

A Survey of Authentication Mechanisms

Status of this Memo

This document is an Internet-Draft and is in full conformance with all provisions of [Section 10 of RFC2026](#). Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as ``work in progress.''

To learn the current status of any Internet-Draft, please check the ``[1id-abstracts.txt](#)'' listing contained in the Internet-Drafts Shadow Directories on [ftp.is.co.za](#) (Africa), [nic.nordu.net](#) (Europe), [munnari.oz.au](#) (Pacific Rim), [ftp.ietf.org](#) (US East Coast), or [ftp.isi.edu](#) (US West Coast).

[1.](#) Introduction

Authentication is perhaps the most basic security problem for designers of network protocols. Even the early Internet protocols such as TELNET and FTP, which provided no other security services, made provision for user authentication. Unfortunately, these early authentication systems were wholly inadequate for the Internet Threat Model [REF] and a vast array of other authentication mechanisms have been introduced in an attempt to close these holes.

The most striking thing about these security mechanisms is how many of them are essentially similar. There are only 7 basic classes of authentication protocol but there are a large number of slightly different protocols with essentially the same security properties. This memo surveys the space of authentication mechanisms, describes the basic classes and provides examples of protocols which fit into each class.

[2.](#) The Authentication Problem

The authentication problem is simple to describe but hard to solve: Two parties are communicating and one wishes to establish its identity to another. The basic scenario is exemplified by TELNET. A

client (on behalf of a user) wishes to remotely access resources on a TELNET server. The user has an account on the server and the server remembers the user's authentication information but the client itself may have no long-term storage and only limited computational capabilities. The client side of the credentials must be able to be carried by the user, either on a small device or in his memory.

[2.1.](#) Authorization vs. Authentication

AUTHORIZATION is the process by which one determines whether an authenticated party has permission to access a particular resource or service. Although tightly bound, it is important to realize that authentication and authorization are two separate mechanisms. Perhaps because of this tight coupling, authentication is sometimes mistakenly thought to imply authorization. Authentication simply identifies a party, authorization defines whether they can perform a certain action.

Authorization necessarily relies on authentication, but authentication alone does not imply authorization. Rather, before granting permission to perform an action, the authorization mechanism must be consulted to determine whether that action is permitted. This document is solely concerned with authentication.

[2.2.](#) Standard Material Something you have, something you know...

[TODO]

[3.](#) Description of Authentication Mechanisms

The next seven sections each describe a single class of authentication technology. In each case, we first describe the technology in general, with possible subsections describing security or implementation issues that are generic to this technology. Once we have described the technology in general we then provide one or more case studies: descriptions of specific protocols which use this authentication technology and the various security or implementation issues that are specific to that protocol. Thus, each section follows the following pattern.

- A Mechanism
(Description)
- A.x risk
(description and countermeasures)
- A.y risk
(description and countermeasures)
- A.z Case Study: Specific Protocol
(description of the protocol)
- A.z.x Protocol Specific problems.
- A.w List of Protocols/Systems tha use this mechanism

[4. Passwords In The Clear](#)

The most commonly used form of authentication is for the client to provide a username/password pair to the server in the clear (e.g. over a TCP channel). The server then verifies the password against the user's stored credentials. If they match, the server allows the client to access the resource.

The most primitive approach is for the server to simply store the user's username and password in a file on the server's disk. This has the serious problem that if the password file is somehow compromised, the attacker has immediate access to all user passwords and can log in as any user. The standard approach, first used by XXX, is to store a message digest of the password instead of the password itself. When the server needs to verify a password, it digests the password and compares the digest against the stored digest. Since a message digest is used, the server cannot recover the user's password from the password file.

[4.1. Password Sniffing](#)

The simplest attack against passwords in the clear is simple password sniffing. The attacker arranges to intercept traffic between the client and the server (this is relatively easy, especially if the attacker is on the same network as one of the endpoints). Since the password traverses the network in the clear, the attacker is easily able to recover the password and can use it for any future authentications.

[4.2. Post-Authentication Hijacking](#)

An attacker who can hijack network connections need not know the user's password at all. He can simply wait for the user to complete his authentication and then take over the connection. This attack is more difficult to mount than password sniffing, but as we'll see later, it can be useful when stronger authentication schemes are

employed.

[4.3.](#) Online Password Guessing

Extensive experience [REF] shows that users choose bad passwords. Common choices include the user's real name, login name, date of birth, and simple dictionary words. An attacker with no special capabilities can therefore attack a server by simply trying known or common usernames and common passwords. This technique was used to great effect by the Morris worm [REF].

The standard countermeasure to this attack is to make it difficult for the attacker to try a large number of passwords. This can be done by incorporating a LIMITED TRY capability. After some number of failed attempts, the system simply locks the account and the user cannot log in even with the correct password. Unfortunately, simple limited try provides the attacker with an easy denial-of-service (DoS) attack--he can lock any account simply by performing failed logins.

A superior approach is to incorporate a delay. For instance, the system might allow the user to immediately try 3 passwords, but after three failures lock the account for 10 seconds, increasing the delay (up to some fixed maximum) for each failure. This is a less effective countermeasure than simple LIMITED TRY but resists the DoS attack better.

[4.4.](#) Offline Dictionary Attack

Even if digested password files are used, it is still often possible for an attacker who recovers the password file to discover user's passwords. The attacker can mount an OFFLINE DICTIONARY ATTACK on the password file. A dictionary attack uses the fact that users tend to choose words rather than random strings in order to narrow the scope of exhaustive search. The attacker simply runs through each word (and common variations) in sequence, comparing the digest of the trial word against the digest in the password file. There are a number of programs available to mount this sort of attack, including [REF].

4.4.1. Shadow Passwords

There are four basic countermeasures to offline dictionary attack. The first is to deny attackers the password digest. In the original UNIX systems, reading the password file was the only way to get information about users and therefore the password file had to be publicly readable. Later systems introduced SHADOW PASSWORDS, whereby the password file contained a dummy password and a second copy of the password file containing the encrypted passwords was unreadable

Rescorla

[Page 4]Intern

except to root. Thus, unprivileged user processes would consult the ordinary password file (now containing dummy passwords) to get user information (such as name, home directory, etc) but only privileged processes can read the encrypted passwords. Of course, sometimes an attacker can convince a privileged process (via bugs) to give him a copy of the file, thus allowing him to attack it.

4.4.2. Iteration

The second type of countermeasure is to make search slower. One approach is to simply make the hash function slower. The original UNIX crypt() function did this by repeating the basic operation (based on DES) 25 times. (The designers also slightly modified the operation so that it couldn't be done with ordinary DES hardware.) The idea here is that noone will notice a second or so delay on login but that making each guess take a second will seriously slow down an attacker. To compensate for the speed of modern computers, rather more iterations are currently required each year.

4.4.3. Salting

If a simple hash of the password is stored in the password file, then an attacker can attack all the passwords in the file in parallel. He simply generates the hash of each candidate and then compares it against each stored hash. In order to prevent this attack, many systems SALT the hash with some random value (which is different for each user). Thus, instead of storing simply $H(\text{password})$ they store $\text{salt} || H(\text{salt} || \text{password})$, with the result that even two users who have the same password will in general not have the same stored password hash. One interesting innovation is to use a secret salt. This requires the attacker to try all possible salts, automatically slowing down the process (thereby making iteration unnecessary).

4.4.4. Stronger Passwords

The reason that dictionary attacks are so easy is that users choose

bad passwords. Even the 8 character UNIX password space allows 2^{56} possible passwords--a search space that is impractical for most attackers to search. One obvious countermeasure is to force users to choose good passwords. This can be done reactively by running a password cracker on your system or proactively by forcing users to use good passwords when they set them. It's also possible to force users to use randomly generated passwords. Unfortunately, unguessable passwords are often less memorable, causing users to write them down. It's not clear that this is an improvement. Security-conscious people are often willing to use complex mnemonics to help remember random passwords but ordinary users are not. One welcome innovation on this front is the replacement of the old UNIX DES-based `crypt()` function

with an MD5-based function that accepts longer passwords, allowing the user to have a meaningful but still harder to guess password.

[4.5.](#) Case Study: HTTP Basic Authentication

HTTP basic authentication [REF] is the original HTTP authentication mechanism. It's a simple username/password scheme. The server prompts the client with a request for authentication (in a `WWW-Authenticate` header). The client responds with the password in an `Authorization` header. The password is base-64 encoded but this doesn't provide any security, just protection from damage in transport.

[4.5.1.](#) Password Caching

Any reasonable Web page fetch consists of a number of HTTP fetches, each of which may require HTTP authentication. Requiring the user to type in his password for each such fetch would be prohibitively intrusive. Accordingly, web clients typically cache the user's password for some time (generally for the lifetime of the browser process.)

In some cases, the browser will cache password on disk so that the user never has to type in the password again. This practice introduces a new security problem: protection of the user's cached passwords. These passwords can be encrypted on disk (under another password) but users often find this inconvenient and so the passwords are often stored on the disk in the clear. This is dangerous on multi-user machines, even ones which provide strong file permissions, since administrators can still read such cache files.

[4.5.2.](#) Pro-active authentication

Requesting a page, receiving an authentication challenge and re-requesting with a password introduces an extra round-trip. This latency can be quite significant if the original request was large, such as with a file PUT. Thus, many clients pro-actively send their cached passwords whenever accessing any URL deeper than the URL for which they were originally prompted.

[4.6.](#) List of Systems that Use Passwords in the Clear

TELNET
HTTP (basic authentication)
SASL (password mode)
RLOGIN
POP (among others)

Rescorla

[Page 6]Intern

IMAP (among others)
(too many others to mention)

[5.](#) One Time Passwords

The simplest approach to preventing sniffing attacks on passwords is to use ONE TIME PASSWORDS. In it's basic form, the user is provided with a list of passwords, each of which can only be used once, making replay attack impossible. The passwords are still transmitted in the clear, but since each one can only be used once, a sniffed password cannot be used as an authenticator.

The major use of one-time password systems is to improve the security of protocols which previously used password authentication. One-time password schemes can be designed such that they require no changes to the client software and only minimal changes to the server software. The user generally needs to have either a physical password list or a token that computes the password, but the client software does not need to be replaced and the wire protocol is unchanged.

None of the one-time password schemes are very useful for automated authentication, since they only provide a limited number of keys. Using automated authentication with S/Key or OTP it is easy to quickly use up a large number of keys. SecureID provides an essentially infinite number of keys but they are changed too infrequently to be usable in most automated systems.

As with ordinary passwords, one time password mechanisms are subject to a number of active attacks. However, even if the attacker captures

a specific authenticator via an active attack, he can use it only once, not indefinitely. [TODO: Are SecureID keys marked unusable so they can't be used again during the 30s window? I imagine so...]

[5.1](#). Case Study: S/Key and OTP

S/Key [REF], invented by Neil Haller and Karn, is a straightforward one time password system that uses some clever implementation tricks. One-Time Passwords (OTP) [REF] is the successor protocol to S/Key, standardized by the IETF. In S/Key, the one time passwords are constructed by iteratively hashing a public seed and a secret. Thus:

$$P[0] = H(\text{Seed}, \text{Secret})$$
$$P[i] = H(P[i-1]).$$

Passwords are used in reverse order. This allows the server to simply store the last password that it received ($P[i]$). The client will next authenticate with $P[i-1]$. The server can verify a password by hashing

it and checking to see if it matches the stored password. Once authentication is complete, the server simply deletes the old password and stores the new one.

S/Key uses a special password encoding that's designed to make it easy for users to type passwords without errors. The 64-bit one-time password is broken up into a sequence of six 11-bit values (with the remaining two bits being used as a checksum). Each 11-bit value is used as an index into a fixed dictionary of 2048 short words. Thus, a password might look like:

INCH SEA ANNE LONG AHM TOUR

This encoding is intended to be easier to type than base64 or hexadecimal. (Though hexadecimal is accepted as well).

S/Key can be used in two modes. In the first, the client is simply provided with a list of passwords on a piece of paper. He uses one at a time and crosses them off as he goes. In this case, the Secret is usually cryptographically random. In the second, the client has a token or a computer program that he uses to calculate the appropriate S/Key key. In this case, the Secret is generally some user-memorable password which the user keys into the program or token.

S/Key scheme has a number of nice properties. First, the password file need not be kept secret, since going from $P[i]$ to $P[i-1]$ requires reversing the message digest, which is believed to be computationally infeasible. (Note: if a text password is used as the secret then the password file is still subject to dictionary attack, but a passive attacker who recovers ANY S/Key authenticator can mount a dictionary attack on it (by iteratively hashing the potential seed), so it's not that important to keep the password file per se protected).

Second, it's easy for the user to rekey: He simply creates a new Secret, generates a set of keys and sends the last one to the server. Note that it's of course possible for an active attacker to hijack a connection and rekey with a key of his choice, thus one time passwords are in general a poor choice when active attack is part of the threat model.

[5.1.1](#). Race Conditions

S/Key has an interesting security flaw: Consider a protocol where passwords are transmitted one character at a time. A passive attacker might wait for the victim to log in and then create his own login connection at the same time. The attacker would then echo the

victim's password character for character, until there was only one character left. At this point the attacker would simply guess the last character and then complete the authentication. This attack is relatively simple to mount because nearly all the words in the S/Key dictionary are 4-characters long and the number of words with any given 3-letter prefix is generally quite small (2 or 3).

The standard countermeasure to this attack is to only allow one pending authentication for a given user at any given time. In order to prevent DoS attacks, there must be a timeout on any such pending connection. OTP implementations are required to implement this or some other countermeasure.

[5.2](#). Case Study: SecureID

Probably the most commonly deployed commercial one time password implementation is SecureID, sold by Security Dynamics (now part of RSA Security). Instead of using a fixed list of keys, SecureID uses a time-dependent key. The user has a token with an LCD displaying a pseudo-random number. That number changes every 30 seconds and is synchronized with an authentication server located at the server.

In order to authenticate the user enters both his password and the time-dependent key (they can be concatenated so that this is transparent to the client program.) The server verifies the password and checks that the time-dependent key is correct for the current time and only then allows login. It's clearly possible for an attacker to capture the password and replay it but without the token he (theoretically) can't generate the right time-dependent key.

[TODO: Talk about the algorithm being exposed and potentially compromised].

[5.3.](#) List of One-Time Password Systems

Note: any system that uses passwords can be adapted to use one-time passwords.

S/Key [REF]

OTP [REF]

SecureID [REF]

[6.](#) Challenge/Response

CHALLENGE/RESPONSE mechanisms fix the sniffing problem associated with ordinary passwords. The basic idea is simple: the verifying party provides a random (or at least unique) challenge and the authenticating party returns some function of the shared key and the

challenge. Generally this function is some sort of message digest. In the simplest form it is $H(\text{challenge} || \text{key})$. A better design is probably to use HMAC [REF], which has stronger security guarantees.

Challenge/response mechanisms are resistant to simple sniffing attacks but in general have all the other security problems of ordinary password systems. Additionally, they are vulnerable to another form of offline dictionary attack and are more vulnerable to password file compromise than correctly implemented password in the clear systems.

Challenge/response mechanisms can be completely hardened against offline dictionary attacks by the use of a sufficiently large randomly-generated shared key instead of a password. Such a password is of course difficult for a user to memorize but is quite useful if it can be statically configured on both sides of a connection.

Unlike simple password mechanisms, challenge/response mechanisms can

be designed which provide both mutual authentication and secure key exchange. Such systems can be made resistant to most forms of active attack, and depending on the strength of the shared key, passive attacks as well.

6.1. Offline Attacks on Challenge/Response

Although a passive attacker cannot mount an ordinary sniffing attack, he can combine sniffing with an offline dictionary attack. The attacker simply captures a single challenge/response exchange and then dictionary searches the password space until he finds a password that produces the correct response for a given challenge. With high probability (though not certainty) this will be the correct password. This problem is inherent in all simple challenge response mechanisms and cannot be fixed without public-key technology.

6.2. Password File Compromise

Challenge/response mechanisms also introduce a new problem: PASSWORD EQUIVALENCE. In order to locally compute (for verification purposes) the appropriate response for a given challenge, the server must store the user's password locally. Thus, if the password file is compromised, the attacker can directly log in to the server, without even needing to crack the password file. We'll call this property WEAK PASSWORD EQUIVALENCE.

A more serious variant of the same problem occurs if users use the same password on multiple systems. Compromise of one system can thus lead to compromise of many. This is called STRONG PASSWORD EQUIVALENCE. This risk should not be overstated--compromise of an ordinary

password system can still lead to attack if the attacker completely compromises the system and can capture people's passwords when they login--but is nevertheless worse in challenge/response than with ordinary passwords. The standard countermeasure is to use a two-stage digesting process, such as:

$$\begin{aligned}\text{STORED} &= H(\text{PASSWORD} \parallel \text{SALT}) \\ \text{RESPONSE} &= H(\text{STORED} \parallel \text{CHALLENGE})\end{aligned}$$

The server stores STORED instead of the password. (Making STORED effectively the password). The server then gives the client both SALT and CHALLENGE, allowing the client to compute RESPONSE from the password alone. Note that the two-stage process only prevents compromise of one system from affecting others. Compromise of a password file

still allows immediate access to the target system.

[6.3](#). Case Study: CRAM-MD5

CRAM-MD5 [REF] is a challenge/response authentication extension for IMAP [REF]. CRAM-MD5 is a classic challenge/response system: the server provides a presumably random challenge and the client transmits an HMAC of the challenge using the shared key as the HMAC key. The interaction looks like this:

```
1 S: * OK IMAP4 Server
2 C: A0001 AUTHENTICATE CRAM-MD5
3 S: + PDE40TYuNjk3MTcwOTUyQHBvc3RvZmZpY2UucmVzdG9uLm1jaS5uZXQ+
4 C: dGltIGI5MTNhNjAyYzdlZGE3YTQ5NWl0ZTZlNzMzNGQzODkw
5 S: A0001 OK CRAM authentication successful
```

The second message from the server (message 3) is the base-64 encoding of the string "<1896.697170952@postoffice.reston.mci.net>". This string must be in the form of an [RFC 822](#) msg-id [[RFC822](#)] and is intended to be globally unique. The client's response (message 4) is computed using HMAC-MD5(password,challenge) and then base-64 encoded for transmission in message 4.

CRAM-MD5 is an improvement on the password-in-the-clear mechanisms that it replaces but still has all the security flaws of basic challenge/response mechanisms. In particular, it is vulnerable to post-authentication hijacking and is strongly password equivalent.

CRAM-MD5 has some interesting security properties with respect to server password file compromise. The RFC encourages servers to store a pre-initialized HMAC context rather than the client's password. Since the password has already gone through the MD5 compression

function, it is believed to be infeasible to recover the password from the context. However, since the HMAC context is sufficient to compute any response without knowing the key, an attacker who recovers the context can impersonate the client without knowing the key. This context will be the same for all servers which share the same password. The result of these facts is that an attacker who recovers the password file from such a server can attack any other server which (1) uses CRAM-MD5 and (2) has a user with the same password. However, it cannot attack other users with the same password on machines with a different authentication mechanism (since that would require direct access to the password rather than the HMAC context).

[6.4.](#) Case Study: HTTP Digest

HTTP Digest Authentication [REF] is a replacement for HTTP's [REF] notoriously weak Basic Authentication mechanism, which used passwords in the clear. Digest Authentication is a challenge/response mechanism with some additional features to prevent hijacking attacks and remove strong password equivalence, as well as to reduce round trip time for multiple requests.

The basic Digest Authentication interaction takes two round trips. In the first, the client requests some document and is rejected. The server's rejection (a 401 Unauthorized) contains an indication that it supports Digest Authentication, a realm string, and a random challenge. The client's subsequent request includes a message digest over the password, the challenge, and part of the HTTP Request.

HTTP Digest offers two types of integrity check (called "qop" for quality of protection). The "auth" scheme covers only the request URI. The "auth-int" scheme protects the URI and the message body, but not the message headers since they may be changed in transit by proxies or other intermediaries. Negotiation of the qop is simple: the server offers a set of acceptable qop values and the client chooses one.

[6.4.1.](#) Message Integrity

As previously noted, simple challenge/response schemes without associated channel security allow an attacker to hijack the connection after authentication has occurred. Since each HTTP request must be individually authenticated, an attacker who takes over the channel cannot transmit new unauthenticated requests over that channel. However, an attacker might attempt to intercept an authenticated request and mount a cut-and-paste attack, leaving the authenticator but changing the contents. This attack is prevented by including the URI in the message digest.

Unfortunately, the URI isn't the only piece of security relevant information in the HTTP request. Both the headers and the body are potentially sensitive. For instance, if HTTP POST is used, FORM input values will be in the message body. The auth-int qop value protects this information, but it is not widely deployed. None of the qop values protects the headers.

It's worth noting that Digest provides protection only for the

request. No authentication is provided for the server, nor is message integrity provided for the response. It's technically possible to provide this feature using a shared key, as is done in S-HTTP [REF], but Digest doesn't do so.

Digest deployment has been somewhat spotty. The popular Netscape Navigator 4 versions do not support it, nor did Internet Explorer prior to version XXX. The fact that there have recently been some reports of incompatibilities between various implementations suggests that only minimal testing has so far occurred.

[6.4.2. Replay Attack](#)

Many HTTP requests are idempotent. In such cases, replay attacks are not a problem since the attacker doesn't get any information that he would not get by sniffing the original request. However, many HTTP transactions have side effects and in such cases preventing replay is important. Unfortunately, the conventional approach of requiring a separate challenge/response exchange for each authentication would double the number of round-trips for each transaction.

HTTP Digest provides two features to avoid these round trips. First, the server can provide a new nonce in a response header. This nonce must be used for the next client request. This feature interacts poorly with request pipelining so HTTP Digest also allows the client to issue multiple requests using a given server challenge by using a request sequence number (the "nonce-count"). [TODO: How widely is this stuff implemented]

[6.4.3. Downgrade Attack](#)

HTTP Digest suffers from two types of downgrade attack. In the first type of attack, the attacker forces the peers to agree on Basic authentication rather than on Digest. There is no realistic way to protect against this attack, other than simply refusing to accept Basic at all.

In the second Downgrade attack, the attacker forces the peers to negotiate a qop of "auth" instead of "auth-int". The downgrade attack would then presumably be followed by an integrity attack on the

client request. This attack could be prevented by requiring the client to include a digest of the server's offered qop values in the client's authenticator. However, that is not the case with the current scheme.

[6.5.](#) List of Challenge-Response Systems

APOP [REF]
ACAP [REF]
HTTP Digest [REF]
AKA [REF]
CRAM-MD5 [REF]
Kerberos [REF]

[7.](#) Anonymous Key Exchange

All three of the mechanisms mentioned so far can be hardened against passive attacks by the use of anonymous key exchange. Essentially, the client and server arrange for a secure channel using some anonymous public key algorithm (such as anonymous DH or RSA without certificates) and then carry out the same communication over that channel that they would previously have done in the clear. This prevents a passive attacker from sniffing the password (with passwords in the clear) or the authenticator (with one time passwords or challenge/response). (It's technically possible to just protect the password, but not generally done).

How much security you believe that anonymous key exchange adds to your protocol depends on your threat model. Since the initial key exchange is completely anonymous, an active attacker can mount a man-in-the-middle (MITM) attack and obtain a password or authenticator. Active attacks are generally more difficult to mount than passive attacks but by no means impossible. [REF]

All of these mechanisms use public key cryptography to perform the initial anonymous key exchange. As a result, performance can be unacceptably slow if the clients are heavily constrained. Most servers are fast enough to keep up with the normal number of required authentications and hardware acceleration solutions are readily available.

[7.1.](#) Case Study: SSH Password Authentication

Secure Shell (SSH) provides a number of authentication mechanisms, but the first step is always to establish a secure channel between the client and the server. SSH is designed not to require certificates: the server merely provides a raw public key to the client. As a countermeasure to man-in-the-middle attack, the SSH client caches the server's public key and generates a warning or error (depending

on the implementation) if that key changes.

In theory, caching the public key protects against MITM attack at any time other than the initial connection to the server. In practice, when users encounter the error that the key has changed, they often simply override the warning or delete the cache entry when the error occurs, assuming, correctly, that the likely case is that the server administrator has just reset the public key (e.g. by reinstalling the software without preserving the old key).

A very careful user can obtain complete security against MITM attacks by obtaining the server's key fingerprint (a message digest of the key) out of band and comparing that to the fingerprint of the key the server offers.

SSH bootstraps off of the system's login mechanisms so it will support either passwords in the clear or one time password authentication. Note that in either case if an attacker mounts a successful man in the middle attack, he will be able to hijack the connection post-authentication, just as he would have if the transaction was performed in the clear. This vulnerability can be alleviated with careful protocol design, as we'll see in the next case study.

[7.2.](#) Case Study: TLS Anonymous DH + Passwords

[TODO]

[7.3.](#) List of Anonymous Key Exchange Mechanisms

SSH (password mode) [REF]
SSL/TLS (anonymous keying) [REF]
???

[8.](#) Zero-Knowledge Password Proofs

All of the mechanisms mentioned so far depend on some sort of shared key. If that shared key is a user-derived password, then it's possible for the attacker to mount an offline dictionary attack on the password, either completely passively (as with CRAM-MD5) or with a single MITM attack (as with TLS anonymous DH). However, a rather clever class of protocols known as Zero Knowledge Password Proofs (ZKPPs) makes it possible to use user-generated passwords without fear of offline dictionary attack

The earliest (and simplest) ZKPP is EKE [REF], designed by Steve Bellovin and James Merritt. The basic idea behind EKE is simple. A DH exchange is performed but the exponentials are encrypted under the password. Since the exponentials are essentially random, it's not

possible to dictionary attack the password. The protocol looks like this:

Client	Server
-----	-----
Name, E(Password, Ya)) ->	
	<- E(Password, Yb), E(K, Challenge-b)
E(K, Challenge-a Challenge-b) ->	
	<- E(K, Challenge-a)

Where K is the DH shared secret == $g(Xa * Xb) \bmod p$

Note that EKE as described above is insecure against password file compromise, since the server must store the password. Augmented EKE [REF] describes a protocol that is secure against this. A large number of other ZKPPs have been proposed, including PDM [REF], SPEKE [REF], and SRP [REF]. These protocols are all roughly equivalent, offering slightly different combinations of security, performance, and message count.

[8.1. Intellectual Property](#)

From a technical perspective, ZKPPs dominate the anonymous key exchange mechanisms described in Section XXX. Their performance is roughly equivalent and their security guarantees are superior. Nevertheless, none of the ZKPPs have achieved widespread deployment. The reason for this appears to be intellectual property. The original EKE work was patented (as of this writing the patent is held by Lucent). All of the variations are either themselves patented or claimed to be covered by some patent or other. There is no ZKPP that is widely agreed upon to be unencumbered or available under a royalty free license.

[8.2. List of Zero Knowledge Password Proof Systems](#)

EKE [REF]
A-EKE [REF]
SPEKE [REF]
SRP [REF]

[9. Server Certificates plus Client Authentication](#)

If you can authenticate one side of the connection (typically the server) then it becomes far easier to provide strong authentication. Anonymous key exchange, cleartext passwords, one time passwords, and challenge/response protocols can all run over an authenticated and encrypted channel. In such a system, there's no need to worry about

active attack, so the authentication protocols don't need to be hardened against it.

Providing an encrypted channel with authentication for the server dramatically reduces the security advantage enjoyed by more complicated schemes over simple passwords. Since the marginal security benefit of such systems is so modest when compared to the increased implementation and deployment complexity, common practice when server authentication is available is to use simple passwords over the encrypted channel.

In addition to making the overall authentication problem simpler, hosting one's application protocol over an encrypted and authenticated channel has a number of other security benefits. First, a properly designed channel security protocol removes the threat of post-authentication hijacking (described in Section XXX). Second, it provides confidentiality and message integrity for the rest of the application traffic, which is in general a good thing.

The primary difficulty with this approach is that providing certificate-based server authentication is not straightforward. The first problem is that the server machine must have a certificate, which entails some inconvenience and cost. Self-signed certificates aren't acceptable in this case (rather, they reduce you to the anonymous key exchange scenario described in Section XXX).

The more serious problem is establishing what the server side identity in the certificate ought to be. Common practice (stemming from practice in HTTPS [REF]) is to have the server's certificate contain the server's fully qualified domain name (FQDN), either in the Common Name or subjectAltName fields, but this is unacceptable if the server does not have a domain name. One can also put the server's IP address in the subjectAltName, but this is inappropriate if that IP address might change. Any protocol which uses this mechanism MUST specify a mechanism for determining the server's expected domain name.

[9.1](#). Case Study: Passwords over HTTPS

Despite the existence of Digest Authentication, the dominant form of strong HTTP authentication is passwords with HTTP over SSL (HTTPS). As mentioned above, this mechanism has superior security properties to Digest (provided that the server has a real certificate) and is easier to deploy, especially if the server wants to use SSL/TLS for channel security in any case.

There are actually two ways to use passwords over HTTPS. The first is

to use HTTP's built in authentication mechanisms (either Digest or Basic) over an HTTPS connection. The second is to perform password

authentication at the application layer, using an HTML form to prompt for the password. The form method is far more popular, primarily because it allows the application designer far greater control over when and how authentication occurs. In particular, the designer can give the password dialog any look he chooses.

In general, if form-based authentication is used, the only available option is to use simple passwords, since HTML has no facilities for performing arbitrary computation or challenge/response passwords. Theoretically, one could perform these operations in a JavaScript or Java program, but in practice this is generally not done.

[9.1.1.1. Authentication State](#)

When Basic or Digest Authentication is used, the client can simply transmit an authenticator with every request. However, if authentication is performed using an HTML form, this approach is impractical, since it would require client interaction for every page fetch. Three approaches for solving this problem are generally proposed.

[9.1.1.1.1. The Token Problem](#)

In general, all HTTP authentication state carrying schemes involve providing the client with some token which it can then present to authenticate future requests. This token must be constructed in such a fashion that it is impossible for the client to tamper with it and obtain access to resources that they would not otherwise be able to access.

There are two basic techniques for constructing tokens. The first is to have the token be self-authenticating, e.g. by having it be the user's information signed or MAC-ed with a key known only to the server. The second is to have it be an index into some database of authenticated users stored on the server. Note that these indices must be unpredictable to prevent one user from guessing another user's token. The self-authenticating approach has the advantage that it does not require persistent storage on the server but the disadvantage that there is no way to mark a token invalid or update it (although they can of course contain an expiry time). When multiple servers are involved, self-authenticating tokens have the additional advantage that they do not require inter-server communication.

[9.1.1.2](#). URL Rewriting

The most general but also most difficult approach is to dynamically rewrite all URLs provided to the client after authentication has occurred. One might, for instance, pass all pages through a CGI

Rescorla

[Page 18]Intern

script, where the arguments include the real page to be accessed and the authenticator token. an example of such a URL is:

```
http://www.example.com/cgi-bin/gw.pl?authenticator  
=MjFkNWQyOGRjYjlmM2IwMmJjMzk0NGFhODg0YTQ4YTcK?page=foo.html
```

The CGI script would then use the authenticator argument to determine the client identity, recover the actual target page and perform the authentication checks. Using a CGI script this way is inconvenient since it requires replicating the server's access control infrastructure. A less intrusive approach involves having a server plugin unwrap the target URL early in the server's processing pipeline, before the access control checks are performed. This allows the server to perform it's normal authentication checks based on the unwrapped identity.

The primary difficulty with URL rewriting is that it all pages must be dynamically generated. Either each page must be generated by a script which embeds the appropriate URLs or the server must postprocess pages to embed them. Either approach makes the system more complex and therefore adds instability. However, before the introduction of cookies, URL rewriting was essentially the only option for token passing.

[9.1.1.3](#). Cookies

The inconvenience of URL rewriting lead to the introduction of HTTP Cookies [[RFC 2109](#)]. Essentially, an HTTP cookie is a token issued by the server and transmitted by the client with requests. The cookies can be labeled to be transmitted only when resources matching various prefixes are dereferenced, including resources on another server. Browsers generally persistently cache cookies between invocations.

Cookies are the method of choice for carrying HTTP state information and can be used to carry all kinds of state besides authentication information. Note, however, that since cookies can be used to transmit information from one server to another, they have been the focus of privacy concerns [REF]. Accordingly, some users choose not to

accept or transmit cookies.

[9.1.1.4](#). HTTPS Session Binding

Each TLS/SSL session has a session identifier, which is used for resuming the session without a full handshake. These session IDs are unique for any given server, so server administrators often think to use the session ID as a search key for the user's information. This is a bad idea. The fundamental problem is that there's no guarantee

that any given session will be resumed. The client need not offer to resume a session and the server need not accept, or may flush its session cache at any time. Thus, using the session ID as a persistent identifier is unwise.

[9.2](#). List of Server Certificate Systems

Lots of protocols over SSL/TLS [REF]
IPsec (under some conditions) [REF]

[10](#). Mutual Public Key Authentication

If both client and server have certificates, then the peers can use mutual certificate authentication. This is done by having both client and server establish that they know the private keys corresponding to their certificates. A wide variety of protocols offer this functionality, including SSL, IPsec, and SSH (SSH actually offers mutual authentication with pre-arranged public keys).

The two most important advantages of public key authentication are that it has no password equivalence and that it can allow authentication between parties who have no prior arrangement.

[10.1](#). Password Equivalence

With public key authentication, the server knows only the client's public key. It is therefore incapable of forging any kind of authentication message from the client. Similarly, knowledge of the public key does not allow an attacker to authenticate to the server. Accordingly, public key techniques never store a password equivalent on the server.

[10.2](#). Authentication between Unknown Parties

One advantage of certificate-based public key authentication systems

--as opposed to those using pre-arranged public keys--is that it allows authentication between parties who have had no prior contact. Authentication of servers with which one has had no prior arrangement happens all the time in the HTTPS context: the user wishes to connect to a host at a given URL and is able to verify that the server certificate matches that URL.

In addition to strict identity verification, it's possible to use certificates to carry authorization information. This allows a central authority to make both authentication and access control decisions for distributed servers merely by issuing certificates. [REF: Policymaker?] describes such a system.

Rescorla

[Page 20]Intern

[10.3.](#) Key Storage

The primary security problem with public key authentication protocol (assuming the basic protocol is designed correctly) is protecting the client's public key. Generally, the server's public key can be protected by hardening the server, but the user often needs to be able to carry his private key around. This can be done in essentially two ways: with a token or by generating the key from a password.

[10.4.](#) Tokens

The general idea of a secure token is relatively simple: you have a tamper-resistant and portable token which carries your private key (and probably your certificate). The token can be interfaced to a computer, typically through a USB jack or a smartcard interface. The private key is generally protected by a PIN, but of course this PIN is known to any computer on which the token is used, since the PIN is sent to the token by the computer. The primary threat to tokens is loss or theft. It's not generally economical to make such tokens completely tamper-proof, so a lost token in the hands of a dedicated attacker means a lost private key.

[10.5.](#) Password Derived Keys

It's generally possible to derive a user's private key from a relatively short password, simply by using the password to seed a cryptographically secure PRNG which is used to generate the private key. Unfortunately, this technique is susceptible to dictionary attack, since an attacker can dictionary search the password space until he finds a password that generates a key pair that matches the signature. Protocols can be designed to resist this attack by exchanging the signed client response under the server's private key, but many

protocols (notably SSL) do not. Accordingly, password derived keys should be viewed as a mechanism for using shared keys with public-key-only protocols, not as a fully public key system.

[10.6](#). Case Study: SMTP over TLS

[TODO]

[10.7](#). List of Mutual Public Key Systems

SSL/TLS (client auth mode) [REF]
IPsec [REF]
S/MIME [REF]

11. Generic Authentication Mechanisms

An approach that has lately gained currency is to use a generic authentication negotiation system. Examples of such systems include SASL [REF] and EAP [REF]. The general idea is that one has a protocol framework which doesn't provide any authentication features per se but instead allows you to negotiate the authentication mechanisms you wish to use. SASL, for instance, allows the negotiation of CRAM-MD5 (a digest-based challenge/response mechanism), SRP, and TLS among other mechanisms.

Generic authentication mechanisms are attractive to application protocol designers because they allow them--in theory--to add security to their protocols without having to directly deal with the security issues. They simply specify that one should use a given framework. They're attractive to security mechanism designers because it's relatively easy to add new mechanisms.

[11.1](#). Downgrade Attacks

The most serious problem with generic authentication mechanisms is their susceptibility to DOWNGRADE ATTACK, in which the attacker interferes with the negotiation to force the parties to negotiate a weaker mechanism than they otherwise would. Consider a set of peers, each of which supports both challenge/response and simple passwords. An attacker can force them into using a simple password and then capture that password.

The standard countermeasure to downgrade attack is to authenticate a message digest of the offered mechanisms. Unfortunately, this protection is only as strong as the weakest common mechanism. If this mechanism is a simple password then no protection against downgrade attack is possible. The possibility of downgrade attack requires that users of generic security mechanisms carefully profile the mechanisms they offer to ensure that they are all adequately strong.

[11.2.](#) Integration with Applications

[TODO]

[11.3.](#) Multiple Equivalent Mechanisms

The ease of adding new security mechanisms to generic authentication layers means that a given authentication layer may have a number of different mechanisms with essentially similar characteristics. For instance, SASL has mechanisms for SecureID [[RFC 2808](#)], OTP [[RFC 2245](#)], and Digest Authentication [[RFC 2831](#)]. In addition, there is currently an Internet-Draft for CRAM-MD5 support in SASL. With the

Rescorla

[Page 22]Intern

exception of Digest, all of these mechanisms offer essentially the same security properties (Digest also allows the negotiation of a shared key for session encryption).

So, why the proliferation of superficially redundant mechanisms? From a security perspective, they could all be replaced by Digest. The reason appears to be legacy authentication mechanisms. Many environments already have S/Key or SecureID installed and the administrators don't want to replace them. This inevitably creates pressure to add every conceivable security mechanism to one's generic framework.

While the desire to support legacy authentication systems is understandable, it should be resisted to the extent possible. Having multiple equivalent mechanisms dramatically increases both implementation complexity and interoperability problems. When designing a new system, designers should choose a very small number of authentication mechanisms, with no more than one of any given class.

[11.4.](#) Excessive Layering

Many of the legacy authentication mechanisms that users and administrators wish to support are themselves generic frameworks of one kind or another. For instance, SASL allows the use of GSSAPI, which itself is a generic framework for a number of mechanisms. This sort of layering dramatically increases both implementation and deployment com-

plexity. For instance, GSSAPI contains mechanisms that are essentially equivalent to Kerberos, but SASL also supports Kerberos directly. Under what conditions should clients use Kerberos directly and under which should they use it through GSSAPI?

Another example of the same problem is the Extensible Authentication Protocol (EAP) [REF], an authentication framework originally designed for PPP [REF]. Note that PPP itself allows multiple authentication mechanisms, so PPP must first negotiate EAP. EAP then negotiates the individual mechanisms. To make matters worse, one of the EAP mechanisms is TLS [REF] which can negotiate it's own authentication mechanisms. Three levels of indirection seems a bit much.

In accordance with the principle of having as few mechanisms as possible, frameworks should avoid mechanisms that are themselves frameworks, in favor of using the second framework's mechanisms directly. "We'll build ours on top of theirs" is a bad policy.

[11.5.](#) List of Generic Authentication Systems

GSS-API [REF]

SASL [REF]

EAP [REF]

12. Sharing Authentication Information

In many cases, users will use the same authentication data for a large number of services. For instance, users may expect to use the same username/password pair for TELNET, IMAP, and FTP. In such cases, it is generally desirable for all such services to share a single set of authentication data. For instance, TELNET, IMAP, and FTP typically all share the same password database.

[12.1.](#) Authentication Services

This problem is made more difficult if the services which must share authentication data reside on different machines. This problem is typically solved (when it is solved, as opposed to simply ignored) by having some unique system which has the credentials. Such a machine may either provide authentication as service (as in Kerberos) or simply provide credentials to authorized machines (YP, NIS). In either case, this protocol needs to be secured.

[12.2.](#) Single Sign-On

A related problem is that users don't necessarily want to have to manually authenticate each time some service wants authentication. Rather, they want to authenticate once and have software take care of the rest. This capability is called SINGLE SIGN-ON. If all authentication will be performed by one program, this can be fixed simply by having the program cache the user's credentials. If credentials need to be shared across multiple services then it's necessary to have some way to pass them from the program which first authenticates to others (or to have some central credential manager). This service is generally called SINGLE SIGN-ON.

As a special case, consider the case where mutually suspicious systems all want to allow a user to authenticate with a single set of credentials. If certificate-based authentication is being used, this is relatively straightforward. In the case where passwords are being used, the typical solution is to have some third party authentication service which authenticates the user and then vouches for the user to the services. Microsoft Passport is one such provider.

[12.3.](#) Case Study: RADIUS

[TODO]

[12.4.](#) Case Study: Kerberos

Kerberos [REF] is a popular authentication/single sign-on service, especially in academic environments. Kerberos is based on the Needham-Schroeder authentication protocol. The authentication server role

Rescorla

[Page 24]Intern

is played by a Key Distribution Center (KDC). When a client first signs on the client proves its identity to the KDC, usually by means of a password shared with the KDC.

Kerberos is unusual in that the authentication service is provided to the client rather than the server. When a client wishes to communicate with a server, it first contacts the KDC and acquires a TICKET. That ticket contains a new symmetric key encrypted for both the client and server. The client can transmit the ticket to the server and use it both to prove its identity and establish a secure channel.

[12.5.](#) List of Authentication Server Systems

Kerberos [REF]

RADIUS [REF]

DIAMETER [REF]

???

[13. Guidance for Protocol Designers](#)

Adding authentication to protocols is difficult and is made even more difficult by the large number of options. This section attempts to provide some guidance to protocol designers. No single document can tell you how to build a secure system, but the following guidelines provide generally good advice. If you feel you need to violate one of these rules of thumb, make sure you know why you're doing it.

[13.1. Know what you're trying to do](#)

The first thing to do is figure out what the security problem you're trying to solve is. Questions to ask include.

[13.1.1. What's my threat model?](#)

Sorting out the threat model is always the first step in deciding what sorts of security mechanisms to use. In the case of authentication you must consider, at minimum. eP

1. What will be the result of various forms of attack?
2. Does the threat model include active attack. (Hint: it should.)
3. Do I need protection for my data or just the authentication. (Hint: probably you do).
4. How valuable is the data being secured? Are exhaustive computational attacks practical?
5. How competent are my users going to be?

[13.1.2. How many users will this system have?](#)

In general, the difficulty of managing a system scales with (or greater than) the number of users. This means that mechanisms which are practical with a small number of users may simply have too much overhead with a large number of users. For example, many token-based solutions charge by the token, which may be a prohibitive expense if there are many users.

[13.1.3. What's my protocol architecture?](#)

In some systems (e.g. POP, IMAP, TELNET), clients connect directly to the server. In others (e.g. HTTP, SIP, RSVP, BGP), authentication may need to be established over multiple hops when the entities have no

independent authentication. Each case requires a different strategy. See Section XXX for more discussion on this topic.

[13.1.4.](#) Do I need to share authentication data

If authentication data needs to be shared, especially between multiple servers, it's generally worth considering some sort of authentication server or using certificates.

[13.2.](#) Use As Few Mechanisms as You Can

Preferably, systems should have only one form of authentication. The more methods of authentication a system allows, the more things there are to go wrong. Remember that a chain is only as strong as its weakest link. In general, there are two reasons why systems allow more than one authentication mechanism. The first is that you're retrofitting a system which already has a large number of authentication mechanisms which cannot be displaced. The second is that users have widely different environments which for some reason cannot use the same authentication mechanism conveniently (e.g. some users have tokens and some do not).

Naturally, designers need to take such considerations into account but they should take reasonable steps to minimize the number of mechanisms. Designers should take special care to minimize the number of equivalent mechanisms. For instance, a system that provides a challenge/response mechanism and a public key based mechanism is a reasonable design, one that provides three different challenge/response mechanisms is not.

This doesn't mean that designers should not use security frameworks where multiple mechanisms are appropriate, but it does mean that they should be avoided unless necessary. Where generic security frameworks are used, they supported mechanisms should be carefully profiled to a

minimal set.

[13.3.](#) Avoid simple passwords

It's widely known that simple plaintext passwords are unsafe, but what's less widely known is that merely providing such a scheme can weaken systems even if stronger mechanisms are present. Consider the case where a system uses a negotiation framework that allows passwords. A downgrade attack can force the user to reveal his password even if both client and server support stronger mechanisms. Accordingly, designers should avoid deploying simple password mechanisms if

at all possible, not just provide stronger mechanisms.

[13.4.](#) Avoid inventing something new

Despite the large number of mechanisms we've discussed, this document describes only a small number of the available authentication mechanisms. There are very few situations in which designers cannot use some preexisting mechanism. This is vastly preferable to designing their own version of one of the standard mechanisms. In particular, designers should avoid designing their own channel security systems. If you want a channel security system, use IPsec or SSL.

[13.5.](#) Use the strongest mechanisms you can

Having the strongest security you can apropos is generally a good plan. It's particularly good advice here, since passwords in the clear, one-time passwords, challenge-response and zero-knowledge password proofs all require the user to have the same kind of credential: a password. (Note that some OTP schemes such as SecureID require a token.) When designing a new system, the ability to provide a familiar interface to a user is valuable, minimizing additional work for client and server implementors is not.

[13.6.](#) Consider providing message integrity

Although most of the authentication mechanisms we've described are themselves resistant to active attacks, many are subject to hijacking after authentication has completed. If your threat model includes active attack (it should), you should strongly consider providing message integrity for all of your protocol messages in order to prevent hijacking.

[14.](#) Scenarios

Despite the proliferation of authentication mechanisms, there are generally one or two optimal mechanisms for each scenario. We attempt to describe those mechanisms here. This section is divided into two

parts, attacking the problem from different angles. In the first, we consider the various kinds of capabilities entities might have and the best mechanisms to use with those credentials. In the second part we discuss a number of different protocol architectures and the potential mechanisms which can be used with those architectures.

[14.1.](#) Capability Considerations

There are three primary authentication scenarios:

- (1) Neither side has a public/private key pair.
- (2) The server has an authenticated key pair (either via a certificate or prior arrangement).
- (3) Both sides have authenticated key pairs

Despite the proliferation of authentication mechanisms, there are only one or two best mechanisms for each scenario. We describe them here.

14.1.1. Neither side has a public/private key pair

Three basic strategies are suitable for the situation where neither side has a key pair: challenge/response, one-time passwords, and ZKPPs. The only situation in which OTP systems are superior to challenge/response systems is when adapting a legacy system in which it is difficult to change the client software. If the client software can be changed, challenge/response offers roughly equivalent security with significantly less management complexity. ZKPP proofs are technically superior to challenge/response but intellectual property considerations make them unsuitable for Internet standardization unless no other option is available.

These considerations make challenge/response the best choice for this scenario. If at all possible, it should be performed under cover of an anonymous key exchange, as described in section XXX. With this adaptation, an attacker needs to mount an active attack in order to dictionary search the password space.

14.1.2. The server has an authenticated key pair

If the server has a key pair which the client can authenticate, then simple username/password encrypted under the server's public key is the preferred authentication mechanism. Challenge/response is in fact weaker in this case because it is password equivalent. Once confidentiality is provided, OTP and ZKPP systems offer significant additional management complexity for marginal security benefit.

14.1.3. Both sides have authenticated key pairs

If both sides have key pairs, the optimal mechanism is mutual public key authentication.

[14.2.](#) Architectural Considerations

In this section, we consider XXX different network architectures and the authentication mechanisms that are most suitable for each.

[14.2.1.](#) Simple Client/Server

The simplest authentication scenario is where the client and the server are connected by some interactive connection. Mercifully, this situation is quite common in such protocols as IMAP, TELNET, etc. In the simple client/server case, mostly any authentication mechanism can be employed and so the choice depends on other factors, such as what credentials are available and the degree of security required.

[14.2.2.](#) Proxied Client/Server

It's quite common for client/server communication to be propagated through some gateway, as happens with HTTP. This situation has two potential authentication problems.

1. How does the client authenticate to the proxy so that the proxy knows to serve it.
2. How does the client authenticate to the server with the proxy in the way.

The problem of authenticating to the proxy looks essentially like the ordinary client/server authentication problem (except in the case where there are multiple proxies in which case authenticating to anything other than the first hop proxy looks rather like problem 2.)

The problem of authenticating through the proxy is rather more difficult. The obstacle is that neither client nor server may not trust the proxy. They therefore need to provide an authentication method (preferably with message integrity) that doesn't require trusting the proxy. This rules out simple passwords and makes one-time passwords extremely questionable. There are three basic strategies available.

[14.2.2.1.](#) Tunnel

If the client and the server establish a tunnel through the proxy then they can behave as if this was an ordinary client/server transaction. Although this rather obviates the point of having a proxy,

it's still a popular strategy and is used with HTTPS. [REF] Since the proxy is untrusted, the application protocol must either be run over a secure channel or hardened against active attacks.

[14.2.2.2. Challenge/Response](#)

A shared symmetric key between client and server can be used for authentication even in the face of a proxy by using standard challenge/response methods (with appropriate protocol modifications to distinguish between protocol data units (PDUs) directed towards the proxy and those directed towards endpoints.) These methods should include integrity protection for the individual PDUs.

On a small scale, this technique works (it's what's used in HTTP when HTTPS is not used) but it quickly becomes unwieldy. If there are a large chain of proxies each of which wishes to authenticate the client, server, other proxies or all three, an enormous number of pairwise keys need to be established and maintained. In a protocol where long proxy chains are expected, symmetric key based authentication is probably impractical.

A variant of this technique is to use a message-based system with symmetric keying such as S/MIME. All PDUs can then be encapsulated in secure messages. Recursive encapsulation can be used to provide authentication to proxies.

[14.2.2.3. Digital Signatures](#)

The final approach is to use public-key based digital signatures. Each endpoint signs each message (possibly with some set of nonces to prevent replay attack). The disadvantage of this approach is that it requires a PKI. The advantage is that it doesn't require pairwise keys. Each proxy in the chain can validate the client and the server based solely on their signatures.

[14.2.3. Store and Forward](#)

A number of important IETF protocols, most importantly, e-mail, are of the store and forward messaging variety. Such protocols have roughly the same security options as proxied protocols except that tunnelling is no longer possible. Additionally, since store and forward protocols are non-interactive, many of the usual challenge/response techniques for preventing replay attack no longer work and so care must be taken to either make one's system idempotent or introduce a specific anti-replay mechanism. The standard technique for store-and-forward situations is message security a la S/MIME.

[14.2.4](#). Multicast

A number of IETF protocols have the property that multicast or broadcast message integrity needs to be provided. For example, routing and DNS both require the ability for a single sender to broadcast authenticated and integrity protected messages to a large number of receivers. There are two relevant cases: In the first, all members of the group are trusted and so it's feasible to have some group key which is used for authenticating all transmissions. This group key may be manually configured or established via some protocol such as GKMP [REF].

In the second case, individual group members are not trusted not to forge messages. such systems, it's not really practical to use symmetric key systems because the sender would need to agree on a key with each recipient (there may not even be a return channel). The only really practical approach in these multicast situations is for the sender to digitally sign each transmission with its private key.

Acknowledgments

Early versions of this document were reviewed by Fred Baker, Lisa Dusseault, Ted Hardie, and Mike St. Johns.

References

[\[TODO\]](#)

Security Considerations

This document describes a number of security mechanisms.

Author's Address

Eric Rescorla <ekr@rtfm.com>
RTFM, Inc.
[2064](#) Edgewood Drive
Palo Alto, CA 94303
Phone: (650)-320-8549

Table of Contents

1. Introduction	2
2. The Authentication Problem	2
2.1. Authorization vs. Authentication	2
2.2. Standard Material Something you have, something you know...	2
3. Description of Authentication Mechanisms	2
4. Passwords In The Clear	3
4.1. Password Sniffing	3
4.2. Post-Authentication Hijacking	3
4.3. Online Password Guessing	4
4.4. Offline Dictionary Attack	4
4.4.1. Shadow Passwords	4
4.4.2. Iteration	5
4.4.3. Salting	5
4.4.4. Stronger Passwords	5
4.5. Case Study: HTTP Basic Authentication	6
4.5.1. Password Caching	6
4.5.2. Pro-active authentication	6
4.6. List of Systems that Use Passwords in the Clear	6
5. One Time Passwords	7
5.1. Case Study: S/Key and OTP	7
5.1.1. Race Conditions	8
5.2. Case Study: SecureID	9
5.3. List of One-Time Password Systems	9
6. Challenge/Response	9
6.1. Offline Attacks on Challenge/Response	10
6.2. Password File Compromise	10
6.3. Case Study: CRAM-MD5	11
6.4. Case Study: HTTP Digest	12
6.4.1. Message Integrity	12
6.4.2. Replay Attack	13
6.4.3. Downgrade Attack	13
6.5. List of Challenge-Response Systems	14
7. Anonymous Key Exchange	14
7.1. Case Study: SSH Password Authentication	14
7.2. Case Study: TLS Anonymous DH + Passwords	15
7.3. List of Anonymous Key Exchange Mechanisms	15
8. Zero-Knowledge Password Proofs	15
8.1. Intellectual Property	16
8.2. List of Zero Knowledge Password Proof Systems	16

9. Server Certificates plus Client Authentication	16
9.1. Case Study: Passwords over HTTPS	17
9.1.1. Authentication State	18
9.1.1.1. The Token Problem	18
9.1.1.2. URL Rewriting	18

9.1.1.3. Cookies	19
9.1.1.4. HTTPS Session Binding	19
9.2. List of Server Certificate Systems	20
10. Mutual Public Key Authentication	20
10.1. Password Equivalence	20
10.2. Authentication between Unknown Parties	20
10.3. Key Storage	21
10.4. Tokens	21
10.5. Password Derived Keys	21
10.6. Case Study: SMTP over TLS	21
10.7. List of Mutual Public Key Systems	21
11. Generic Authentication Mechanisms	22
11.1. Downgrade Attacks	22
11.2. Integration with Applications	22
11.3. Multiple Equivalent Mechanisms	22
11.4. Excessive Layering	23
11.5. List of Generic Authentication Systems	23
12. Sharing Authentication Information	24
12.1. Authentication Services	24
12.2. Single Sign-On	24
12.3. Case Study: RADIUS	24
12.4. Case Study: Kerberos	24
12.5. List of Authentication Server Systems	25
13. Guidance for Protocol Designers	25
13.1. Know what you're trying to do	25
13.1.1. What's my threat model?	25
13.1.2. How many users will this system have?	26
13.1.3. What's my protocol architecture?	26
13.1.4. Do I need to share authentication data	26
13.2. Use As Few Mechanisms as You Can	26
13.3. Avoid simple passwords	27
13.4. Avoid inventing something new	27
13.5. Use the strongest mechanisms you can	27
13.6. Consider providing message integrity	27
14. Scenarios	27
14.1. Capability Considerations	28
14.1.1. Neither side has a public/private key pair	28
14.1.2. The server has an authenticated key pair	28
14.1.3. Both sides have authenticated key pairs	29
14.2. Architectural Considerations	29

14.2.1.	Simple Client/Server	29
14.2.2.	Proxied Client/Server	29
14.2.2.1.	Tunnel	29
14.2.2.2.	Challenge/Response	30
14.2.2.3.	Digital Signatures	30
14.2.3.	Store and Forward	30
14.2.4.	Multicast	31
14.2.4.	Acknowledgments	31

14.2.4.	References	31
	Security Considerations	31
	Author's Address	31