

Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: August 31, 2010

E. Rescorla
RTFM, Inc.
G. Lebovitz, Ed.
Juniper Networks, Inc.
Internet Architecture Board
IAB
February 27, 2010

A Survey of Authentication Mechanisms
draft-iab-auth-mech-07.txt

Abstract

Authentication is a common security issue for the design of Internet protocols. A wide variety of authentication technologies are available. A common problem is knowing which technology to choose or which of a variety of essentially similar implementations of a given technique to choose. This memo provides a survey of available authentication mechanisms and guidance on selecting one for a given protocol.

Status of this Memo

This Internet-Draft is submitted to IETF in full conformance with the provisions of [BCP 78](#) and [BCP 79](#).

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."

The list of current Internet-Drafts can be accessed at <http://www.ietf.org/ietf/lid-abstracts.txt>.

The list of Internet-Draft Shadow Directories can be accessed at <http://www.ietf.org/shadow.html>.

This Internet-Draft will expire on August 31, 2010.

Copyright Notice

Copyright (c) 2010 IETF Trust and the persons identified as the

Internet-Draft

Authentication Mechanisms

February 2010

document authors. All rights reserved.

This document is subject to [BCP 78](#) and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the BSD License.

This document may contain material from IETF Documents or IETF Contributions published or made publicly available before November 10, 2008. The person(s) controlling the copyright in some of this material may not have granted the IETF Trust the right to allow modifications of such material outside the IETF Standards Process. Without obtaining an adequate license from the person(s) controlling the copyright in such materials, this document may not be modified outside the IETF Standards Process, and derivative works of it may not be created outside the IETF Standards Process, except to format it for publication as an RFC or to translate it into languages other than English.

Internet-Draft

Authentication Mechanisms

February 2010

Table of Contents

| | | |
|------------------------|-------------------------------------------------------------------------------|--------------------|
| 1. | Introduction | 6 |
| 2. | The Authentication Problem | 7 |
| 2.1. | Authorization vs. Authentication | 7 |
| 2.2. | Authentication Building Blocks | 8 |
| 2.3. | Clients and Servers | 8 |
| 3. | Basic Authentication, Attacks, and Counter Measures | 9 |
| 3.1. | Password Sniffing | 9 |
| 3.2. | Post-Authentication Hijacking | 9 |
| 3.3. | Online Password Guessing (aka Brute Force Attack) | 10 |
| 3.4. | Offline Dictionary Attack | 10 |
| 3.4.1. | Shadow Passwords | 11 |
| 3.4.2. | Iteration | 11 |
| 3.4.3. | Salting | 11 |
| 3.4.4. | Stronger Passwords | 12 |
| 3.4.5. | Encrypted Channel | 12 |
| 3.5. | Phishing | 12 |
| 3.6. | Case Study: HTTP Basic Authentication | 13 |
| 3.6.1. | Password Caching | 13 |
| 3.6.2. | Proactive authentication | 13 |
| 3.7. | List of Systems that Use Passwords in the Clear | 13 |
| 4. | One Time Passwords | 14 |
| 4.1. | Case Study: S/Key and OTP | 15 |
| 4.1.1. | Race Conditions | 16 |
| 4.2. | Case Study: SecurID | 17 |
| 4.3. | List of One-Time Password Systems | 17 |
| 5. | Challenge/Response | 18 |
| 5.1. | Offline Attacks on Challenge/Response | 19 |
| 5.2. | Password File Compromise | 19 |
| 5.3. | Case Study: CRAM-MD5 | 20 |
| 5.4. | Case Study: HTTP Digest | 21 |
| 5.4.1. | Message Integrity | 21 |
| 5.4.2. | Replay Attack | 22 |
| 5.4.3. | Downgrade Attack | 22 |
| 5.5. | List of Challenge-Response Systems | 23 |

| | | |
|------------------------|-------------------------------------------------------------------------|--------------------|
| 6. | Anonymous Key Exchange | 23 |
| 6.1. | Case Study: SSH Password Authentication | 24 |
| 6.2. | List of Anonymous Key Exchange Mechanisms | 25 |
| 7. | Zero-Knowledge Password Proofs | 25 |
| 7.1. | Intellectual Property | 26 |
| 7.2. | List of Zero Knowledge Password Proof Systems | 26 |
| 8. | Server Certificates plus User Authentication | 26 |
| 8.1. | Case Study: Passwords over HTTPS | 28 |
| 8.1.1. | Authentication State | 28 |
| 8.2. | List of Server Certificate Systems | 30 |
| 9. | Mutual Public Key Authentication | 30 |
| 9.1. | Password Equivalence | 31 |

| | | |
|-------------------------|-------------------------------------------------------------------|--------------------|
| 9.2. | Authentication between Unknown Parties | 31 |
| 9.3. | Key Storage | 32 |
| 9.4. | Tokens | 32 |
| 9.5. | Password Derived Keys | 32 |
| 9.6. | Case Study: SMTP over TLS | 33 |
| 9.7. | List of Mutual Public Key Systems | 33 |
| 10. | Generic Issues | 33 |
| 10.1. | Channel Security Protocols | 33 |
| 10.1.1. | Limited Authentication Options | 34 |
| 10.1.2. | Limited Application Integration | 34 |
| 10.1.3. | List of Channel Security Protocols | 34 |
| 10.2. | Authentication Frameworks | 35 |
| 10.2.1. | Downgrade Attacks | 37 |
| 10.2.2. | Multiple Equivalent Mechanisms | 37 |
| 10.2.3. | Channel Bindings | 40 |
| 10.2.4. | Excessive Layering | 41 |
| 10.2.5. | List of Authentication Frameworks | 42 |
| 11. | Sharing Authentication Information | 42 |
| 11.1. | Authentication Services | 42 |
| 11.2. | Single Sign-On | 43 |
| 11.3. | Case Study: RADIUS | 43 |
| 11.4. | Case Study: Kerberos | 44 |
| 11.5. | List of Authentication Server Systems | 44 |
| 12. | Guidance for Protocol Designers | 44 |
| 12.1. | Know what you're trying to do | 44 |
| 12.1.1. | What's my threat model? | 45 |
| 12.1.2. | How many users will this system have? | 45 |
| 12.1.3. | What's my protocol architecture? | 45 |
| 12.1.4. | Do I need to share authentication data? | 46 |

| | | |
|-------------------------|------------------------------------------------------|--------------------|
| 12.2. | Use as few mechanisms as you can | 46 |
| 12.3. | Avoid simple passwords | 47 |
| 12.4. | Avoid inventing new frameworks | 47 |
| 12.5. | Use the strongest mechanisms you can | 47 |
| 12.6. | Consider providing message integrity | 48 |
| 13. | Scenarios | 48 |
| 13.1. | Capability Considerations | 48 |
| 13.1.1. | Neither side has a public/private key pair | 49 |
| 13.1.2. | One side has an authenticated key pair | 49 |
| 13.1.3. | Both sides have authenticated key pairs | 50 |
| 13.2. | Architectural Considerations | 50 |
| 13.2.1. | Simple Connection | 50 |
| 13.2.2. | Proxied Client/Server | 50 |
| 13.2.3. | Store and Forward | 52 |
| 13.2.4. | Multicast | 52 |
| 14. | Acknowledgements | 53 |
| 15. | References | 53 |
| 15.1. | Normative References | 53 |
| 15.2. | Informative References | 53 |

| | | |
|-----------------------------|---------------------------------------------------|--------------------|
| Appendix A. | IAB Members at the time of this writing | 59 |
| | Authors' Addresses | 60 |

1. Introduction

Authentication is perhaps the most basic security problem for designers of network protocols. Even the early Internet protocols such as TELNET [[RFC0854](#)] and FTP [[RFC0959](#)], which provided no other security services, made provision for user authentication. Unfortunately, these early authentication systems were wholly inadequate for the Internet Threat Model [[RFC3552](#)] 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. This document is aimed at protocol designers, more so than system deployers.

In [section 2](#) we will review the problem space around authentication. It will define what we are trying to accomplish when using authentication mechanisms, will contrast authorization and authentication, and review the basic building blocks of authentication.

In [section 3](#) we introduce basic password authentication systems, describe attacks that formed against them, and the counter measures that addressed such attacks. These descriptions will lay a foundation of understanding and terminology useful for the mechanisms described in the sections that follow [section 3](#).

The next six sections, starting with [section 4](#), 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 uses 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 known Protocols/Systems that use this mechanism

In order to understand the pros and cons of each mechanism, it's important to have a clear idea of the threat model for the environment in which your protocol will be deployed. [[RFC3552](#)] provides more information on threat modelling.

[2.](#) The Authentication Problem

The authentication problem is simple to describe but hard to solve: Two parties (or devices/daemons/endpoints/etc.) are communicating and one wishes to establish its identity to another, or establish the identity of another. The basic scenario is exemplified by TELNET [[RFC0854](#)]. 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. Note that the major technical division here is between situations in which there is any client-side storage (for instance for cryptographic keys) and ones where any authentication material must be memorized.

[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 often 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 validates the identity of a party -- that the party really is who they claim to be; authorization defines whether they can perform a certain action.

Authorization necessarily relies on authentication, but authentication alone does not imply authorization. Having already

successfully passed some authentication step, but 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 the first step, authentication.

[2.2.](#) Authentication Building Blocks

The classic formulation of authentication is that there are three different elements that can be employed, alone or in combinations together::

1. Something you have--a physical token like a key.
2. Something you know-- something known only to you, not recorded anywhere where another could obtain and use it, a secret, e.g., a password
3. Something you are--some physical characteristic unique to you, e.g. a thumbprint.

The best authentication mechanisms combine two or more of these mechanisms. For instance, if you use a driver's license or a passport to authenticate, that's something you have (the license) and something you are (your resemblance to the picture on the license). In practice, biometric authentication mechanisms work poorly over the Internet. Biometric authentication mechanisms work best where the relying party can directly verify the presence of the person being authenticated. In general this is not possible over the Internet because the relying party does not control the authenticating party's computer. Thus, it is difficult to distinguish real authentications from replay attacks mounted by attackers who have captured the user's biometric information. So the best Internet authentication mechanisms will involve a token plus a secret.

[2.3.](#) Clients and Servers

Most of the protocols which run on the Internet are inherently asymmetric, with one peer taking the role of the client and the other the server. Because the servers are generally fixed machines with a fixed IP address and the clients may have any IP address, many protocols (such as SSH or TLS) operate by attempting to authenticate the server first and, if that succeeds, to then authenticate the client. This occurs because the client wants to disclose its identity (and certainly its credentials) only to the "real" server, not an imposter. When such systems are used in peer-to-peer contexts, it is still necessary for one peer to take on the client role and one the server. Typically, the party which spoke first (the initiator) is treated as the client.

Even protocols which are peer to peer (such as IKE) require one party

to speak first. In such cases, it's appropriate to refer to that party as the initiator and the other as the responder.

[3.](#) Basic Authentication, Attacks, and Counter Measures

This section will describe the early forms of authentication systems, the attacks they attracted, and the counter measures taken. These descriptions will lay a foundation of understanding and terminology that is useful for understanding the mechanisms in the sections that follow it.

The most basic form of authentication is for the client to provide a username/password pair to the server. 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 described by Wilkes [[Wil68](#)], is to store the output of a one-way function (typically a cryptographic message digest (see [[RFC4949](#)] for a definition of terms like this)) of the password instead of the password itself. When the server needs to verify a password, it computes the function using the password as input and compares the output against the stored output. Because the function is one way, the server cannot recover the user's password from the password file. Therefore, anyone gaining access to the password hash file also cannot recover the user's password from the password file.

[3.1.](#) Password Sniffing

The simplest attack against passwords delivered by client 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.

[3.2.](#) 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.

This attack can be prevented. The connection must use some form of per-packet (or per segment) cryptographic authentication which can be neither mimicked nor guessed by a man-in-the-middle. More on this later.

3.3. Online Password Guessing (aka Brute Force Attack)

Extensive experience [[Klein90](#)] 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 [[Worm88](#)]

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 60 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.

3.4. Offline Dictionary Attack

Even if digested password files are used, it is still often possible for an attacker who recovers the digested password, or password file, to discover user's passwords. The attacker can mount an OFFLINE DICTIONARY ATTACK on the password or 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 the classic Crack [[Crack](#)] program.

An OFFLINE DICTIONARY ATTACK can occur in two ways. First, using

sniffing, an attacker who sits along the path of packets between two hosts can observe the transmission, extract a digested password, and work offline to find a matching password. A second variation occurs when an attacker recovers the entire digested password file for a system.

There are four basic countermeasures to offline dictionary attack.

[3.4.1.](#) Shadow Passwords

The first countermeasure to an offline dictionary attack 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 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.

[3.4.2.](#) 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.

[3.4.3.](#) 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

Rescorla, et al.

Expires August 31, 2010

[Page 11]

Internet-Draft

Authentication Mechanisms

February 2010

slowing down the process (thereby making iteration unnecessary).

[3.4.4.](#) 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 stronger passwords. This can be done reactively by running a password cracker on your system or proactively by forcing users to use stronger 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.

After a lengthy discussion about passwords and their entropy, NIST 800-63 [[SP800-63](#)], [Appendix A](#), page 52 suggests a system that uses:

- o a minimum of 8 character passwords, selected by subscribers from

- o an alphabet of 94 printable characters,
- o required subscribers to include at least one upper case letter, one lower case letter, one number and one special character, and;
- o Used a dictionary to prevent subscribers from including common words and prevented permutations of the username as a password.

[3.4.5.](#) Encrypted Channel

Another countermeasure is to deny inline attackers a view of the password. TLS is an example of a protocol that provides for this. Using certificates (described in a different section below), the client first verifies the identity of the server, then the two establish an encrypted channel. Once the encryption is in place, the HTTP authentication occurs in which the client's password digest is sent to the server. The TLS encryption prevents an attacker from seeing the authentication digest, and thus from attempting an OFFLINE DICTIONARY ATTACK.

[3.5.](#) Phishing

Even an attacker with no access to the victim's network can capture a user's password with a social engineering attack (often called PHISHING). In the basic attack, the attacker sends the victim an

email posing as some site that the victim has a relationship with (e.g., eBay, a bank, a web-based email service, or a corporate IT-Support Helpdesk) and containing a URL [[RFC4248](#)], [[RFC4266](#)] for the user to dereference. When the user dereferences the URL he is prompted for his password, which he often supplies. (See also [[DTH06](#)]).

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

HTTP basic authentication [[RFC2617](#)] 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 WWWAuthenticate 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 due to transport reencoding.

[3.6.1.](#) 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 multiuser machines, even ones which provide strong file permissions, since administrators can still read such cache files.

[3.6.2.](#) Proactive authentication

Requesting a page, receiving an authentication challenge and rerequesting 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 proactively send their cached passwords whenever accessing any URL deeper than the URL for which they were originally prompted.

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

Rescorla, et al. Expires August 31, 2010 [Page 13]

Internet-Draft Authentication Mechanisms February 2010

- FTP (when none of GSSAPI-KRB5, SRP, or TLS are negotiated)
- TELNET (when neither AUTH or STARTTLS are used)
- HTTP (basic authentication)
- SASL (password mode)
- RLOGIN
- POP (among others mechanisms)
- IMAP (among other mechanisms)
- (too many others to mention)

The next six sections each describe a single class of authentication technology, according to the format described in the introduction

Section 1

4. One Time Passwords

The simplest approach to preventing sniffing attacks on passwords is to use ONE TIME PASSWORDS. In its basic form, the user is provided with a password (or list of passwords) that 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.

One-time passwords are generally used in one of two ways: a single one-time password used to bootstrap a trust relationship, or a continuous use of one-time passwords, i.e. new one-time password used at each login.

A one-time password may be used to bootstrap a trust relationship. For example, a user might be given a single one-time password to access a system. Once authenticated to that system, the user will be asked to supply a new, personalized password that will be used from then on. Similarly, a one-time password can be used to authenticate a first exchange of long-term keys (e.g. asymmetric keys) between two parties. This system is particularly good when many users or end-point machines will be connecting to a well-known, central system, but the user/end-points are not pre-known to the server. An example is a smart-phone-based email client doing a first time registration from the a new smart phone to the provider's server. The phone will ship with the provider's server's IP address, and public key. pre-installed The user will be given the one-time password and enter it via the smart phones UI on the first connection only. The user initiates a secure connection to the server, and uses the server's

verification material to confirm the server's identity. It then uses the one-time password to authenticate itself to the server. It will then send it's public key (from a pre-computed asymmetric key pair) to the server, again, authenticated by the one-time password. The

server verifies the received messages using the one-time password, and confirms the identity of the host. It can then trust the credentials, the public key, passed to it by the host. Henceforth the two parties will authenticate one to the other using their public keys. The one-time password is expired and discarded.

The second main use of one-time passwords occurs when the user (generally) has 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.

The remainder of this section will describe three specific case studies of continuous use one-time password implementations available today: S/Key, OTP and SecurID. None of these 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. SecurID 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.

4.1. Case Study: S/Key and OTP

S/Key [[RFC1760](#)], invented by Neil Haller and Phil Karn, is a straightforward one time password system that uses some clever implementation tricks. One-Time Passwords (OTP) [[RFC2289](#)] 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:

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

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 defined 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 mode, 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 secret).

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.

4.1.1. 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 at timeout on any such pending connection. OTP implementations are required to implement this or some other countermeasure.

[4.2.](#) Case Study: SecurID

Probably the most commonly deployed commercial one time password implementation is SecurID, sold by RSA Security Inc. Instead of using a fixed list of keys, SecurID uses a time-dependent, six digit key. The user has a token with an LCD displaying a pseudo-random number. That number changes at an interval between 30 seconds and 2 minutes and is synchronized with an authentication server located at the server. SecurID also has the advantage that it employs two-factor authentication (as describe above [Section 2.2](#)), combining something you have, the token, with something only you know, a personal identification string, or PIN.

In order to authenticate the user enters both his PIN and the time-dependent key (they can be concatenated so that this is transparent to the client program.) For example where the user's PIN is "2Enl0*/b" and the OTP on the token currently reads "041 980", the user will simply enter in the password field of the application "2Enl0*/b041980". This value will then be used as the password for whatever authentication function the client and server are using. The server verifies the password and checks that the time-dependent key is correct for the current time and only then allows login. If sent in the clear (versus being used in some challenge/response mechanisms; see next section), 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 (unless the replay is executed within the same time window as the current value). When sent using a cryptographic message digest function, this weakness is mitigated.

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

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

Single One-Time-Password used for bootstrapping
S/Key [[RFC1760](#)]

Rescorla, et al.

Expires August 31, 2010

[Page 17]

Internet-Draft

Authentication Mechanisms

February 2010

OTP [[RFC2289](#)]
SecurID [[RFC2808](#)]

5. 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 $H(\text{challenge} || \text{key})$, where H is a cryptographic message digest and $||$ denotes message concatenation. A better design is probably to use HMAC [[RFC2104](#)] 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.

There also explicit challenge response systems, where users are stepped through a challenge and response exchange where they must use a one-time-password token system. For example, as a user attempts to login to a Telnet session, the server sends a challenge. The user sees this challenge appear on the screen. Holding an electronic one-time-password token, she uses its key pad to key in that challenge. The token responds by displaying a one-time response to that challenge. The user concatenates the token's response with her PIN and sends it back over the Telnet session login. The server checks that both the correct PIN and one-time-password are used.

A challenge-response system can also be turned into a secure channel protocol by using the shared key to establish cryptographic keys

which are then used to encrypt the traffic. In this context, a CHANNEL is a security association between two endpoints through which further security exchanges may occur. A secure channel would be offer encryption, authentication and integrity-checking on a per-packet basis. TLS-PSK [[RFC4279](#)] is one example of such a system.

[5.1.](#) 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. This problem is inherent unless public-key methods are incorporated within the challenge-response protocol, as will be discussed in Sections [6](#) and [7](#), or the challenge-response transactions are carried over secure channels (which themselves must be authenticated).

[5.2.](#) 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:

$$\text{STORED} = H(\text{PASSWORD} || \text{SALT})$$
$$\text{RESPONSE} = H(\text{STORED} || \text{CHALLENGE})$$

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.

SCRAM [[I-D.newman-auth-scram](#)] describes one defense against this sort of attack. The server stores a hashed version of the password, and must prove that it knows it using challenge-response. The client then provides the preimage for the hashed password, thus demonstrating that it knew the original password. With this system, an attacker who recovers the password file can immediately impersonate the server to the client, but not the client to the server. However, if he impersonates the server to the client he can capture the preimage and can then impersonate the client to the server.

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

CRAM-MD5 [[RFC2195](#)] is a challenge/response authentication extension for IMAP [[RFC3501](#)] 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 email address [[RFC5322](#)] 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 postauthentication 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).

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

HTTP Digest Authentication [[RFC2617](#)] is a replacement for HTTP's 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 (the field specifying them is 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.

[5.4.1.](#) 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 protect 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 [[RFC2660](#)], but Digest doesn't do so.

Digest deployment has been somewhat spotty in the past. For instance, the popular Netscape Navigator 4 versions did not support it. More recently, Internet Explorer 7.0 +, Mozilla Firefox 2.0+, Netscape 7+ all support digest authentication. SIP [[RFC3261](#)] requires Digest authentication and it is near universal there.

[5.4.2.](#) 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").

[5.4.3.](#) 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 send Basic at all--note that the server refusing to accept it does not help, since the attacker can impersonate the server.

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.

5.5. List of Challenge-Response Systems

APOP [[RFC1939](#)]
HTTP Digest [[RFC2617](#)]
AKA [[AKA](#)]
CRAM-MD5 [[RFC2195](#)]
Kerberos password-based authentication [[RFC4120](#)]

6. 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 peers arrange for a secure channel using a key establishment mechanism that does not authenticate either side. Public key algorithms such as Diffie-Hellman and RSA can be used in this way. Once the key is established you can encrypt all the traffic. and any data which is transmitted over the channel is secure from eavesdroppers. This includes data such as passwords or authenticators.

The problem with this system is that it's subject to what's called a man-in-the-middle (MITM) attack. Because the cryptographic key establishment mechanism is unauthenticated, it is possible for an attacker to intercept communications between the peers (say Alice and Bob) and pose as Alice to Bob and Bob to Alice. The attacker can then forward traffic between them and get access to whatever's being encrypted.

The MITM attack on Diffie-Hellman key exchange is shown in the following figure. Y_c , Y_a , and Y_s are used to denote the client, attacker, and server public keys respectively.

| Client | Attacker | Server |
|----------|----------|----------|
| ----- | ----- | ----- |
| Y_c -> | | |
| | Y_a -> | |
| | | <- Y_s |
| | <- Y_a | |

At the end of this exchange the client thinks that the server's public key is Y_a and the server thinks that the client's public key is Y_a . However, in reality both have established a shared secret with the attacker. Thus, when the password is transmitted over channel the attacker sees it.

So we see that with if only one side of the connection is CRYPTOGRAPHICALLY authenticated this attack cannot be detected. Thus, how much security you believe that anonymous key exchange adds to your protocol depends on your threat model. Active attacks are generally more difficult to mount than passive attacks but by no means impossible [[Bell89](#)]

All of these mechanisms use public key cryptography to perform the initial anonymous key exchange. As a result, performance can be unacceptably slow if one side (e.g., a handheld device) is heavily constrained. Such concerns were more relevant when the Diffie-Hellman technology first came to commercial maturity in the late 1990s. As processing speeds increase per Moore's Law, and as smart phones are given more and more powerful processors, this concern reduces. Most Internet servers are fast enough to keep up with the normal number of required authentications and hardware acceleration solutions are readily available. This is not to say that performance is of now concern. Moore's law has to some extent been counter-balanced by an increase in the size of common keys. 2048-bit keys are now quite common and the potential for even larger keys has lead to an increased interest in elliptic curve cryptography.

[6.1.](#) 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 may 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. Machines and their user interfaces can easily be made to perform this check in a predictable way. For example, if one machine uses SSH as a secure channel for management of a second machine, the application on the first machine can prompt the administrator for the second's fingerprint and not continue until the

string is received. It can then check upon every connection that what it sees from the second machine matches what was entered.

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.

Another option is to combine a one-time-password with public keys. In such a system, a host first authenticates itself to the other using the one-time-password, then, once the first SSH channel is established, securely passes a public key for long term use. On the next connection, the public key will be used, and the fingerprint of that key will be used to validate it.

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

SSH (password mode) [[RFC4251](#)]

IPsec using IKEv2 with unauthenticated public keys, aka BTNS
[[RFC5386](#)]

SSL/TLS (anonymous keying) [[RFC5246](#)]

[7.](#) 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 [[Bell92](#)], designed by Steve Bellovin and James Merritt. EKE is based on Diffie-Hellman, but instead of sending the key shares (the public keys) in the clear they are encrypted using a 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) -> | |

Rescorla, et al.

Expires August 31, 2010

[Page 25]

Internet-Draft

Authentication Mechanisms

February 2010

<- E(K, Challenge-a)

Where K is the DH shared secret $= g(X_a * X_b) \bmod p$, E(blah, blah) is encryption using the elements within the paranthesis, Ya = public key of client, Yb = public key of server, and Challenge-a and Challenge-b are random stings.

Note that EKE as described above is insecure against password file compromise, since the server must store the password. Augmented EKE [[Bell94](#)] describes a protocol that is secure against this. A large number of other ZKPPs have been proposed, including PDM [[KP01](#)], SPEKE [[Jab96](#)], and SRP [[RFC2945](#)]. These protocols are all roughly equivalent, offering slightly different combinations of security, performance, and message count.

[7.1.](#) Intellectual Property

From a technical perspective, ZKPPs dominate the anonymous key exchange mechanisms described in [Section 6](#). Their performance is roughly equivalent and their security guarantees are superior. The major ZKPPs are EKE, A-EKE, SPEKE, and SRP. there are a number of Intellectual Property Rights in this area, some of which are on file with the IETF (www.ietf.org/ipr).

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

EKE [[Bell92](#)]

A-EKE [[Bell94](#)]

PDM [[KP01](#)]
SPEKE [[Jab96](#)]
SRP [[RFC2945](#)]

8. Server Certificates plus User Authentication

If you can authenticate one side of the connection (typically a 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

Rescorla, et al.

Expires August 31, 2010

[Page 26]

Internet-Draft

Authentication Mechanisms

February 2010

when server authentication is available is to first establish the encrypted channel, and then send simple passwords over the encrypted channel. This includes systems such as passwords over SSL/TLS and passwords over IPsec VPNs.

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 3.2](#)). Second, it provides confidentiality and message integrity for the rest of the application traffic, which is in general a good thing.

This approach is especially applicable in client server systems where the server is well known and the clients are many. Examples include a web site being hit by many users, a remote access gateway serving many remote workers, or a content service being connected to by many subscribing applications. This approach is less well suited to peer-to-peer or mesh connections.

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 effort, configuration, and cost. The use of self-signed certificates can ease the operational and cost issues while preserving security as long as the sha-1 fingerprint of that certificate's key is both listed in the self-signed certificate and delivered in a secure and trusted way to the end-point. Self-signed certificates without explicit verification aren't acceptable in this case (rather, they reduce one to the anonymous key exchange scenario described in [Section 6](#)).

The more serious problem is establishing what the server side name in the certificate ought to be. Common practice (stemming from practice in HTTPS [[RFC2818](#)]) 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. Further, use of IP address is insufficient in cases where the "server" is actually a service intended to appear to users as one server, but in reality virtualized across several servers and IP addresses. Any protocol which uses this mechanism must specify a mechanism for determining the server's expected domain name.

One concern here is what happens if the server has a certificate that has the wrong name or that is signed by a Certificate Authority that

the user's client does not recognize. Many such (web browser) clients present a dialogue warning the user that the connection may be under attack and offering to let him ignore the error. The common Internet user will have no idea what this means, the implications, how to determine if the threat is real or just a configuration error, and therefore will not know how to react. The easiest path to their immediate goals is usually to chose the option to ignore the error. Obviously, if users do this routinely (and it is widely believed that they do) then they can be subject to an active attack.

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

Despite the existence of Digest Authentication, the dominant form of strong HTTP authentication is passwords with HTTP over SSL/TLS

(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.

[8.1.1.](#) 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.

[8.1.1.1.](#) 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.

[8.1.1.2.](#) 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 script, where the arguments include the real page to be accessed and the authenticator token. an example of such a URL is:

```
=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 its 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.

[8.1.1.3.](#) Cookies

The inconvenience of URL rewriting lead to the introduction of HTTP Cookies [[RFC2695](#)]. 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 [[RFC2965](#)]. Accordingly, some users choose not to accept or transmit cookies.

Note that [[RFC2964](#)] specifically recommends against the usage of cookies for carrying authentication and authorization information. Nevertheless, this practice is nearly universal on the Web.

[8.1.1.4.](#) 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.

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

- HTTP over TLS (HTTPS) [[RFC2818](#)]
- SMTP over TLS [[RFC3207](#)]
- XMPP over TLS [[RFC3290](#)]
- SIP over TLS [[RFC3261](#)]
- IPsec (under some conditions)
- SSH (under some conditions)

[9.](#) 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 direct prior arrangement together, but who have prior arrangement with some third mutually trusted party, and some local configuration by which they will be able to accept each other's credentials..

[9.1.](#) 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.

[9.2.](#) 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. [\[BFL96\]](#) describes such a system.

Note that each party does need to do some fairly complex configuration and bootstrapping in order to contact a previously unknown party in this way. This work includes: locating a trusted third party, securely downloading and installing that third party's certificate and public key (which has complex and nested security challenges of its own), determining the protocols and configuration to be used for certificate request, retrieval, and revocation checking and life-cycle maintenance, generating a public-private key pair, crafting a certificate request, sending the certificate

request, retrieving and installing the granted certificate, and local configuration about which connections should employ the certificate.

[9.3.](#) Key Storage

The primary security problem with public key authentication protocol (assuming the basic protocol is designed correctly) is protecting the private keys of the certificate issuer first and foremost, and secondarily of the clients. In server applications and many non-mobile client applications, the key is simply stored on disk, often encrypted under a password-derived symmetric key. In applications where the user must carry his authentication information around, this can be done in essentially two ways: with a token or by generating the key from a password.

[9.4.](#) 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 portable media interface, like USB drive, compact flash, SD, PCexpress, smartcard, etc. 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.

There are two major types of tokens: those which are pure memory for key storage and those which do the cryptography on the token. The first are substantially cheaper but less secure because they give the key to the host computer.

[9.5.](#) 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 pseudorandom number generator (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/TLS) 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.

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

SMTP can be combined with TLS as described in [\[RFC3207\]](#). This provides similar protection to that provided when using IPsec. Since TLS certificates typically contain the server's host name, recipient authentication may be slightly more obvious, but is still susceptible to DNS spoofing attacks. Protection is provided against replay attacks, since the data itself is protected and the packets cannot be replayed.

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

- SSL/TLS (client auth mode) [\[RFC5246\]](#)
- IPsec IKE [\[RFC4306\]](#)
- S/MIME [\[RFC3850\]](#)

[10.](#) Generic Issues

[10.1.](#) Channel Security Protocols

Building a full security system into each application protocol is extremely expensive in terms of design and implementation effort. One common approach is to design a generic channel security protocol which provides a generic secure channel abstraction between a pair of endpoints. The endpoints of the channel can be authenticated at setup time and then all data flowing between them is automatically secured, allowing the application to be mostly agnostic about the security properties. SSL/TLS, SSH, and IPsec all provide this sort

of functionality.

TLS [[RFC5246](#)] provides a good example of the basic pattern, as shown below.

```
Client                                     Server
-----                                     -----
<----- TLS Handshake ----->

Application message (protected by TLS) ----->
<----- Application message (protected by TLS)
...
```

At the beginning of the TLS session, the client initiates a TCP connection to the server (TLS only works over TCP, but DTLS [[RFC4347](#)] serves a similar function for UDP), but instead of sending application data, the client and the server perform a TLS handshake, which can authenticate the server and/or the client, and which establishes cryptographic keys which are then used to protect all

future traffic. This cryptographically binds any application layer traffic to the authentication performed in the handshake.

A channel security protocol is not itself an authentication technology. Rather, it's built on top of an authentication technology, or on top of multiple technologies. Most such protocols support multiple types of authentication. For instance, TLS can be used with X.509 certificates, OpenPGP certificates [[RFC5081](#)], shared keys [[RFC4785](#)], and passwords [[RFC5054](#)].

[10.1.1.](#) Limited Authentication Options

Because a secure channel protocol needs to be able to establish cryptographic keys, the authentication options are necessarily somewhat limited. In particular, mechanisms such as passwords in the clear (both in the reusable and one-time varieties) may not be available. (See [Section 6.1](#) for one approach to work around this limitation.)

[10.1.2.](#) Limited Application Integration

Because the secure channel protocol sits beneath the application

layer protocol rather than being integrated with it, the level of integration between the two protocols is fairly loose. This is an advantage in that the application security protocol need not change at all in order to use a channel security protocol. All that is needed is for the implementation to arrange for the channel security protocol to run underneath.

The disadvantage is that the application protocol tends to have limited visibility into what the channel security protocol is doing. IPsec provides an extreme example of this: because much of the stack typically lives in the kernel, the application cannot even portably specify security properties or determine which properties apply to a given class of traffic association (there are APIs for this such as PF_KEY [[RFC2367](#)] but they are not universally deployed). Even with more tightly coupled protocols such as SSH or TLS, the applications are typically limited to setting general policy and interrogating the state of the association. They cannot, for instance, control the protection properties of individual PDUs.

[10.1.3.](#) List of Channel Security Protocols

IPsec [[RFC4301](#)]
SSH [[RFC4251](#)]
SSL/TLS [[RFC5246](#)]

DTLS [[RFC4347](#)]

[10.2.](#) Authentication Frameworks

Another popular approach is to use a pluggable authentication framework. The general idea behind a pluggable application framework is that you would like the application protocol actively involved in the authentication (unlike with a channel security protocol) but that you want to avoid specifying all of the details. Typically, the protocol framework doesn't provide any authentication features per se but instead allows you to negotiate the authentication mechanisms you wish to use. SASL [[RFC4422](#)], for instance, allows the negotiation of plaintext passwords, CRAM-MD5 (a digest-based challenge/response mechanism), and TLS among other mechanisms. Another example is in IKEv2 where the EAP framework is used to allow various forms of user

authentication, e.g. EAP-MD5, OTP (like SecurID) or Generic Token Card. GSS-API is another example of an authentication framework.

Authentication frameworks are appealing to security mechanism developers since they enable mechanisms to be supported by multiple protocols by writing a single specification. In general, it is easier to provide support for a mechanism with a framework than to integrate a security mechanism within each protocol which might use it.

Generic authentication mechanisms are attractive to application protocol designers because when properly used, they allow protocol designers to treat mechanism-specific details in an abstract manner.

While frameworks still require protocol designers to determine the threats and required security services (e.g. need for authentication/integrity/confidentiality/replay protection, protection against active attacks, etc.) as well as naming of the conversation endpoints, details of individual mechanisms can be abstracted. For example, it is not necessary for a protocol designer to concern themselves about how to locate a Kerberos KDC, or what information the latest revision of example.com's proprietary authentication token requires; these issues are handled by the framework.

While frameworks inherently provide abstraction benefits for protocol designers, the detail hiding is generally imperfect, especially from the perspective of implementers. For instance, if the framework provides mechanisms with a wide variety of security levels, designers and implementors need to be conscious of what security is provided with each level. This is often difficult to get right.

Some applications such as DNSSEC focus on providing a service to the Internet at large, that is inter-domain services. For these

applications, where interoperability of authentication between parties who have no prior association is critical, having an authentication mechanism that is "mandatory-to-implement," as opposed to a pluggable authentication framework, is likely to be the right approach. Other examples include BGP's authentication mechanism TCP-AO, and DKIM. In these types of situations, an authentication framework is likely to add significant complexity. If there is not a compelling reason to use an authentication framework in such

Internet-wide, inter-domain protocols, then it should not be used.

For applications that are often used in intra-domain contexts, i.e. within a single organization, and where end-users are authenticating within the application, frameworks may be more appropriate. This is especially true when authentication is used in a context where parties have prior associations that they use to establish credentials. With respect to intra-domain authentication, we have seen considerable diversity in the credential types that are used. Some organizations adopt PKI and smartcards, some use token cards, others use passwords or OTP systems. Authentication frameworks enable that diversity to be supported within a single architecture. For example, SSH is typically used for a party who has established a credential to access some service. Similarly, applications like IMAP, XMPP, and LDAP are used within a context of a prior relationship. It is desirable that products from one vendor interoperate with products from another vendor. However, it is more important within an intra-domain deployment that products (like an SSH system) accessing related resources (like an LDAP server) be able to use the same authentication mechanisms. That is, the example.com administrators are more concerned that whatever SSH implementation they choose can support an authentication mechanism that is also supported in the IMAP implementation they choose, than they are having all SSH implementations share an authentication mechanism. Authentication frameworks aim to allow protocol implementers to develop applications that support this deployment goal.

For protocols in the class (intra-domain), a framework may be used if it is available. If a framework is chosen, each protocol must define a mandatory-to-implement authentication mechanism. However, the framework will permit vendors to implement multiple authentication mechanisms so that those deploying implementations may choose the same mechanism across protocols. In such cases, designers should use an existing framework like EAP, SASL, or GSS-API as opposed to attempting to create something from scratch. These frameworks have taken much (re-)work to get to their current states, with more work ongoing. Attempting to replicate these efforts from scratch is not recommended, and strongly discouraged.

However, interoperability difficulty has emerged where many disparate

authentication mechanisms all use the same credentials. Therefore,

consideration must be given to limiting the number of specified mechanisms for any one class in an authentication framework, and, again, at least one "mandatory-to-implement" mechanism must be specified. See more on this in section [Section 10.2.2](#).

[10.2.1](#). Downgrade Attacks

One of 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. This issue is generally worse with frameworks which do not provide channel security because the weakest provided mechanism is often quite weak. 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, as is done in the handshakes of both IKE and TLS. However, this is not possible if a simple password mechanism is supported (as is the case in many frameworks), and policy enables it to be negotiated, because the attacker can simply capture the password in flight.

Note that if the client can establish an authenticated, integrity protected channel to the server (as is done in SSH), then the client authentication mechanism can be negotiated without fear of downgrade. Some protection against downgrade attacks can also be provided by having an endpoint cache the other endpoint's offers and complain if less secure mechanisms than were previously offered suddenly becomes available. This approach obviously bears the risk of false positives under simple misconfiguration.

Finally, downgrade prevention can be achieved by users of generic security profiling the mechanisms they offer to ensure that they are all adequately strong--at least strong enough to provide downgrade detection.

[10.2.2](#). Multiple Equivalent Mechanisms

The ease of adding new security mechanisms to generic authentication layers enables the development of multiple mechanisms with similar characteristics or even multiple mechanisms supporting the same authentication technology. This diversity has the potential to introduce interoperability problems as well as additional complexity.

Trouble arises when we have many disparate authentication mechanisms

using the same credentials. One particularly egregious example is token card support in EAP, where we have a variety of EAP mechanisms supporting RSA SecurID:

- a. Use of EAP Generic Token Card (GTC) defined in [\[RFC3748\]](#), along with a tunneling mechanism such as PEAPv0 [\[MS-PEAP\]](#), PEAPv1, EAP-TTLSv0 [\[RFC5281\]](#) or [\[RFC5216\]](#).
- b. Use of EAP-RSA along with a tunneling mechanism such as PEAPv0.
- c. Use of EAP Protected One Time Password (POTP) [\[RFC4793\]](#).

Given this level of diversity, it is common today for popular EAP peer and server implementations from different vendors to be unable to negotiate a common EAP method for SecurID support. In practice, the fact that none of these mechanisms are designated as "mandatory-to-implement" has made it very difficult for customers to put together multi-vendor deployments with any hope of interoperability -- yet the non-interoperable vendors can each claim that they implement "standards" by supporting an IETF RFC or Internet-Draft.

Another example occurs with pre-shared key mechanisms. [\[RFC3748\]](#) defined EAP-MD5; since this mechanism did not support key generation it did not satisfy the security requirements outlined in [\[RFC4017\]](#) for use on wireless networks. In order to address this weakness, additional mechanisms supporting key generation have subsequently been defined and published as Informational RFCs, including EAP-SIM [\[RFC4186\]](#), EAP-AKA [\[RFC4187\]](#), EAP-PSK [\[RFC4764\]](#), EAP-PAX [\[RFC4746\]](#), EAP-SAKE [\[RFC4763\]](#). To address the lack of a standardized mechanism, the IETF EMU WG has produced a standards-track pre-shared key method known as EAP-GPSK [\[RFC5433\]](#).

Often the proliferation of mechanisms is driven by the need to support widely deployed authentication technologies, particularly those embodied in hardware which enable "what you have" authentication. Aside from manufacturing and distribution costs, deployment of these mechanisms may involve training or backend integration costs which can only be recouped after a considerable period of use.

However, when a limited set of standardized mechanisms is defined, specification for protocol authors and deployment for network operators becomes far more successful. Whenever feasible, limiting a set of standardized mechanisms is recommended, and should be encouraged. At the very least, specifying a "mandatory-to-implement" is a must.

For example, today EAP authentication within RADIUS [[RFC3579](#)] is now

widely supported, and implementations offering mechanisms satisfying the security requirements outlined in [[RFC4017](#)] are common in such implementations as FreeRADIUS. As a result, greenfield client or server deployments rarely have a need for use of EAP-MD5, and the development of standardized pre-shared key mechanism may eventually enable replacement of EAP-MD5 as the mandatory-to-implement EAP authentication mechanism.

The real point here it to ensure that multiple authentication mechanisms aren't trying to authenticate the same credential type, rather than to arbitrarily limit the number of authentication mechanisms. For example, while neither Kerberos nor (D)TLS are authentication frameworks, each now supports multiple credential types. This is both powerful and desirable for customers, since it means that they don't have to support each credential type with each application individually.

However, having multiple different ways to authenticate with the same credential type, enabling vendors to claim compliance without interoperating, would be likely to result in customer frustration. This can be avoided by standardizing a "mandatory-to-implement" mechanism for each credential type, ensuring interoperability out-of-the-box."

In order to encourage interoperability and the reduction of complexity, it is recommended that the IETF standardize only a small number of authentication mechanisms within a pluggable authentication framework. Proliferation of mechanisms should be limited to no more than one for any given class within the framework. Recalling [section 10.2](#), having a small number of mechanisms and clearly stating this minimal set in the protocol specification is particularly important when all implementations on the Internet will need to use the same mechanisms for authentication in order to interoperate. Examples of such Internet wide protocols have included DKIM, DNSSEC, and infrastructure protocols like BGP. Support for a standardized password-based (includes pin + OTP) mechanism is highly recommended for protocols where end-users (as opposed to unattended machines) will be involved.

When working to limit the number of mechanisms, designers should take care not to break the architecture of an existing framework. For example, for SASL, it goes against the architecture to have mechanism-specific information such as specific mechanism restrictions in a protocol. Care must also be taken that such restrictions do not lead to mechanism-specific details making their way directly into protocols. Such layering violations make it harder to revise mechanisms in the future or to change the set of appropriate mechanisms if proven necessary over time. Experience has

demonstrated that we are likely to need to change the set of mechanisms over time, as new technologies or new requirements emerge.

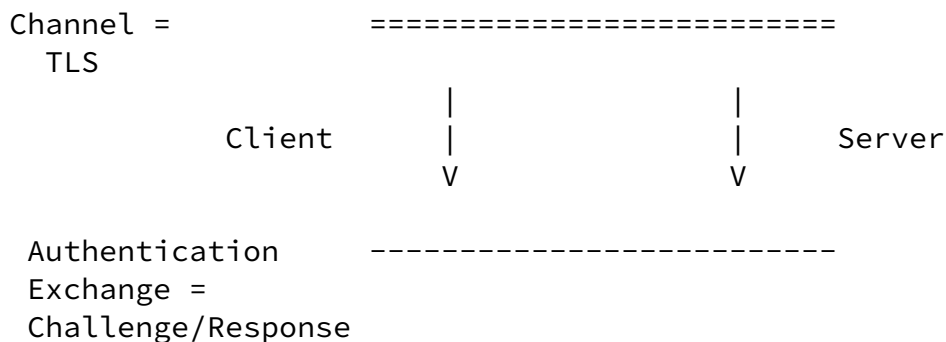
[10.2.3.](#) Channel Bindings

Many applications desire channel security but tighter integration of authentication with the application than is typically provided by channel security protocols. A common approach is to run the application protocol on top of a channel security protocol (most commonly TLS) but to use an authentication framework (most commonly SASL) for client (and sometimes server) authentication. As described in [Section 6](#) this is potentially subject to man in the middle attack. As described in [Section 6.1](#) and [Section 8](#) if the server can be authenticated by the channel security protocol, then a MITM attack is not possible.

If the server cannot be so authenticated, then the authentication performed by the framework must be cryptographically bound to the cryptographic context formed by the channel security protocol (this is often called a CHANNEL BINDING) so that the authentication framework will fail if a MITM attack is underway.

For example, consider the figure below. If TLS is used without client or server authentication to provide only privacy (via encryption) and per-packet authentication and integrity (meaning I know that the packet came from the same person with whom I started this connection, and that it hasn't been changed along the way), it is still susceptible to a MitM attack. A challenge-response mechanism may be used inside the TLS to add end-point authentication. If so, one must include something from the TLS exchange in the challenge-response exchange in order to actually protect against a MitM attack. I.e. the channel and the authentication exchange must be bound

together.



Channel Binding Example

For instance, if the authentication framework is using a challenge-

response mechanism, the response computation could include a characteristic value from channel security protocol, thus forcing the response given to the attacker and the response expected by the server to be different. Note that care must be used in selecting the characteristic value to ensure that the attacker cannot force the values to be the same for both connections.

[[I-D.altman-tls-channel-bindings](#)],
[[I-D.williams-ipsec-channel-binding](#)] and
[[I-D.williams-sshv2-channel-bindings](#)] describe selection of values for TLS, IPsec, and SSH.

Note that this technique cannot be used with non-cryptographic mechanisms such as simple passwords or one-time passwords. If these mechanisms are to be used in environments where MITM attacks are a concern, then the server must be authenticated by the channel security protocol.

[10.2.4.](#) Excessive Layering

Many of the legacy authentication mechanisms that users and administrators wish to support are themselves generic frameworks of one kind or another. In general, when two security frameworks are run together with one as a mode of the other, it becomes very difficult to make assertions about the security properties of the composed system. Among the issues are:

- o The state machines can become interlinked, causing confusion at one layer about the state of the other layer. For instance, TLS has a simple two round trip exchange, but [[I-D.nir-tls-eap](#)] extends that with a generic "EAPMsg" that may occur an arbitrary number of times without transitions in the TLS state machine.
- o Understanding the composed system becomes difficult. Experts in one security protocol often are not experts in all, and unless the encapsulation boundaries are very carefully drawn, analyzing the composed protocol may require an unavailable level of general expertise.
- o Any proofs of security that may be available for one of the systems almost certainly depend on knowledge of the available cryptographic mechanisms, but if one of those mechanisms is a framework, then those proofs no longer apply.

These issues have been encountered within the Extensible Authentication Protocol (EAP), defined in [[RFC3748](#)] Where EAP runs over link layers that support authentication mechanisms other than EAP (such as PPP or IEEE 802.16), it may be necessary to first negotiate use of EAP, and then within EAP, to negotiate the specific EAP mechanism to be used. This may introduce security vulnerabilities. For example, since neither PPP authentication

negotiation nor EAP mechanism are secured, it is necessary for both PPP and EAP authentication policy to be pre-provisioned on the EAP peer and server in order to prevent bidding down attacks.

Since EAP supports a wide range of security mechanisms, support for multiple link layer authentication mechanisms is generally unnecessary, and in general greenfield link layer designs supporting EAP are best advised to forgo other approaches.

The issue of multiple negotiation layers is also encountered within EAP methods. While some EAP methods (such as EAP-TLS [[RFC5216](#)] only support a single authentication mechanism, other such as EAP-FAST [[RFC4851](#)] and [[I-D.funk-eap-ttls-v0](#)] act as "tunneling methods", providing for negotiation of an "inner EAP method". As noted in [[RFC3748](#)] [Section 7.4](#), unless the inner and outer authentication mechanisms are cryptographically bound, tunneling methods are vulnerable to a man-in-the-middle attack.

In accordance with the principle of having as few mechanisms as possible, applications should try to avoid having multiple negotiation layers. If that is not possible, applications should profile a single negotiation layer. If application Foo is to be used with framework Bar which supports authentication methods Alpha and Bravo, itself supports framework Baz, which supports authentication methods Alpha, Bravo, and Charlie, Foo should indicate whether Alpha and Bravo are to be supported via Bar or Baz.

[10.2.5.](#) List of Authentication Frameworks

GSS-API [[RFC2743](#)]

SASL [[RFC4422](#)]

EAP [[RFC3748](#)]

[11.](#) 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.

[11.1.](#) 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.

[11.2.](#) Single Sign-On

A related problem is that it's undesirable to have users manually authenticate each time some service wants authentication. First, it's inconvenient for the users. Second the cognitive load associated with frequent authentication seems likely to lead to

careless use of credentials, enabling attacks such as phishing. 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). 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, the protocols are straightforward since all relying parties can have the same verifier. 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.

11.3. Case Study: RADIUS

RADIUS, defined in [[RFC2865](#)], is a protocol for Authentication, Authorization and Accounting (AAA), commonly implemented in Network Access Servers (NASes). NAS devices are often constrained in terms of their CPU power, memory, or non-volatile storage. As a result, it may be difficult for them to implement a variety of authentication mechanisms. Also, given that access networks may contain hundreds or even thousands of NAS devices, management concerns may lead to implementation of a centralized authentication scheme. As a result, NAS devices may not perform authentication directly, instead delegating this to one or more authentication servers.

When utilizing AAA servers for authentication, NAS devices act as "pass-through" devices, forwarding authentication exchanges between the user and the AAA server. Such an arrangement implicitly assumes the AAA server acts as a trusted third party, and that communication between the NAS and AAA server is authenticated and integrity and replay protected.

As described in "The Network Access Identifier" [[RFC4282](#)] and "Chargeable User Identity" [[RFC4372](#)], there are circumstances in which the user desires to keep its identity confidential both to a potential attacker that may be snooping on the conversation between

the user and the NAS, as well as to the NAS itself. In these circumstances, only the AAA server may authenticate the identity of the user, and the NAS may only be provided with a "temporary identity" sufficient for authorization and billing purposes. "

[11.4.](#) Case Study: Kerberos

Kerberos [[RFC4210](#)] is a popular authentication/single sign-on service. Kerberos is based on the Needham-Schroeder authentication protocol. The authentication server role 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.

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

Kerberos [[RFC4120](#)]
RADIUS [[RFC2865](#)]
Diameter [[RFC3588](#)]

[12.](#) 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.

[12.1.](#) 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:

12.1.1. What's my threat model?

Sorting out the threat model (see [[RFC3552](#)]) is always the first step in deciding what sorts of security mechanisms to use. In the case of authentication you must consider, at minimum:

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?

12.1.2. 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.

The complexity compared to scale of users will also be affected depending on whether all the users are "managed" or are "unmanaged". Managed users are those whose machines are owned, operated and managed by the organization deploying the system. Unmanaged users are those whose machines are not own, operated or managed by the deploying organization. System administrators tend to have far more control over managed machines, e.g. a company issued laptop or desktop machine, as compared to unmanaged machines, e.g. someone's home computer or personal smart phone. The complexity may further decrease for managed machines when a central management system is available for making configuration changes quickly and easily to all devices in the system. A central management system is unlikely for a solution with unmanaged machines.

12.1.3. 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 13.2.2](#) for more discussion on this topic.

[12.1.4.](#) 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.

[12.2.](#) Use as few mechanisms as you can

In the best case, each system would only have one or a small number of forms of authentication. The more methods of authentication a system allows, the more things there are to go wrong, and the more difficult it is to gain solid interoperability. Unfortunately, this is not always possible. 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). Even for the same user, some of the user's machines could accept a token, e.g. a laptop, while others may not, e.g. a smart phone.

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 mechanisms that use the same underlying keying material in different ways. 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 using the same passwords/keys presents serious complexity challenges and should be avoided if possible. Again, this is not always possible in systems with legacy authentication mechanisms but should be avoided in new designs.

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 there's a good a priori case for diversity in authentication mechanisms. Where generic security frameworks are used, designers need to carefully analyze the threats relevant to each mechanism in the context of the specific application layer

protocol environment. In order to minimize the attack surface, individual deployments would be wise to specify policies which disallow mechanisms which are unnecessary in their environment, even if they are specified in the protocol. For instance, if users are expected to use challenge response, then optimally clients would be configured not even to perform plaintext passwords, as this represents a security threat no matter what the server configuration is.

[12.3.](#) 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. The difficulty is that simple passwords almost never provide the user with any form of server authentication. 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.

Even when an authenticated and encrypted channel to the server is available, the use of cleartext passwords places strong requirements on the protection provided by encryption, in part because the same plaintext is transmitted repeatedly. [[RFC3579](#)] and [[CHVV03](#)] describe examples of such situations.

Accordingly, designers should avoid deploying simple password mechanisms if at all possible, not just provide stronger mechanisms.

[12.4.](#) Avoid inventing new frameworks

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 authentication frameworks or channel security systems. If you want an authentication framework, use SASL or GSS-API or (if you're in a network access context) EAP. If you want a channel security system, use IPsec, TLS, or DTLS. Note that none of these systems can be blindly dropped into an existing system and provide adequate

security. Care must be taken to analyze the protocol being secured and determine the correct interaction model. [[RFC5406](#)] provides guidance on this topic for IPsec.

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

Having the strongest security you can propose 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 SecurID 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. NIST Spec. Pub. 800-63 [[SP800-63](#)] provides good guidance about the

Rescorla, et al.

Expires August 31, 2010

[Page 47]

Internet-Draft

Authentication Mechanisms

February 2010

minimum requirements for various applications.

[12.6.](#) 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.

Message integrity provides a cryptographic indication in each message that, when validated, confirms that the message has not been altered since being sent from a trusted host. When applied correctly, it is often possible to use one cryptographic operation to achieve both the message integrity and the message authentication. For example, doing a digest authentication operation over a suitably large portion of the message (or payload, or packet, depending on the protocol in question) using a connection key known by both parties provides both message integrity and message authentication. IPsec, TLS, DTLS and SSH are all examples of protocols that provide message integrity and message authentication in one operation.

[13.](#) 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 capabilities. In the second part we discuss a number of different protocol architectures and the potential mechanisms which can be used with those architectures.

[13.1.](#) Capability Considerations

There are three primary authentication scenarios: (1) Neither side has a public/private key pair. (2) One side 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.

[13.1.1.](#) 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 however, in at least two cases (SACRED and IPS), IETF WGs have chosen not to require ZKPPs due to IPR concerns.

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 6](#). With this adaptation, an attacker needs to mount an active attack in order to dictionary search the password space.

[13.1.2.](#) One side has an authenticated key pair

If the server has a key pair which the client can authenticate, then several alternatives are available for password authentication.

Simple username/password encrypted under the server's public key is the preferred authentication mechanism. Rather than encrypting directly under the server's public key, the standard practice here is to use the server's key to establish a secure channel and then pass the password over that channel. Challenge/response is in fact weaker in this case because it is password equivalent.

In this situation, using a single OTP only to authenticate the client to the server during a first connection, as a bootstrap, is quite useful. Once authenticated, and a secure channel is established, the client can pass some long term credential to the server, like its self-generated key pair's public key. Then after, the public key is used for the client authentication proof. This is how some mobile devices "register" and bootstrap secure connections to servers (e.g. how the handset registers to it's home mail server).

Assuming that an authenticated server key pair is available, the use of a continuous list of OTPs and ZKPP systems offer significant additional management complexity for marginal security benefit.

However, the difficulties involved in establishment of an authenticated server key pair may be substantial. These issues include:

- a. Provisioning of trust anchors. In a number of scenarios, such as

establishment of network access from an unprovisioned host, trust anchors may not be pre-populated or utilization of pre-populated trust anchors may introduce security vulnerabilities. In such circumstances, either trust anchors need to be verified after the fact, leaving the client vulnerable to active attack, or out-of-band provisioning mechanisms need to be provided.

- b. Certificate validation. In practice, the deployment of Certificate Revocation Lists (CRLs) or Online Certificate Status Protocol (OCSP) may present practical difficulties.

- c. Man-in-the-Middle attacks. In order to avoid trust anchor

provisioning or certificate validation, "leap of faith" approaches such as that used within SSH may be appealing. However, such an approach assumes that an attacker cannot gain sufficient access to disrupt the initial authentication attempt which establishes trust in the server public key pair. In some scenarios (e.g. client authenticating to a server in a restricted environment), this assumption may be valid; in other scenarios (wireless network authentication), it may not be.

[13.1.3.](#) Both sides have authenticated key pairs

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

[13.2.](#) Architectural Considerations

In this section, we consider 3 different network architectures and the authentication mechanisms that are most suitable for each.

[13.2.1.](#) Simple Connection

The simplest authentication scenario is where the peers are connected by some interactive connection. Mercifully, this situation is quite common in such protocols as IMAP, TELNET, etc. In this simple 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.

[13.2.2.](#) 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 either client nor server may not trust the proxy and they do not want to involve it in their authentication. 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.

[13.2.2.1.](#) 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 [[RFC2817](#)], [[RFC2818](#)]. Since the proxy is untrusted, the application protocol must either be run over a secure channel or hardened against active attacks.

[13.2.2.2.](#) 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.

[13.2.2.3.](#) 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.

[13.2.3.](#) 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 tunneling 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.

[13.2.4.](#) 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 GSAKMP [[RFC4535](#)].

In the second case, individual group members are anticipated to possibly forge messages. With 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.

[14.](#) Acknowledgements

Eric Rescorla was the original author and editor of this document, for versions -00 through -05. Gregory Lebovitz assumed the editor role starting in draft -06. Early versions of this document were reviewed by Fred Baker, Lisa Dusseault, Ted Hardie, and Mike St. Johns. Thanks to Jeffrey Altman, Sam Hartman, Paul Hoffman, John Linn, and Nico Williams for their reviews and comments. Bernard Aboba contributed extensive sections of this document, performed several reviews, and kept the meticulous issue tracker, which received plenty of contributions.

Though published in 2010, the vast majority of the work done on this document occurred under the IAB teams of 2003-2006. Your efforts are remembered and appreciated.

[15.](#) References

[15.1.](#) Normative References

[15.2.](#) Informative References

[SP800-63]

"National Institute of Standards and Technology, "Electronic Authentication Guideline: Recommendations of the National Institute of Standards and Technology", SP 800-63", 2004.

[AKA]

"Technical Specification Group Services and System Aspects; 3G Security; Security Architecture (Release 5) 3GPP TS 33.102 V5.1.0."

[BFL96]

Blaze, M., Feigenbaum, J., and J. Lacy, "'Decentralized trust management", IEEE Symposium on Security and Privacy '96."

[Bell89]

Bellovin, S., "'Security Problems in the TCP/IP Protocol Suite", Computer Communications Review", 1989.

[Bell92]

Bellovin, S. and M. Merritt, "'Encrypted Key Exchange: Password-based protocols secure against dictionary

attacks", Proceedings of the IEEE Symposium on Research in Security and Privacy '92.", 1992.

- [Bell94] Bellovin, S. and M. Merritt, "'Augmented Encrypted Key Exchange: a Password-Based Protocol Secure Against Dictionary Attacks and Password File Compromise". AT&T

Rescorla, et al.

Expires August 31, 2010

[Page 53]

Internet-Draft

Authentication Mechanisms

February 2010

Bell Laboratories Technical Report", 1994.

- [CHVV03] Canvel, B., Hiltgen, A., Vaudenay, S., and M. Vuagnoux, "'Password Interception in a SSL/TLS Channel", Advances in Cryptology CRYPTO 2003.", 2003.
- [DTH06] Dhamija, R., Tygar, J., and M. Hearst, "'Why Phishing Works", CHI 2006", 2006.
- [Crack] Muffet, A., "CRACK v 5.0a".
- [Wil68] Wilkes, M., "'Time-Sharing Computer Systems", American Elsevier New York."
- [Worm88] Spafford, E., "'The Internet Worm Program: An Analysis",".
- [Jab96] Jablon, D., "'Strong Password-Only Authenticated Key Exchange", Computer Communication Review", 1996.
- [KP01] Kaufman, C. and R. Perlman, "'PDM: A New Strong Password-Based Protocol", Proceedings of the 10th USENIX Security Symposium '01", 2001.
- [Klein90] Klein, D., "'Foiling the Cracker: A Survey of Improvements to Password Security'", 1990.
- [I-D.altman-tls-channel-bindings]
Altman, J., Williams, N., and L. Zhu, "Channel Bindings for TLS", [draft-altman-tls-channel-bindings-07](#) (work in progress), October 2009.
- [I-D.funk-eap-ttls-v0]
Funk, P. and S. Blake-Wilson, "EAP Tunneled TLS Authentication Protocol Version 0 (EAP-TTLSv0)", [draft-funk-eap-ttls-v0-05](#) (work in progress), April 2008.

[I-D.newman-auth-scam]
Menon-Sen, A., Melnikov, A., Newman, C., and N. Williams,
"Salted Challenge Response (SCRAM) SASL Mechanism",
[draft-newman-auth-scam-13](#) (work in progress), May 2009.

[MS-PEAP] Microsoft Corporation, "MS-PEAP: Protected Extensible
Authentication Protocol (PEAP) Specification",
January 2010.

[I-D.nir-tls-eap]
Nir, Y., Sheffer, Y., Tschofenig, H., and P. Gutmann, "TLS
using EAP Authentication", [draft-nir-tls-eap-06](#) (work in

Rescorla, et al.

Expires August 31, 2010

[Page 54]

Internet-Draft

Authentication Mechanisms

February 2010

progress), April 2009.

[I-D.williams-ipsec-channel-binding]
Williams, N., "End-Point Channel Bindings for IPsec Using
IKEv2 and Public Keys",
[draft-williams-ipsec-channel-binding-01](#) (work in
progress), April 2008.

[I-D.williams-sshv2-channel-bindings]
Williams, N., "Channel Binding Identifiers for Secure
Shell Channel", [draft-williams-sshv2-channel-bindings-00](#)
(work in progress), November 2007.

[RFC0854] Postel, J. and J. Reynolds, "Telnet Protocol
Specification", STD 8, [RFC 854](#), May 1983.

[RFC0959] Postel, J. and J. Reynolds, "File Transfer Protocol",
STD 9, [RFC 959](#), October 1985.

[RFC1760] Haller, N., "The S/KEY One-Time Password System",
[RFC 1760](#), February 1995.

[RFC1939] Myers, J. and M. Rose, "Post Office Protocol - Version 3",
STD 53, [RFC 1939](#), May 1996.

[RFC2104] Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-
Hashing for Message Authentication", [RFC 2104](#),
February 1997.

- [RFC2195] Klensin, J., Catoe, R., and P. Krumviede, "IMAP/POP AUTHorize Extension for Simple Challenge/Response", [RFC 2195](#), September 1997.
- [RFC2289] Haller, N., Metz, C., Nesser, P., and M. Straw, "A One-Time Password System", [RFC 2289](#), February 1998.
- [RFC2367] McDonald, D., Metz, C., and B. Phan, "PF_KEY Key Management API, Version 2", [RFC 2367](#), July 1998.
- [RFC2617] Franks, J., Hallam-Baker, P., Hostetler, J., Lawrence, S., Leach, P., Luotonen, A., and L. Stewart, "HTTP Authentication: Basic and Digest Access Authentication", [RFC 2617](#), June 1999.
- [RFC2660] Rescorla, E. and A. Schiffman, "The Secure HyperText Transfer Protocol", [RFC 2660](#), August 1999.
- [RFC2695] Chiu, A., "Authentication Mechanisms for ONC RPC",

Rescorla, et al.

Expires August 31, 2010

[Page 55]

Internet-Draft

Authentication Mechanisms

February 2010

[RFC 2695](#), September 1999.

- [RFC2743] Linn, J., "Generic Security Service Application Program Interface Version 2, Update 1", [RFC 2743](#), January 2000.
- [RFC2808] Nystrom, M., "The SecurID(r) SASL Mechanism", [RFC 2808](#), April 2000.
- [RFC2817] Khare, R. and S. Lawrence, "Upgrading to TLS Within HTTP/1.1", [RFC 2817](#), May 2000.
- [RFC2818] Rescorla, E., "HTTP Over TLS", [RFC 2818](#), May 2000.
- [RFC2865] Rigney, C., Willens, S., Rubens, A., and W. Simpson, "Remote Authentication Dial In User Service (RADIUS)", [RFC 2865](#), June 2000.
- [RFC2945] Wu, T., "The SRP Authentication and Key Exchange System", [RFC 2945](#), September 2000.
- [RFC2964] Moore, K. and N. Freed, "Use of HTTP State Management",

[BCP 44](#), [RFC 2964](#), October 2000.

- [RFC2965] Kristol, D. and L. Montulli, "HTTP State Management Mechanism", [RFC 2965](#), October 2000.
- [RFC3207] Hoffman, P., "SMTP Service Extension for Secure SMTP over Transport Layer Security", [RFC 3207](#), February 2002.
- [RFC3261] Rosenberg, J., Schulzrinne, H., Camarillo, G., Johnston, A., Peterson, J., Sparks, R., Handley, M., and E. Schooler, "SIP: Session Initiation Protocol", [RFC 3261](#), June 2002.
- [RFC3290] Bernet, Y., Blake, S., Grossman, D., and A. Smith, "An Informal Management Model for Diffserv Routers", [RFC 3290](#), May 2002.
- [RFC3501] Crispin, M., "INTERNET MESSAGE ACCESS PROTOCOL - VERSION 4rev1", [RFC 3501](#), March 2003.
- [RFC3552] Rescorla, E. and B. Korver, "Guidelines for Writing RFC Text on Security Considerations", [BCP 72](#), [RFC 3552](#), July 2003.
- [RFC3579] Aboba, B. and P. Calhoun, "RADIUS (Remote Authentication Dial In User Service) Support For Extensible Authentication Protocol (EAP)", [RFC 3579](#), September 2003.

Rescorla, et al.

Expires August 31, 2010

[Page 56]

Internet-Draft

Authentication Mechanisms

February 2010

- [RFC3588] Calhoun, P., Loughney, J., Guttman, E., Zorn, G., and J. Arkko, "Diameter Base Protocol", [RFC 3588](#), September 2003.
- [RFC3748] Aboba, B., Blunk, L., Vollbrecht, J., Carlson, J., and H. Levkowetz, "Extensible Authentication Protocol (EAP)", [RFC 3748](#), June 2004.
- [RFC3850] Ramsdell, B., "Secure/Multipurpose Internet Mail Extensions (S/MIME) Version 3.1 Certificate Handling", [RFC 3850](#), July 2004.
- [RFC4017] Stanley, D., Walker, J., and B. Aboba, "Extensible Authentication Protocol (EAP) Method Requirements for Wireless LANs", [RFC 4017](#), March 2005.

- [RFC4120] Neuman, C., Yu, T., Hartman, S., and K. Raeburn, "The Kerberos Network Authentication Service (V5)", [RFC 4120](#), July 2005.
- [RFC4186] Haverinen, H. and J. Salowey, "Extensible Authentication Protocol Method for Global System for Mobile Communications (GSM) Subscriber Identity Modules (EAP-SIM)", [RFC 4186](#), January 2006.
- [RFC4187] Arkko, J. and H. Haverinen, "Extensible Authentication Protocol Method for 3rd Generation Authentication and Key Agreement (EAP-AKA)", [RFC 4187](#), January 2006.
- [RFC4210] Adams, C., Farrell, S., Kause, T., and T. Mononen, "Internet X.509 Public Key Infrastructure Certificate Management Protocol (CMP)", [RFC 4210](#), September 2005.
- [RFC4251] Ylonen, T. and C. Lonvick, "The Secure Shell (SSH) Protocol Architecture", [RFC 4251](#), January 2006.
- [RFC4279] Eronen, P. and H. Tschofenig, "Pre-Shared Key Ciphersuites for Transport Layer Security (TLS)", [RFC 4279](#), December 2005.
- [RFC4282] Aboba, B., Beadles, M., Arkko, J., and P. Eronen, "The Network Access Identifier", [RFC 4282](#), December 2005.
- [RFC4301] Kent, S. and K. Seo, "Security Architecture for the Internet Protocol", [RFC 4301](#), December 2005.
- [RFC4306] Kaufman, C., "Internet Key Exchange (IKEv2) Protocol", [RFC 4306](#), December 2005.

- [RFC4347] Rescorla, E. and N. Modadugu, "Datagram Transport Layer Security", [RFC 4347](#), April 2006.
- [RFC4372] Adrangi, F., Lior, A., Korhonen, J., and J. Loughney, "Chargeable User Identity", [RFC 4372](#), January 2006.
- [RFC4422] Melnikov, A. and K. Zeilenga, "Simple Authentication and

Security Layer (SASL)", [RFC 4422](#), June 2006.

- [RFC4535] Harney, H., Meth, U., Colegrove, A., and G. Gross, "GSAKMP: Group Secure Association Key Management Protocol", [RFC 4535](#), June 2006.
- [RFC4746] Clancy, T. and W. Arbaugh, "Extensible Authentication Protocol (EAP) Password Authenticated Exchange", [RFC 4746](#), November 2006.
- [RFC4763] Vanderveen, M. and H. Soliman, "Extensible Authentication Protocol Method for Shared-secret Authentication and Key Establishment (EAP-SAKE)", [RFC 4763](#), November 2006.
- [RFC4764] Bersani, F. and H. Tschofenig, "The EAP-PSK Protocol: A Pre-Shared Key Extensible Authentication Protocol (EAP) Method", [RFC 4764](#), January 2007.
- [RFC4785] Blumenthal, U. and P. Goel, "Pre-Shared Key (PSK) Ciphersuites with NULL Encryption for Transport Layer Security (TLS)", [RFC 4785](#), January 2007.
- [RFC4793] Nystroem, M., "The EAP Protected One-Time Password Protocol (EAP-POTP)", [RFC 4793](#), February 2007.
- [RFC4851] Cam-Winget, N., McGrew, D., Salowey, J., and H. Zhou, "The Flexible Authentication via Secure Tunneling Extensible Authentication Protocol Method (EAP-FAST)", [RFC 4851](#), May 2007.
- [RFC4949] Shirey, R., "Internet Security Glossary, Version 2", [RFC 4949](#), August 2007.
- [RFC5054] Taylor, D., Wu, T., Mavrogiannopoulos, N., and T. Perrin, "Using the Secure Remote Password (SRP) Protocol for TLS Authentication", [RFC 5054](#), November 2007.
- [RFC5081] Mavrogiannopoulos, N., "Using OpenPGP Keys for Transport Layer Security (TLS) Authentication", [RFC 5081](#), November 2007.

- [RFC4248] Hoffman, P., "The telnet URI Scheme", [RFC 4248](#), October 2005.
- [RFC4266] Hoffman, P., "The gopher URI Scheme", [RFC 4266](#), November 2005.
- [RFC5216] Simon, D., Aboba, B., and R. Hurst, "The EAP-TLS Authentication Protocol", [RFC 5216](#), March 2008.
- [RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", [RFC 5246](#), August 2008.
- [RFC5281] Funk, P. and S. Blake-Wilson, "Extensible Authentication Protocol Tunneled Transport Layer Security Authenticated Protocol Version 0 (EAP-TTLSv0)", [RFC 5281](#), August 2008.
- [RFC5322] Resnick, P., Ed., "Internet Message Format", [RFC 5322](#), October 2008.
- [RFC5386] Williams, N. and M. Richardson, "Better-Than-Nothing Security: An Unauthenticated Mode of IPsec", [RFC 5386](#), November 2008.
- [RFC5406] Bellovin, S., "Guidelines for Specifying the Use of IPsec Version 2", [BCP 146](#), [RFC 5406](#), February 2009.
- [RFC5433] Clancy, T. and H. Tschofenig, "Extensible Authentication Protocol - Generalized Pre-Shared Key (EAP-GPSK) Method", [RFC 5433](#), February 2009.

[Appendix A](#). IAB Members at the time of this writing

Marcelo Bagnulo
Gonzalo Camarillo
Stuart Cheshire
Vijay Gill
Russ Housley
John Klensin
Olaf Kolkman
Gregory Lebovitz
Andrew Malis
Danny McPherson
David Oran
Jon Peterson
Dave Thaler

Internet-Draft

Authentication Mechanisms

February 2010

Authors' Addresses

Eric Rescorla
RTFM, Inc.
2064 Edgewood Drive
Palo Alto, CA 94303
USA

Email: ekr@rtfm.com

Gregory Lebovitz
Juniper Networks, Inc.
1194 N. Mathilda Ave.
Sunnyvale, CA 94089-1206
USA

Email: gregory.ietf@gmail.com

Internet Architecture Board
IAB

Rescorla, et al.

Expires August 31, 2010

[Page 60]