for an attacker to intercept the request and substitute, say, their own name for Bob's. This could possibly fool Alice into talking to the attacker thinking it is Bob. The inclusion of Bob's name, under encryption by S_A is the KDC attesting to the valid counter-party of the key K_{AB}.

The reasoning is likewise for the inclusion of Alice's name in T_{AB}.

4.5.3 Application Request/Reply

Alice sends Bob the ticket, and an authenticator, the time encrypted by K_{AB}. Bob responds with the time sent plus one encrypted by K_{AB}.

Alice tells Bob,

$$\text{AP-REQ} = T_{AB}, \mathcal{E}_{K_{AB}}(t)$$

and Bob responds,

$$\text{AP-REP} = \mathcal{E}_{K_{AB}}(t+1)$$

4.6 Key Agreement: Diffie-Hellman

The authentication protocols described rely on a secret shared between parties for both establishing identity (authentication) and as an encryption key for communication (confidentiality). These are separate issues. *Key agreement* is the act of establishing an encryption key between two parties for the purposes of confidentiality, not necessarily establishing authenticity.

Such protocols have a sense of magic. How can a key be agreed upon by two parties, over a public channel, if those parties have had no prior communication? What would distinguish the pair from an eavesdropper so that a key is exchanged between the parties but remains unknown to the eavesdropper that has monitored the entire conversation?

This can be done with *complexity theoretic security assurances* by calculations that mix private randomness of the two parties, the private randomness shared over the public channel under a special one-way function. The special property of the one-way function is that a holder of the result of the function can correctly make some calculations on the input to the function, without actually knowing the input. (This is one example of a larger discussion called homomorphic encryption, which allows very general computation to be done on values using only the encrypted presentation of those values.)

The Diffie-Hellman key agreement protocol depends on a complexity assumption, called the Diffie-Hellman assumption. Let p be a prime, and g a generator for Z/pZ^*. That is, for every $x \in [1, p-1]$ there exists a unique $y \in [0, p-2]$ such that $x = g^y \bmod p$. and y is said to be the discrete log of x to the base g in the integers mod p.

There are fast ways of exponentiating, but no known efficient way to take discrete logs. However, this depends somewhat on the prime p. Some primes (smooth primes) are weak: for these primes fast algorithms do exist. Also, given x it is possible to quickly tell if i is odd or even. Removing these cases, that it is hard to calculate i from x is the Discrete Log (DL) assumption. The Diffie-Hellman key agreement algorithm uses a related assumption.

Definition 1 (Diffie-Hellman Assumption:) *Given p and g as described, from $x_1 = g^{y_1}$ and $x_2 = g^{y_2}$ it is computational difficult to compute $x = g^{y_1 y_2}$ from x_1 and x_2 alone (all calculations are modulo p.)*

The DH assumption is that from g^a and g^b it is hard to calculate g^{ab}. If discrete logs are possible, then the DH assumption does not hold, because we can know even a and b. However, maybe there is some other way of getting g^{ab} that does not need to take the log of g^a or g^b. That is the subtle difference between the DH assumption and the DL assumption.

Alice and Bod want to agree on a key. Alice or Bob announce p and g. Then Alice chooses a random a and sends Bob g^a. Bob chooses a random b and sends Alice g^b. They both then compute $g^{ab} \pmod p$. Alice does this by taking Bob's g^b and raising it to the power of a, and Bob does this by taking Alice's g^a and raising it to the power of b.

The Diffie-Hellman key agreement protocol is secure if the DH assumption holds.

4.7 Encrypted Key Exchange (EKE)

The challenge response schemes previously described leak password information. Any server, wishing to compromise a client, has access to the input-output pair under the challenge-response function. From this, a brute force attack on the password can begin. A scheme by Bellovin and Merritt [Bellovin 1992] uses a Diffie-Hellman key exchange to mask the user's password. As far as I know of, there are no practical protocols of major importance that use this scheme.

To review Diffie-Hellman: a prime p and an generator g of F_p is made public knowledge by some trusted authority. Party B sends party M the value $g^{r_B} \bmod p$; party M sends party B the value $g^{r_M} \bmod p$, where r_B and r_M are randomly selected values, secret to B and M. Each party computes $g^{r_B r_M} \bmod p$. This value is now known to B and M but, under the D-H assumption, cannot be calculated by any other party, including a party who witnessed all other values.

Bellovin-Merritt performs D-H with the exchanged partial secrets encrypted by the shared password. They then confirm to each other that they know the shared secret.

$$
\begin{array}{lcl}
\begin{array}{rcl} r_B & \in_R & Z \\ M_1 & = & \mathcal{E}_{BM}(g^{r_B} \bmod p) \end{array} & \mathcal{B} \xrightarrow{M_1} \mathcal{M} & \\
 & \mathcal{B} \xleftarrow{M_2} \mathcal{M} & \begin{array}{rcl} r_M & \in_R & Z \\ K & = & (g^{r_B})^{r_M} \bmod p \\ M_2 & = & \mathcal{E}_{BM}(g^{r_M} \bmod p) \end{array} \\
\begin{array}{rcl} R_1 & \in_R & Z \\ K & = & (g^{r_M})^{r_B} \bmod p \\ M_3 & = & \mathcal{E}_K(R_1|0) \end{array} & \mathcal{B} \xrightarrow{M_3} \mathcal{M} & \\
 & \mathcal{B} \xleftarrow{R_1, M_4} \mathcal{M} & \begin{array}{rcl} R_2 & \in_R & Z \\ M_4 & = & \mathcal{E}_K(R_2|1) \end{array} \\
 & \mathcal{B} \xrightarrow{R_2} \mathcal{M} &
\end{array}
$$

Figure 8: Bellovin-Merritt

Because $\mathcal{E}_K(R_1|0)$ and R_1 are public, A dictionary attack can be attempted against $K = g^{r_B r_M} \bmod p$. But recovering $g^{r_B r_M} \bmod p$ tells nothing about $g^{r_B} \bmod p$ and $g^{r_M} \bmod p$, so a dictionary attack against the key BM has no place to start.

4.8 Central Authentication Service (CAS)

CAS is an indirect authentication mechanism invented at Yale University which uses standard HTTP capabilities. It makes use of standard HTTP mechanisms, particularly cookies, redirects and HTTP query strings, to send the user from the web application to an authentication server, for credential validation, and then back to the web application, having authenticated by valid credentials (i.e. username and password).

When the browser first contacts the web application, it is redirected to the CAS server's login method, carrying the name of the service that is requesting the login. This is done by placing *login* as the path part of the URL and the service name as the value of the *service* name of a GET query. E.g.,

https://cas.cs.miami.edu/login?service=https://web.cs.miami.edu/secret

Note that the service name is also a URL. This must be so because CAS will redirect back to the service after authentication. Note also the use of SSL for HTTP, indicated by the *https* protocol specification in the URL.

The server *cas.cs.miami.edu* will present an HTML form requesting username and password, with the submit action again to */login*. When the form is returned, if the credentials are correct, CAS will redirect the browser back to the URL given by the service, appending a *service ticket* to the URL as a GET query. E.g.,

https://web.cs.miami.edu/secret?ticket=ST-92834-m34Aa83f7a3f

The web application then contacts CAS with this service ticket to verify its validity. It does this by using the pathname */validate*, and provides as a GET query the service name and the ticket,

https://cas.cs.miami.edu/validate?service=https://web.cs.miami.edu/secret&ticket=ST-92...a3f

The response is either an the text YES and the username the was used in the login, or the text NO.

There is an additional mechanism for the browser to cache something similar to the Kerberos TGT, so that repeat visits to the CAS authentication server do not require credential validation. When the CAS server redirects back to the web application, using the service URL with the service ticket appended, it sets as a browser cookie a *ticket granting cookie*. Browser cookies are associated with URLs, and the browser sends any set cookies it has for a URL when it sends requests to that URL. In this case, when the ticket granting cookie is present, when the browser visits the */login* location, it will send the cookie. The CAS server can evaluate the cookie. If it is valid, rather than returning a login form, it returns a service ticket, with a redirect, immediately.

CAS 2.0 introduced *proxy tickets*, which allow web applications to use CAS service to request services of other web applications. A proxy ticket is not one-time use, a service ticket is. A proxy ticket is requested when the service ticket is validated. Along the service and ticket parameters, the HTTP request will encode a PGT Callback URL into the request. The CAS server will return, along with the information about the service ticket, a *proxy ticket IOU*, and will deliver the proxy ticket to the callback URL by an HTTP GET with parameters both the proxy ticket IOU and the proxy ticket.

The proxy ticket can then be verified using the */proxyValidate* path on the CAS server, and a valid proxy ticket will return the call back URL to which this ticket was supplied.

4.9 SAML and OpenID

To do.

5 Digital Signatures and the Public Key Infrastructure

5.1 Public key authentication

Authentication on the web requires the that parties with no prior conversations authenticate, if only asymmetrically. With the exception of some of the One Time Password schemes, the two parties share a secret. Establishment of that shared secret is by a prior and private conversation between the two parties.

Public key cryptography offers the ability for secure communication between parties that do not share a secret. A public key scheme produces from a source of randomness a key pair, one of which is made public, and the other is maintained private. The two keys are precisely mathematically related so that what is encrypted by the public key can be exactly decrypted by the private key. However, without knowledge of the private key, even with knowledge of the private key, decryption is unlikely.

$$m = D_{SK} E_{PK}(m)$$

It was noted that such public key encryption schemes give a way to publicly attest to the authenticity of a message, if the message were first decrypted by the secret key, and the decryption published alongside the message. Any possessor of the public key can then verify that the message is an authentic statement by the possessor of the secret key but encrypting the message, and comparing the result to the message.

$$\text{If } T = D_{SK}(m) \text{ then } E_{PK}(T) == m.$$

This leads to the following authentication algorithm. Suppose Alice wished to know that the counter party with whom she speaks is Bob. Assume Bob has a public/secret key pair, $\langle PK, SK \rangle$, and has published his public key, such that Alice can reliably obtain a copy.

Alice challenges Bob with a fresh random value R, to which Bob should respond with $T = D_{SK}(R)$. Alice then verifies $E_{PK}(T) == R$, and accepts the authenticity of Bob if the verification succeeds.

5.2 Public key infrastructure

What remains is the practical difficulty of Alice being able to reliably obtain a copy of Bob's public key. If the public key obtained is not Bob's then the verification is of no value in proving Bob's identity. To this end, there have been defined *Certificates*, which are documents including a *Common Name*, typically a web-address, since simply "Bob" is ambiguous as to which Bob, and a public key, and a digital signature upon the name-key pair, where the signature is by a well-known, trusted party, called a *Certificate Authority* (CA), whose public key is delivered to Alice during a secure and reliable conversation. In practice, the public key of the CA ships with the install medium for your operating system.

In fact, what is signed by the CA is not a certificate, but a cryptographic hash of the certificate. In addition, it is possible that the digital signature upon Bob's certificate is not from a CA, but from an entity with a certificate, and that certificate is signed by a CA. This is common. The *root of trust* is a handful of CA's, and these CA's delegate their signing power to other agents, also called CA's, by signing their certificates, with a notation on the certificate that this certificate has the authority to sign other certificates. Such certificates are called *signing certificates.*

To establish the identify of Bob, Bob provides its certificate, and any certificates in the chain of certificates to the root of trust (or Alice can search the web for these certificates). Alice verifies each signature in the chain, to establish that Bob's certificate authentically speaks for the owner of the private key associated with the public key in Bob's certificate, e.g. "www.its-me-bob.org". Alice then asks Bob to sign a sample document with the secret key, and she verifies it with the public key.

This architecture of signed certificates, leading back to the root of trust, and system of signing and verification, is called the *Public Key Infrastructure* (PKI).

6 Appendices

6.1 DES, 3DES, DESX

The Data Encryption Standards DES and 3DES are described in FIPS 46-3. First adopted in 1977, DES has withstood attack up to this day. Its 56-bit key length, however, is too short to withstand brute force attack, see the EFF's book on the subject. 3DES and DESX extend the useful life of DES by introducing more key bits.

DES is a 64-bit block cipher with 56-bit keys. The space of inputs, $[0, 2^{64} - 1]$ is mapped back onto itself in an invertible manner (a permutation) according to one of 2^{56} patterns, given the key. DES runs in two modes, applying the same key to produce the permutation or the inverse permutation, corresponding to encryption mode or decryption mode, respectively.

We need to describe precisely the security of DES. There are many ways to do this, but the following is sufficient. Suppose we are given a number of pairs (x_i, y_i) related by $y_i = \text{DES}_k(x_i)$. That is, y_i is the encryption of x_i using key k. What is the most efficient algorithm for finding k given these pairs? At present, the best approach is to select one pair (x_o, y_o) and scan all keys k looking for the value k_o such that $y_o = \text{DES}_{k_o}(x_o)$. Since only this brute force method exists, DES is considered unbroken. However there are not that many keys to try, so the brute force method is effective.

To continue the usefulness of DES, triple DES uses three keys, k_1, k_2, k_3 and encrypts the input three times, first with k_1, then with k_2, then with k_3. In addition, the second encryption runs DES in decryption mode.

$$\text{3DES}_{k_1,k_2,k_3}(x) = \text{DES}_{k_3}(\text{DES}^{-1}_{k_2}(\text{DES}_{k_1}(x)))$$

There are three modes of operation allowed: all three keys independent, all keys the same, and $k_1 = k_3$ with k_2 independent. There are two motivations for these patterns of usage. The mode $k_1 = k_2 = k_3$ reduces 3DES to DES, so hardware supporting 3DES trivially supports DES. The mode $k_1 = k_3$, with k_2 independent, is interesting since we can show that three independent keys give no more security than two. In general, cascading an encryption with n keys only gives the security of $2\lfloor (n+1)/2 \rfloor$ keys. Hence there is no point in using three independent keys over two independent keys.

Here is the proof that n keys give no more security than $\lfloor (n+1)/2 \rfloor$ keys. Divide n as evenly as possible into n_1 and n_2. If n is even, $n_1 = n_2 = \lfloor (n+1)/2 \rfloor$, else $n_1 = n_2 + 1 = \lfloor (n+1)/2 \rfloor$. Given the pair (x, y) make a table of $n_2\text{-DES}^{-1}_{k_{n_1+1},\ldots,k_n}(y)$ as the keys $k_{n_1+1}, \ldots, k_n$ take on all values.

Then try to match a value with n_1-$\mathrm{DES}_{k_1,\ldots,k_{n_1}}(x)$ as the keys $k_1,\ldots,k_{n_1}$ take on all values. A match provides n keys such that n-$\mathrm{DES}_{k_1,\ldots,k_n}(x) = y$. Try additional (x, y) pairs to verify the keys. If additional pairs don't work out, continue the search.

Although a large table of values is required (2^{64} 64-bit blocks), by attacking the encryption half-forward and half-backward we search the key spaces $k_1,\ldots,k_{n_1}$ and $k_{n_1+1},\ldots,k_n$ separately, with time at most twice the time to search the larger key space.

DESX is another approach to extending the usefulness of DES. Three keys are used, k, k_1 and k_2, where k is 56-bits, k_1 and k_2 are 64-bits.

$$\mathrm{DESX}_{k,k_1,k_2}(x) = k_1 \oplus \mathrm{DES}_k(k_2 \oplus x)$$

Question: is there an argument against $k_1 = k_2$?

6.2 Hash functions: MD4, MD5 and SHA

MD4 and MD5 are property of RSA Security and have been made publicly available for any use. A description of MD4 can be found in RFC 1186, and of MD5 in RFC 1321.

A hash function if a function which is easy to compute but hard to invert. Compared to an encryption, a hash function cannot be uniquely inverted since there are less bits in the output than in the input, therefore there are many inputs which can give the same output. Even so, it is hard to compute even a single possible input for an output. Furthermore, it is hard even to compute to inputs which give the same output, where the output is not specified before-hand. This is called *collision resistance.*

The statistics of collision resistance differs from that of inversion. Calling our hash function $\mathcal{H}$, finding an x given y such that $y = \mathcal{H}(x)$ requires on average $1/|Y|$ trials, where $|Y|$ is the size of the space of outputs. MD4 and MD5 have 512 bit input spaces and 128 bit output spaces. Hence we expect to find such an x in 2^{128} trials. However, finding different x_1 and x_2 such that $\mathcal{H}(x_1) = \mathcal{H}(x_2)$ requires only 2^{64} trials, as we argue: On the first 2^{64} calculations of $\mathcal{H}(x_1)$ we assume pessimistically that there are no collisions. Then the fraction $1/2^{64}$ of the output space is occupied. Therefore we expect a collision within the next 2^{64} values of x_1 attempted.

A hash function is secure if the statistical strength, as described above, is also the best known emperical strength, that is, no better attack is known.

Hash functions constructions allow hashing of arbitrary length bit strings into the output space. On construction is to apply the function iteratively on a padded input which is a multiple of 512 bits. The initial state is set to a known constant K and each next 512 bits of input provides the state for the next 512 bits of input,

$$\begin{aligned} h_0 &= \mathcal{H}(m_0, K) \\ h_i &= \mathcal{H}(m_{i-1}, h_{i-1}), \quad i > 0 \end{aligned}$$

MD4 and MD5 prescribe mandatory padding of the input to an even multiple of 512 bits which includes appending the original length of the input to the input as part of the padding.

MD4 has been broken by H. Dobbertin, *Cryptanalysis of MD4*, 3-ird Fast Soft. Encry., LNCS 1039, Springer-Verlag, 1966, 53–69. See also J. of Cryptology.

MD5 is now also considered broken. A hash collision was first demonstrated on 17 August 2004, by Xiaoyun Wang, Dengguo Feng, Xuejia Lai and Hongbo Yu. Since then, there have been improved attacks. Currently and an attack by Vlastimil Klima finds collisions in MD5 in only minutes on a standard PC.

SHA-1 is a modified MD5 proposed by NIST for Federal Government applications. See FIPS 180-1. It uses 5 32-bits words of internal state, compared to MD5's 4 32-bit words. That is, the output is 160-bits rather than128-bits. It still works on 512-bit inputs. Significant weakness in SHA-1 have been found.

6.3 Zero-Knowledge: Fiat-Shamir

The Bellovin-Merritt removes password guessing, but both parties, Bellovin and Merritt, need to know a common secret. This means, Merritt knows Bellovin's password and can therefore impersonate Bellovin if ever their friendship goes sour. The technology of zero-knowledge is an interaction between two parties, prover and verifier, in which the prover convinces the verifier of a fact, however, the verifier learns nothing at all from the interaction, except that the fact is true. In addition, nothing can be gained either by prover or verifier by cheating, that is, by failing to implement the protocol correctly or my making misstatements.

In the context of passwords, the prover is the client, the verifier is the server. The server is convinced by the client that the client knows the password but receives no other information from the interaction except this belief. A cheating server cannot extract more information from the client than in a fair exchange; and a cheating client cannot convince the server that it knows the password.

This all proceeds in a probabilistic setting. Either party can be replaced by a random number generator which, purely by chance, completes the protocol as would a true prover or verifier. However this is true of all security, our guarantees are all bounded by the probability of a luck guess.

The Fiat-Shamir scheme is the oldest and simplest. There are several others which address the impracticality of the scheme, due to its large computational and communication overhead. We describe this scheme to make the concepts clear, and as yet I don't know of any commercial implementations, so an impractical scheme is a good as any other.

The mathematical background is the following: given two distinct primes p and q the ring of integers mod $n = pq$ has certain elements with square roots, and others without square roots. It is not known, given an element $x \in Z_n$ where $x = y^2$ how to compute y without first factoring n into p

and q, and it is not known how to efficiently factor n. The asymmetry is intriguing: given an x of my choosing, choosing first a y and then presenting $x = y^2$, I can provide a square root of x, but no one else can. The Fiat-Shamir scheme provides to the server proof that I know a y for x such that $y^2 = x$ without revealing y.

A trusted authority chooses p and q and makes $n = pq$ public. Each client registers with the trusted authority a public v, where $v = s^2 \bmod n$, and the client holds s secret. The protocol is as follows:

1. *Committment:* The client sends the server a random $x = r^2 \bmod n$, keeping the randomly selected r private.
2. *Challenge:* The server asks for either the square root of x or the square root of vx.
3. *Decommittment:* The client sends either r or sr, according to the server's challenge.

This protocol is repeated t times until the server is sufficiently sure that the client knows s.

The client must always be able to correctly decommit for the server to be convinced and to authenticate the client. If the client knows s than it can always correctly decommit. If the client does not know s then it cannot possibly be able to decommit both x and vx, for if it could, the ratio of these decommittments is s, which we claim the client does not know. Since the client cannot predict the challenge, for t iterations of the protocol, it has a $1/2^t$ chance of deceiving the server. These are the guarantees of authenticity for the server.

The proof that the server learns nothing is ingenious. The server receives a stream of t random numbers. Since they are truly random, the server needn't have interacted with the client for these numbers. Rather, the server could have generated these numbers itself. It would generate its own r and send itself $r^2 \bmod n$ if it intends to ask for the square root of x, or send itself $r^2/v \bmod n$ if it intends to ask for the square root of vx. It is not possible to distinguish an x formed by the square of a random r from an x formed by the square of r divided by v, both a equally distributed random elements $x \in Z_n$ such that x has a square root mod n.

Since the entire conversation between client and server can be replaced with a conversation of the server with itself, the server cannot have learned anything which depends on the client. It has learned nothing it did not already know.

7 References

Bellovin, S, and M. Merritt, *Encrypted key exchange: password-based protocols secure against dictionary attacks,* IEEE Comp. Soc. Symp. on Reseach in Sec. and Privacy, May 1992. pp 72–84.

J. Bonneau, M. Just and G. Matthews, *What's in a Name?*, Financial Cryptography '10, 2010.

Burrows, M., M. Abadi, R. Needham, *A logic of authentication*, SRC Report 39, 1990.

Dobbertin, H., *Cryptanalysis of MD4*, 3-ird Fast Soft. Encry., LNCS 1039, Springer-Verlag, 1966, 53–69. See also J. of Cryptology.

Electronic Privacy Information Center, *United States v. Scarfo*, `http://www.epic.org/crypto/-scarfo.html`.

FIPS, *Data Encryption Standards (DES)*, FIPS PUB 46-3, (1999).

FIPS, *Secure Hash Standard*, FIPS PUB 180-1 (1995).

FIPS, *Password Usage*, FIPS PUB 112 (1985).

Fiat, A. and A. Shamir, *How to prove yourself: practical solutions to identification and signature problems*, CRYPTO '86 (LNCS 263), 1987, pp. 186–194.

FreeBSD Handbook, Section 10.5 S/Key and OPIE.

Haller, Neil M., *The S/KEY(TM) one-time password system,* Proceedings of the Symposium on Network and Distributed System Security, 1994.

Klima, Vlastimil, *Tunnels in hash functions: MD5 collions with a minute,* Cryptology ePrint Archive, 2006/105. `http://eprint.iacr.org/2006/105`.

Lamport, L., *Password authentication with insecure communication*, Comm. of the ACM, Vol 24, 11, Nov. 1981, pp. 770–772.

Massey, James L., *Guessing and Entropy*, Proc. IEEE Int. Symp. on Info. Th., 1994, p. 204.

Morris, R. and K. Thompson, *Unix password security*, Communications of the ACM, 22(11):594, Nov. 1979.

Morris, R. and K. Thompson, *Password Security: a case history*, Comm. of the ACM, Vol. 22, Nov. 1979, 594–597.

Myers, J., and M. Rose, *RFC 1939: Post Office Protocol - Version 3,* May 1996.

Needham, R. and M. Schroeder, *Using encryption for authentication in large networks of computers*, Comm. of the ACM, Vol. 21, Dec 1978, pp. 993–999.

Otway, D., and O. Rees, *Efficient and timely authentication*, Operating systems review, Vol 21, No. 1, Jan. 1987, p 7.

Rigney, C, et. al., *RFC 2865: Remote Authentication Dial In User Service (RADIUS)*

Rivest, R., *RFC 1186: The MD4 Message Digest Algorithm*, October 1990.

Rivest, R., *RFC 1321: The MD5 Message Digest Algorithm*, April 1992.

The Risks Digest, *Fake ATM Machine Steals PINs,* Vol 14, Issue 60, May 1993.

Smith, Richard E., *Authentication: From passwords to public keys,* Addison-Wesley, 2002.

F. Stajano, F–L Wong and B. Christianson, *Multichannel protocols to prevent relay attacks*, Financial Cryptography '10, 2010.

Wang, Xiaoyun, Dengguo Feng, Xuejia Lai and Hongbo Yu, *Collisions for Hash Functions MD4, MD5, HAVAL-128 and RIPEMD*, Cryptology ePrint Archive, 2004/199.
`http://eprint.iacr.org/2004/199`.

Zorn, G, and S. Cobb, *Microsoft PPP CHAP Extensions*, October 1998. RFC 2433.

Zorn, G, *Microsoft PPP CHAP Extensions, Version 2*, January 200. RFC 2759.