NETWORK WORKING GROUP
Internet-Draft
Intended status: Informational
Expires: December 4, 2012

N. Williams
Cryptonector

June 2, 2012

A Proposals for Classification and Analysis of HTTPbis Authentication Proposals
draft-williams-httpbis-auth-classification-00

Abstract

This document proposes a classification scheme for HTTPbis authentication proposals, to help with analysis and selection.

Status of this Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

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."

This Internet-Draft will expire on December 4, 2012.

Table of Contents

1. Introduction

The HTTPbis WG is accepting proposals for new authentication systems for HTTPbis, the successor to Hypertext Transport Protocol (HTTP) version 1.1[[RFC2616]]. This document proposes a classification system for these proposals. Several axes of classification are proposed, and several simplified imagined or likely authentication systems are used to illustrate the classification system.

The author assumes that the WG is interested primarily in new user authentication proposals, with ones that provide mutual authentication (of users and servers to each other) being in scope. The author also assumes that Transport Layer Security (TLS) [[RFC5246]] will continue to be used by HTTPbis for cryptographic session protection.

Some familiarity with authentication systems is assumed. A glossary is provided.

1.1. Conventions used in this document

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [[RFC2119]].

1.2. Scope

This document considers user authentication only in the context of HTTP applications, whether they be web applications or otherwise. Authentication of the service is also in scope, but authentication methods that authenticate only the user to the service (with the service authenticated by Transport Layer Security (TLS)) are in scope.

There are at least two entities involved in authentication in this context: the user (on the client side), one or more of the web server host or the web server application/service, and any trusted third parties that an authentication mechanism might involve.

1.3. Glossary

This section defines terms as they are used in _this_ document. Readers are strongly encouraged to read this section before reading any subsequent section.

API Application Programming Interface. These are interfaces between an application and a feature that is abstracted into a "library" - a service provided by the platform's operating system.

API Layer A complex Internet application might require a large number of APIs, such as, for example, one for every network layer. In practice it is more common to have a single API that encompasses all network layers below it, with the component providing that API likely invoking other APIs itself. which in turn invoke other APIs. For example, a web application might use a library that presents a single API to all of the HTTP network stack from HTTP all the way down to IP. Note that there need not be a direct correspondence of network and API layers.

Authentication The process of establishing the veracity or origin of some statement (e.g., of an entity's identity), usually by proxy (e.g., with key-pairs to an asymmetric key cryptographic system "speaking for" the authenticated entities). In this document, and unless otherwise stated, "authentication" will refer to authentication of identity of entities such as "users", "hosts", and "services".

Authentication Mechanism A cryptographic protocol for authenticating entity identities. Note that this does not cover POSTing usernames and passwords in forms, but it does cover bearer token mechanisms (if just barely).

Authentication Method A scheme for authenticating entity identities. An authentication method can be non-cryptographic, covering HTTP Basic authentication and usernames&passwords POSTed from HTML forms.

Authentication Framework A protocol into which other authentication mechanisms may be plugged in. For example: SASL[[RFC4422]], GSS-API[[RFC2743]], EAP[[RFC3748]], among others.

Bearer Token A technique for authentication that involves a message that can be presented by the authenticating entity to another. No proof of possession is required for using bearer tokens, which means that the token can be presented by any entity possessing the token, which in turn means that bearer tokens must be sent with confidentiality protection, as otherwise eavesdroppers can steal them and use them to impersonate the subject.

Channel Binding A security protocol composition and analysis tool. The purpose of channel binding[[RFC5056]] is to "bind" a secure channel (at one layer in the network stack) into an authentication protocol running at a higher layer in the stack, thereby ensuring

that the channel is end-to-end and "speaks for" its end-points.

Confidentiality protection Cryptographic encryption of data. Confidentiality protection is/must always be used with integrity protection as well.

Data authentication Data origin authentication, a.k.a., integrity protection.

Integrity protection Cryptographic protection against modification of data. See also "data authentication", above.

Mechanism Shorthand for "authentication mechanism", a protocol defining messages to be exchanged in order to authenticate one party to another (or two parties to each other).

Mutual Authentication Authentication of a user and a server/service to each other.

Mutual Authentication (key confirmation sense) In some protocols key exchange is bound to authentication of the service to the user such that the service is finally authenticated when it sends a proof-of-possession of the exchanged session key back to the user. Protocols that use RSA key transport (e.g., TLS in common usage), Diffie-Hellman with a persistent public key for the server, or Needham-Schroeder protocols (such as Kerberos[[RFC4120]]), perform server authentication in this way. A client may not always care to receive key confirmation. For example, a Kerberos client for a lossy logging application might not care that confidentiality protected data ends up at the wrong server, as long as unintended servers can't decrypt the data. Some clients may send application data optimistically ahead of key confirmation from the server. Such data should generally be confidentiality protected, and the protocol should not be subject to MITM attacks where the MITM can somehow modify what optimistic data is sent, nor should an active attacker be able to replay such optimistic data.

Network Layer A layer in the OSI or Internet network model. Examples of layers that are relevant to HTTP applications: IP, TCP/UDP, TLS, HTTP, and the application layer.

Proof of Possession A technique for authentication that involves using a cryptographic operation to "prove" (not necessarily in a rigorous sense) that the entity that creates the proof has access to a private/secret key to a cryptosystem (e.g., a private RSA key, a secret AES key, etcetera).

Public Key Infrastructure (PKI) An authentication system based on public key cryptography and supporting hierarchical transitive trust via trusted third parties known as Certificate Authorities (CAs).

Relying Party An entity that authenticates another. For example, in PKI the entity that validates another's certificate as part of the process of authenticating that other entity, is a relying party.

SCRAM Salted Challenge Response Authentication Mechanism (SCRAM)[[RFC5802]], a SASL[[RFC4422]] and GSS mechanism based on password-derived pre-shared keys and challenge/response. SCRAM is intended as the successor to SASL's DIGEST-MD5, and possibly to HTTP's DIGEST-MD5.

Server A system with one or more IP addresses, serving HTTP on one more TCP ports on those IP addresses. [A general definition would not be constrained to HTTP only, but for the purposes of this document this is good enough.]

Service An entity providing a service or services for an application. Typically -but not always!- a service is closely related to a host server, which may provide several services. Usually we need to distinguish between the various services that a single host provides, thus we often need to authenticate the _service_ rather than the host server. For HTTP applications a service may be a collection of resources available on one (or more) ports on a given server.

Trust (in authentication) This word, "trust", is a terrible word: it means too many things to too many people. But it's also a very convenient word when everyone understands the meaning to be accorded to it in any given context. For the time being this document will use this word, "trust", as follows: to trust an entity is to accept as fact assertions -relating to other entities- made by the trusted entity. Alternative phrasing: to trust an entity is to rely on it to make assertions relating to other entities the truth of which cannot otherwise be ascertained. For example, in a PKI a relying party relies on the certification authorities (and related infrastructure) to make statements of facts of the form "the public key <key> belongs to <subject name>" (details elided). We only use "trust" in connection to "trusted third parties" - when an authenticated entity makes assertions about itself we do not speak of trusting them to do so.

Trust (in user interfaces) One of the many alternative meanings of "trust", and the only alternative one used in this document, relates to user interfaces, namely: a trusted user interface is one that the user can somehow ascertain that it is presented by the operating system or browser platform and _not_ by some possibly malicious peer.

Trust Path Continuing with the horrible word "trust", we use "trust path" to the note the list of trusted third parties involved in authenticating an entity to a relying party. This list is ordered, though it could conceivably be set of lists when multiple trust paths are possible.

Trusted Third Party An entity that can be relied up -by those relying parties that trust it- to make assertions relating to other entities, typically assertions about how to authenticate those entities and/or of facts relevant to authorization at the relying party.

[[anchor1: Fill out! Add some entries for OAuth, Kerberos, Basic, DIGEST-MD5, EAP, GSS, SASL, ...]]

2. Background

Web applications today use a variety of user authentication methods, many of which are somewhat or deeply unsatisfying. Almost all of these methods involve the user-agent being mostly dumb - not participating in any cryptographic protocols other than TLS.

The most common user authentication methods used in web applications today include:

- Username and password POSTed to the server from an HTML form. Usually the URL to post to is an HTTPS URL. Not as often the URL of the HTML page containing the form is also an HTTPS URL.
- HTTP Basic or DIGEST-MD5 authentication.
- Out-of-band methods:
  - PINs sent to user devices via SMS (POSTed along with passwords)
  - OTP tokens (POSTed along with passwords)
  - login URLs e-mailed to the user
  - passwords e-mailed to the user

Not much use is made of TLS user certificates, though that is available as well.

These methods are somewhat-to-highly unsatisfactory for a variety of reasons:

- Users have to remember/carry too many passwords, even when they have many fewer "identities" (typically in the form of e-mail addresses).
  - Credential sharing becomes a problem: compromise of one site can result in compromise of user accounts at unrelated sites. Also, a malicious site posing as a friendly site can do the same.
- The service is generally not authenticated to the user. TLS does authenticate the server, but not necessarily the service, and anyways only to the best of the TLS server PKI's ability.
  - This problem derives in part from the nature of the HTTP URI scheme: by identifying server hosts rather than services the HTTP URI scheme fails to provide the user and user-agent with

enough information by which to identify, and thence authenticate, a service. New URI schemes may be required.

- User credentials are too easy to "phish".
- OTP and out-of-band methods do not protect against MITMs, and thus depend on the integrity of TLS and the TLS server PKI.
- HTTP/Negotiate[[RFC4559]], which effectively uses GSS-API[[RFC2743]] mechanisms, usually NTLM [XXX Add reference] or Kerberos[[RFC4120], [RFC4121]].

Additionally, there is no strong concept of "sessions" in web applications. Sessions, such as they are, consist of HTTP requests and responses united into a session by the web cookies they bear. Not all web cookies are used for identifying sessions, and there is no simple "logout" functionality. The biggest problem with web cookies is that they are too easy to misuse or steal (e.g., given the occasional TLS vulnerability, such as BEAST [XXX Add references!]).

Furthermore, there are uncomfortable user interface (UI) problems. In particular it is difficult to convey to the user information about the server's/service's identity and how it is authenticated (if at all).

HTTP applications that are not web application have similar issues, though some of them can also use SASL[[RFC4422]]. Non-web HTTP applications also may not need cookies, instead using a single HTTP/1.1 persistent connection over which to issue all requests that make up a session - such applications have a stronger sense of session than web applications do.

[[anchor2: XXX Finish this section.]]

2.1. Threat Models

[[anchor3: Talk about threat models and which are appropriate for HTTPbis. Discuss the Internet threat model and its flaws (namely/ primarily, the local security assumption).]]

2.2. On Trust

[[anchor4: Describe issues w.r.t. "trust", such as transitivity, introductions, and so on. This is important for evaluating proposals. A proposal that replaces the TLS server PKI's primacy with... another system with similar transitive trust issues may not be a useful proposal. On the other hand, it seems impossible to avoid transitive trust when scaling to Internet scale. Understanding

this may help, for example, give impetus to improvements to the TLS server PKI, or it may guide replacements, understand scalability, and so on.]]

2.3. On Mutual Authentication and URI Schemes

[[anchor5: Describe the limitations imposed by the Internet threat model when there is no mutual authentication. Describe the two types/senses of mutual authentication: authenticating the server (in addition to the client) and key confirmation. Describe the limitations, imposed by the HTTP URI scheme, on service identification and authentication.]]

2.4. On Authentication Mechanism Message Counts

All authentication mechanism require some number of messages in order to authenticate an entity. For example, TLS generally requires two round-trips, while OAuth requires a single message from the client to the server. Here we count only messages from the HTTP client to the HTTP server; additional message exchanges may be required involving trusted third parties.

The number of authentication messages that must be exchanged for a given authentication mechanism is important. The API of at least one important credential management facility is premised on authentication mechanisms having exchanges of just one message - adding new API is possible, but it would take a long time for applications to begin using it. Thus mechanisms that require just one message are at a premium (but see the next section).

The number of authentication messages is also important for latency reasons: since authentication message exchanges are synchronous, each round trip time is added to the latency observed by the user.

The number of messages that an authentication mechanism needs to exchange with infrastructure (e.g., trusted third parties) also affects latency, but at least applications need never be aware of messages exchanged with infrastructure - these can be abstracted away by the APIs. Some authentication mechanisms have fast re-authentication facilities such that the latency cost of infrastructure messaging need not be incurred as frequently as the entity authenticates to others.

[[anchor6: ...]]

2.5. On Channel Binding and One-Message Authentication Mechanisms

Channel binding [[RFC5056]] is the act of binding authentication at one network layer to key exchange at a lower network layer. When this occurs within the same layer we don't call it channel binding, but the same concept is involved. For example, TLS PSK and user certificates are cryptographically bound to whatever key exchange method is used, but because this happens naturally within TLS we don't call it channel binding. [Expand on this for the benefit of those not familiar with RFC5056.]

Normally channel binding requires mutual authentication, either in the key confirmation sense or in the sense of actually authenticating the server. In order to see why imagine a one-message user authentication system: a man-in-the-middle (MITM) at a lower layer might be able to steal this one message, close the connection to the real client, then impersonate the client to the server. There are ways of preventing this, but they are not as general as requiring mutual authentication is.

At one point a SASL mechanism, "YAP", was proposed that requires just one message and provides channel binding. In order to prevent the message theft problem described above YAP requires that tls-unique channel bindings be used, which effectively eliminates MITMs at the TLS layer. At the time the SASL community rejected this proposal, mostly on account of not wanting to have SASL be aware of the type of channel bindings data used by the application. In retrospect, however, the idea has merit.

Now consider an authentication system that begins life as a bearer token and later is upgraded to be a bearer token that is encrypted in the server's public key, the same public key as is expected to be used by the server in TLS. This bearer token can still be stolen and used by the thief... unless the TLS client knows to ensure that the same public key is used at both layers. But how might the client know how to do that? If the client passes the server's certificate to the client's IdP then the most the IdP can do is apply certificate validation, including certificate/CA/public key pinning options; if the IdP doesn't do this then the MITM will be able to decrypt the bearer token and then re-encrypt it in the real server's public key. This can be overcome by having the IdP do better certificate validation or knowing the target server's certificate a priori, with all the same problems as the traditional TLS server PKI (which is not necessarily a problem). It's not clear how one might successfully apply unique or client end-point channel bindings to a bearer token authentication system, but if there's a way to do so it would help.

YAP is a proof-of-possession mechanism, of course, thus it is quite

simple to apply channel binding types other than server end-point, thus making YAP secure against message theft and re-use where a bearer token system could not be. The point being that a secure half round trip (one message) user authentication mechanism is feasible.

Work is ongoing to develop a version of OAuth that is capable of channel binding. [XXX Add references!]

2.6. Logon Sessions

[[anchor7: Discuss the binding of HTTP requests (and responses) to logon sessions. Discuss logout.]]

2.7. Web Cookies, a Form of Bearer Tokens

[[anchor8: Discuss cookies as a form of bearer token and how the situation is not as dire as with bearer tokens for user authentication. Discuss alternatives based on MACing portions (or all) of the HTTP requests (and responses) or the channel bindings data for the TLS channel.]]

2.8. User Interface Issues

[Discuss phishing issues, in particular the difficult of creating user interfaces in web apps that cannot be spoofed by either server impersonators or MITMs. Reference Sam Hartman's anti-phishing I-D [[I-D.hartman-webauth-phishing]].]

3. Classification Axes

Several orthogonal classification axes are proposed:

1. Dependence on/independence of the TLS server PKI;

2. Solutions based on bearer tokens vs. ones based on proof of possession;

3. Layer at which user authentication takes place: TLS, HTTPbis, or the application layer (note: distinguishing network layer from API layer);

4. Whether the client, the server, or both, engage in infrastructure messaging;

5. Number of messages exchanged / "round trips";

6. Trust establishment: pair/group-wise non-transitive, federated or otherwise transitive, hierarchical vs. mesh;

7. Threat modeling;

8. Explicit versus implicit session management;

9. In-band / out-of-band.

[[anchor9: Maybe add something about separation of password verifier access, to limit the attack surface area for password recovery?]]

[[anchor10: Note: The author assumes that all acceptable proposals will have HTTPbis continue to depend on TLS for transport security - for confidentiality (encryption) and integrity (authentication) protection of data exchanged by the HTTPbis client and server. If this assumption is incorrect then we can add one more axis of classification: dependence on / independence of TLS.]]

These nine classification axes are largely orthogonal to each other. Other classification criteria are also possible and may be added in future versions of this Internet-Draft. Some such possible additional criteria are subjective, such as, for example: ease of deployment, ease of implementation, etcetera. Perhaps the WG can come to consensus regarding desirable properties based on objective classification to narrow the set of proposals to consider. Or perhaps the WG can consider a large number of proposals and use objective classification to guide any applicability statements for the proposals accepted. Ideally the WG can apply objective classification first, then for each "bucket" of similar proposals the

WG could consider more subjective classification criteria.

3.1. Dependence on TLS Server PKI

The web today depends utterly on the "TLS server PKI" for security. This would be just fine were it not for the systemic weaknesses in the TLS server PKI: the lack of name constraints, the large number of trust anchors, the large number of certificate authority (CA) compromises, and so on. Building on the TLS server PKI and thus assuming its being sufficiently secure, is quite tempting, as it may simplify various aspects of user authentication (not least by providing server authentication a priori, thus saving the designers the need to provide server authentication themselves).

This classification axis is very simple: either a proposed solution depends on the TLS server PKI or it doesn't. Some shades of black are imaginable in this case (if not likely).

3.2. Bearer Tokens vs. Proof of Possession

A bearer token is a message the presentation of which is sufficient to authenticate the presenter. Stolen bearer tokens may be used to trivially impersonate the subject, thus bearer tokens generally require confidentiality protection in any protocols over which they might be exchanged, and generally depend on authenticating the relying party first.

Proof of possession systems consist of some secret/private key(s), an authenticator message the "proves" possession of the secret or private key(s) used in the construction of the authenticator, and a token not unlike a bearer token but which securely indicates to the relying party(ies) what keys the user must have used in the construction of the authenticator. The relying party then validates the authenticator to establish that the user did indeed possess the necessary secret/private key(s) to the best of the cryptographic capabilities of the authentication system used.

3.3. Layer at which Authentication Protocol Operates

It is possible to design user (and mutual) authentication mechanisms that can work at any end-to-end layer between the HTTPbis client and server. The relevant layers are:

- TLS,

- HTTPbis,

o and the application layer.

We dismiss out of hand the possibility of that layer being TCP or IPsec, though admittedly they are also end-to-end layers where user authentication could theoretically be done.

We distinguish between network layers and API layers (see glossary). A solution at the application _network_ layer might nonetheless be implemented at the HTTP _API_ layer (and vice-versa).

User authentication is generally something that a transport layer cannot know to initiate on its own: the application must be in control of when (server- and client-side) to authenticate, how (server- and/or client-side), with what credentials / as whom (client-side). This means that authentication in the transport layer requires APIs that give the application a measure of control. HTTP API capabilities will vary, but HTTPbis is a good opportunity to standardize an abstract API outlining capabilities and semantics to be exposed to applications by an HTTP stack.

Note that on the user-agent side the platform may provide user interaction facilities for authentication, thus simplifying user authentication APIs. The application, on the server side, remains in control over when to initiate authentication.

End-to-end session cryptographic protection is best done in the lowest possible transport layer. For HTTP applications, historically this means TLS; though it'd be technically feasible to provide protection at lower layers it does not appear to be a realistic option at this time.

User authentication is best "bound" into transport security layers, in this case TLS. When user authentication is moved to higher layers a "channel binding" problem arises: we would like to ensure that no man-in-the-middle exists in the transport layer, with the MITM terminating two TLS connections. For more information about channel binding see [[RFC5056]].

UI and API issues are quite different for web applications versus non-web applications. The former have rich UI elements (all of HTML's) and programming models (scripting, particularly through JavaScript). One problem that is particularly severe for web applications, is the ability of server impersonators to emulate all imaginable graphical user interfaces that the native user-agent might wish to use to distinguish itself from the applications it runs. Regardless of what layer implements authentication this problem will arise in web applications.

3.3.1. HTTP- vs. Application-Layer Authentication in the Network Stack

It's important to note that there need not be much difference between HTTP-layer and application-layer user authentication, at least if we assume a standard application-layer user authentication convention. For argument's sake let's assume an application-layer user authentication convention like the one in [[I-D.williams-rest-gss]], and let's assume two possible HTTPbis HTTP-layer authentication solutions: one that is most similar to HTTP/1.1's and one that uses a new verb for authentication. Then let's look at what each of these three solutions look like on the wire using the SCRAM mechanism for cases where the client already knows it has to authenticate. For brevity we elide any HTTP request and response where the server indicates that the client must authenticate, as well as any requests/responses involving negotiation of mechanism to use.

```
C->S: HTTP/1.1 POST /rest-gss-login
      Host: A.example
      Content-Type: application/rest-gss-login
      Content-Length: nnn

      SCRAM-SHA-1,,MIC
      n,,n=user,r=fyko+d2lbbFgONRv9qkxdawL

S->C: HTTP/1.1 201
      Location http://A.example/rest-gss-session-9d0af5f680d4ff46
      Content-Type: application/rest-gss-login
      Content-Length: nnn

      C
      r=fyko+d2lbbFgONRv9qkxdawL3rfcNHYJY1ZVvWVs7j,
      s=QSXCR+Q6sek8bf92,i=4096

C->S: HTTP/1.1 POST /rest-gss-session-9d0af5f680d4ff46
      Host: A.example
      Content-Type: application/rest-gss-login
      Content-Length: nnn

      c=biws,r=fyko+d2lbbFgONRv9qkxdawL3rfcNHYJY1ZVvWVs7j,
      p=v0X8v3Bz2T0CJGbJQyF0X+HI4Ts=

S->C: HTTP/1.1 200
      Content-Type: application/rest-gss-login
      Content-Length: nnn

      A
      v=rmF9pqV8S7suAoZWja4dJRkFsKQ=
```

Figure 1: REST-GSS Login w/ SCRAM Example

Figure 1

```
C->S: HTTP/1.1 LOGIN
      Host: A.example
      Content-Type: application/SASL
      Content-Length: nnn

      SCRAM-SHA-1,,MIC
      n,,n=user,r=fyko+d2lbbFgONRv9qkxdawL

S->C: HTTP/1.1 201
      Location http://A.example/login-session-9d0af5f680d4ff46
      Content-Type: application/SASL
      Content-Length: nnn

      C
      r=fyko+d2lbbFgONRv9qkxdawL3rfcNHYJY1ZVvWVs7j,
      s=QSXCR+Q6sek8bf92,i=4096

C->S: HTTP/1.1 LOGINCONTINUE /login-session-9d0af5f680d4ff46
      Host: A.example
      Content-Type: application/SASL
      Content-Length: nnn

      c=biws,r=fyko+d2lbbFgONRv9qkxdawL3rfcNHYJY1ZVvWVs7j,
      p=v0X8v3Bz2T0CJGbJQyF0X+HI4Ts=

S->C: HTTP/1.1 200
      Content-Type: application/SASL
      Content-Length: nnn

      A
      v=rmF9pqV8S7suAoZWja4dJRkFsKQ=
```

Figure 2: HTTPbis w/ New Verb Login w/ SCRAM Example

Figure 2

```
C->S: HTTP/1.1 GET /location/of/interest/to/app
      Host: A.example

S->C: HTTP/1.1/401 Unauthorized
      Server: HTTPd/0.9
      Date: Sun, 10 Apr 2005 20:26:47 GMT
      WWW-Authenticate: <list of mechanisms>
      Content-Type: text/html
      Content-Length: nnn

      <error document>

C->S: HTTP/1.1 GET /location/of/interest/to/app
      Host: A.example
      Authorization: SCRAM-SHA-1,,MIC
                     n,,n=user,r=fyko+d2lbbFgONRv9qkxdaw

S->C: HTTP/1.1 4xx
      WWW-Authenticate: C
                        r=fyko+d2lbbFgONRv9qkxdawL3rfcNHYJY1ZVvWVs7j,
                        s=QSXCR+Q6sek8bf92,i=4096
      WWW-Authenticate-Session: 9d0af5f680d4ff46

C->S: HTTP/1.1 GET /location/of/interest/to/app
      Host: A.example
      Authorization-Session: 9d0af5f680d4ff46
      Authorization: c=biws,r=fyko+d2lbbFgONRv9qkxdawL3rfcNHYJY1ZVvWVs7j,
                     p=v0X8v3Bz2T0CJGbJQyF0X+HI4Ts=

S->C: HTTP/1.1 200
      WWW-Authenticate: A
                        v=rmF9pqV8S7suAoZWja4dJRkFsKQ=
      Content-Type: ...
      Content-Length: nnn

      <content>
```

Figure 3: Extended HTTP/1.1 Style Login w/ SCRAM Example

Figure 3

There's not much difference between the first two examples. The third example has several important differences relative to the first two examples:

- The URL is sent to the server before any chance to have completed mutual authentication, should the selected mechanism provide mutual authentication. If the client knows a priori to

authenticate and the URL contains sensitive information then the client has no choice but to leak this information prior to completing mutual authentication, thus the client becomes dependent on TLS for authenticating the server even when the client could authenticate the server more strongly via the selected HTTP authentication mechanism. This is an important weakness.

- The whole sequence involves multiple requests/responses, which goes against the stateless nature of HTTP. State is needed in all three examples, but the first example is RESTful, while the second employs a would-be new verb that provides for stateful authentication. The third example simply cannot be thought of as remotely RESTful. Perhaps this is not a problem.

  - Alternatively mechanisms requiring multiple round trips can be ruled out of scope. This would rule out quite a few desirable mechanisms!

The main difference on the wire between a generic HTTP-layer user authentication framework (like the one in the second example) and an application-layer equivalent (as in the first example) can be so minimal as to make the choice of layer seem like splitting hairs.

3.3.2. HTTP- vs. Application-Layer Authentication in the API Stack

There are HTTP stacks that make it possible to implement HTTP authentication methods in the application (e.g., FCGI in web servers), and nothing would prevent HTTP stacks from implementing a _standard_ application-layer user authentication protocol either. The APIs offered by an HTTP stack should look remarkably similar regardless of which layer the user authentication protocol is technically at. Once again, the difference between HTTP-layer and standard application-layer user authentication is minimal.

Note however that if the HTTP stack does not implement authentication, leaving it to the application to do so, then the application developer runs the risk of making mistakes in the implementation, such as failing to implement channel binding where possible. Thus it is generally best if the HTTP stack implements authentication - even if TLS is used for user authentication, the HTTP stack should provide a singular API for authentication.

3.3.3. Choice of Layer

The choice of layer is clearly more important for APIs than on the wire. On the wire the choice of layer is minimal, trivial even, when the choice is between HTTP and the application layer.

If the WG agrees that the distinction between HTTP-layer and application-layer user authentication is or should be minimal then how should the WG pick one of those two layers, if it decides not to pursue TLS-layer user authentication?

A standard application-layer authentication scheme implies no changes to HTTP itself, and may not rely on any particular features of HTTP/1.1 or HTTPbis, thus it may be usable even with HTTP/1.0. This is true of the REST-GSS proposal[[I-D.williams-rest-gss]], which is also RESTful. This must be of some value.

An HTTP-layer authentication solution must either: a) not support multi-round trip mechanisms, b) add verbs, or c) not be RESTful. (a) works with HTTP/1.0, (b) would not work with HTTP/1.0. [The author believes that RESTfulness is desirable.]

3.3.4. User Authentication in the TLS Layer

Issues:

- The transport cannot know when to require user authentication (on the server side) or when to initiate it (on the client side). Simply always initiating user authentication creates privacy problems: the user may not want to disclose their identity all the time!

- To address the problem of when to require or initiate user authentication the TLS implementation must provide suitable APIs to the application. And since the application will generally decide that authentication is required only after (possibly well after) a TLS connection is setup, the user generally must be authenticated by renegotiating TLS, which in turn means that two round trips will be needed just for that, at minimum, even if the user authentication mechanism selected requires fewer round trips. This is inefficient, though not fatal.

- The TLS community has resisted proposals for user authentication mechanisms with arbitrary round trip counts before [references? this is in reference to Stefan's TLS-GSS proposal...]. This may no longer be true (or perhaps the author is misunderstanding or misremembering the events in question), but if it is still the case then the range of choices for user authentication in TLS is significantly curtailed.

- Several major TLS implementations defer certificate validation until the peer's Finished message is received. This means that unless one is using TLS renegotiation (with the inner connection's server certificate being the same as in the outer connection's)

the user's identity and the payloads related to user authentication will be revealed to the server before the server is authenticated.

- User Interface issues:
  - A user authentication framework and future mechanisms will likely need to interact with the user. In some cases this may be best done through a platform component, such as a credential management facility. In other cases this may best be done by the application. Driving user interaction from within the TLS layer presents a slight complication: any interaction has to be effected through application- or platform-provided code paths. Adding interaction to existing TLS implementations may not be trivial.
  - ...

Benefits:

- Where the platform can provide credential management and user interaction then user authentication in TLS can greatly simplify HTTP applications: no user authentication APIs or UIs are then needed in the application.
  - Note however that the user may have a hard time identifying the context in which they are being prompted by the system for credentials or credential selection. This is usually not a problem in smart-phone and other such small devices, where it is generally clear what application is in the foreground, and therefore the context of a prompt. But this is not necessarily so on other platforms.
- Non-web applications typically know a priori when they wish to authenticate. Typical non-web applications that use HTTP/1.1 over a single TLS connection, with an application session consisting of all the HTTP requests performed over that one connection. For such applications having user authentication in the TLS layer may be the simplest way to get user authentication into the application.

3.4. Party Responsible for Infrastructure Messaging

[[anchor11: XXX Add references for OCSP, AAA, ...]]

"Infrastructure" consists, for the purposes of this document, of services such as Identity Providers (IdPs), Certificate Revocation Lists (CRLs) and their servers, Online Certificate Status Protocol

(OCSP) responders, Kerberos Key Distribution Centers (KDCs), RADIUS/ DIAMETER servers, etcetera. These are services that run on parties other than a client (e.g., a web browser / user agent) and an application server. In some cases infrastructure services may be physically co-located with the client or server, but by and large they are physically separated; infrastructure services are always logically separate from the client and server. [XXX Move this to glossary.]

Some protocols require that the client do all or most of the message exchanges with infrastructure, some require that the server do this messaging, some require both to do some messaging. In some cases a server might proxy a client's messages to infrastructure. There are advantages to the client doing this messaging: namely a simpler server, less subject to denial of service / resource consumption attacks. [Are there advantages to the server doing this messaging?]

Consider a protocol like Kerberos. Kerberos relies on Key Distribution Center (KDC) infrastructure, and it relies on the client doing all the messaging needed to ultimately authenticate it to a server. Kerberos can be used in a way such that the relying party proxies this messaging for the client (see IAKERB), but even so the client had to communicate with the KDCs in order to ultimately authenticate to the relying party - IAKERB is simply a proxy mechanism.

Now consider an authentication mechanism based on PKI. The only online infrastructure in a PKI are the CRLs and OCSP responders. Of course, a Certificate Authority (CA) can also be online, as in kca [add reference], a CA that authenticates clients via Kerberos and which issues fresh, short-lived certificates. Private keys for certificates can also be served by online services such as SACRED and browserid. The method of validating certificates currently considered ideal is for the possessor of certificate's private key to send both, the certificate and a current/fresh OCSP response for it (or, rather, responses, for the entire certificate chain), thus the PKI relying party should ideally not have to contact infrastructure; in practice CRL checking is still the more commonly used method, requiring infrastructure messaging on the relying party side.

The responsibility for infrastructure messaging varies widely.

3.5. Number of Messages

The number of messages that must be exchanged in order to authenticate a peer varies a lot by authentication mechanism. Some require just one message from the client to the server. Others require a reply message from the server. Others require some larger

number of messages (typically three or four). Yet others require a variable number of messages.

Typically key exchange is also required in order to provide confidentiality and integrity protection to the transport. Key exchange protocols also vary in number of messages required. Key exchange and authentication may be combined, either directly in a single network layer, or across layers via channel binding.

One-message authentication protocols:

- OAuth
- Kerberos (w/o key confirmation)
- Public key signature schemes when authenticating only the client
- Diffie-Hellman (when the client knows the server's DH public key a priori, and w/o key confirmation)
- RSA key transport (w/o key confirmation)
- all bearer token protocols (but see [ref to on channel bindings section])

Two-message authentication protocols:

- Kerberos
- Diffie-Hellman with fixed public keys
- RSA key transport

Authentication protocols with three or more messages, or with arbitrary numbers of messages:

- Most/all zero-knowledge password proof protocols (e.g., SRP) (usually three or four messages)
- SCRAM, and other challenge-response protocols (usually three or four messages)
- IAKERB (usually four messages)
- Pluggable frameworks (SASL, GSS, EAP) (arbitrary message counts, usually dependent on what mechanism is selected)

It's worth pointing out that TLS is a three- to four-message

protocol, but when providing confidentiality protection for the client identity it becomes a six- to eight-message protocol (though there is a proposal to improve this, getting back to three to four messages [add reference to Marsh's I-D]).

Some authentication protocols can provide key exchange, others cannot. Similarly, not all mechanisms can provide channel binding.

The total number of messages required is important. These message exchanges are always ordered and synchronous; no progress can be made by the application until they are completed. Over long distances the time to complete each round trip add up to noticeable latency, and there is much pressure to get this latency down to an absolute minimum.

Integrating user authentication into TLS has the clear allure of potentially cutting down the number of round trips necessary, but it's not clear that this can be achieved in every case. In particular it may not be clear that a client has to authenticate until after a TLS connection is established over which the client may request access to some resource that requires authenticated clients.

3.6. Trust Establishment

Pair-wise pre-shared keying systems require careful initial key exchange, but otherwise have no transitive trust issues: every pair of entities that has shared keying can communicate without the aid of any other entity. However, pair-wise pre-shared keying does not scale to the Internet as it is O(n^2), and it requires either "leap of faith" (a.k.a., trust on first use, or TOFU) or physical proximity for the key pre-sharing. Physical proximity

Authentication mechanisms that scale to the Internet of necessity require some degree of trust transitivity. That is, there must be many cases where Alice and Bob can communicate with each other only because they can authenticate each other by way of one or more third parties (e.g., Trent) that each of them trust a priori.

There are a number of issues with trust transitivity:

- Trusted third parties can mount MITM attacks on the parties that rely on them
  - Compromise of trusted third parties, therefore, has far reaching, negative effects
  - The longer a trust path, the less trustworth -so to speak- it is

- Policy for determining acceptable trust paths is difficult to express
- Mechanisms for establishing trust paths are often manual and prone to error or abuse

There are several ways to use transitive trust. In hierarchical transitive trust we organize the trusted third parties in such a way that there should be a trust path for every pair of entities of interest (e.g., every user to every server, every user to every user, ...) - think of PKI. In mesh systems trust transits through every entity's "friends" - think of PGP.

There may be other models of transitive trust, such as one with islands of trust. An islands of trust model would consist of federations of transitive trust (using hierarchical or mesh models) that are much smaller than the entire Internet, but large enough to be of use to large numbers of users. For example, an online merchant might provide for authentication of all users to a set of participating vendors [XXX expand on this].

Given the need for transitive trust and the serious drawbacks of transitive trust, some workarounds may be necessary, such as:

- Policy language for choosing suitable trust paths
- Facilities for limiting the length of, or otherwise shortening trust paths
  - By, for example, providing for bootstrapping of shorter trust paths when a given trust path involves an "introducer" trusted third party.
- "Pinning" facilities to force changes in the infrastructure to proceed in ways which make some MITM attacks harder to mount
- Auditing -and compromise detection- facilities by which to show that trusted third parties are not mounting MITM attacks
- Revocation facilities that actually work
- Root keys that are rarely used and live in HSMs
- Fast re-keying as a method for dealing with trusted third party compromise

For an example of pinning, consider a TLS extension where self-signed, persistent user certificates are used, possibly one per-

origin for pseudonymity purposes. The user agent can enroll the user certificates at their corresponding origin servers such that thereafter no MITMs are possible that can impersonate the user to the server. Of course, such a scheme suffers from needing a fall-back authentication method when the user's device(s) that store the relevant private keys are lost. Users would need to be able to fall-back on an alternative authentication method for re-enrollment, likely one that is susceptible to attack or else is inconvenient. In this cases the pinning is on the server side; keep in mind that pinning need not only be used on clients, but may be used even in the distributed trust infrastructure (e.g., to shorten trust paths).

Ideally an authentication facility for HTTP/2.0 should support a variety of trust establishment models, as it is not clear that one mode is superior to the others. (Though certainly the hierarchical model is likely the scheme that can have the most universal reach, and therefore most minimize user credentials needed. However, users may not mind having a small number of logon credentials for a trust island model.)

3.7. Threat Modeling

[[anchor12: Cover the Internet threat model. Discuss the end-to-end model and the hop-by-hop semantics of transitive trust.]]

3.8. Explicit versus Implicit Session Management

[[anchor13: Discuss lack of / weakness of application session concept on the web. Discuss the historically limited application of TLS sessions to HTTP apps. Discuss desirability of a real concept of session and logout.]]

3.9. In-Band versus Out-of-Band Authentication

[[anchor14: Discuss out-of-band user authentication systems such as ones where "tokens" are sent to users' mobile phones via SMS, as well as systems where a "login URL" is sent to the user via e-mail.]]

4. Analysis of Some Possible Authentication Proposals

[Cover:

- Authentication mechanisms:
  - Bearer token systems
  - Other half round trip systems, including Kerberos, OAuth
  - PK w/ SACRED, browserid, smartcards
  - ZKPPs
  - Challenge/response password-based mechanisms (DIGEST-MD5, SCRAM)
- Generic auth frameworks
  - GSS, SASL, EAP (anything else? IKEv2? SSHv2?)
- Authentication in TLS, HTTP, and above HTTP
- OTP and out-of-band (SMS, e-mail) auth, both as part of authentication mechanisms and as port of traditional webauth.
- Traditional webauth (passwords posted in forms), possibly with password wallets (stateful and stateless)

]

[[anchor15: What else to cover?]]

5. Author's Recommendations

It seems likely that no single user authentication method will satisfy the needs of all web applications. Nor can we predict the future. Moreover, some weak authentication approaches are perfectly safe for accessing low-value resources, or in contexts where the Internet threat model is overkill. This argues for a multitude of solutions, and possibly a pluggable system.

The author proposes the following:

1. For all authentication mechanisms (i.e., cryptographic authentication methods) use the GSS-API, possibly through the thin shim of RFC5801 [XXX change into reference].

    1. do this above HTTP in the network stack, but...

    2. ...recommend that this be implemented by HTTP stacks, rather than by applications. I.e., authentication above HTTP on the wire, within HTTP as far as APIs are concerned.

2. Encourage development of authentication mechanisms that fit the chosen authentication framework and which have the following features:

    1. federation (even though it implies trusted third parties)

    2. strong initial user authentication (e.g., with ZKPPs)

    3. minimized password verifier attack surface area (e.g., minimize the number of servers that have access to password verifiers)

    4. trust path bootstrapping

    5. short trust paths

    6. auditable trusted third parties

    7. [preferably] mutual authentication

3. Standardize weak authentication mechanisms (e.g., passwords POSTed in forms) to facilitate the development of effective password managers. [This is primarily for low-value sites.]

4. Specify HTML and JavaScript interfaces for initiating authentication, including the name of the service to authenticate to. This will allow login pages to have a customized look, yet

allow for login operations to be performed by the browser platform using a strong authentication mechanism. Specifically there must be a method for kick-starting authentication such that the user and/or device identity and credential input does not happen through HTML forms but through browser/platform trusted user interfaces.

5. Specify a new URI scheme that identifies services rather than hosts. For example: svc:<service>@<domainname>/<local-part> . An option to embed service authentication information (possibly a digital signature, or a URL referring to a digital signature) may prove useful.

    1. Also specify a service location protocol.

6. Specify an abstract API for interfacing HTTPbis applications to HTTPbis.

6. References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, August 2008.

[RFC2616] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and T. Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1", RFC 2616, June 1999.

[RFC5056] Williams, N., "On the Use of Channel Bindings to Secure Channels", RFC 5056, November 2007.

[RFC4120] Neuman, C., Yu, T., Hartman, S., and K. Raeburn, "The Kerberos Network Authentication Service (V5)", RFC 4120, July 2005.

[I-D.williams-rest-gss] Williams, N., "RESTful Hypertext Transfer Protocol Application-Layer Authentication Using Generic Security Services", draft-williams-rest-gss-00 (work in progress), June 2011.

[I-D.hartman-webauth-phishing] Hartman, S., "Requirements for Web Authentication Resistant to Phishing", draft-hartman-webauth-phishing-09 (work in progress), August 2008.

[RFC2818] Rescorla, E., "HTTP Over TLS", RFC 2818, May 2000.

[RFC4422] Melnikov, A. and K. Zeilenga, "Simple Authentication and Security Layer (SASL)", RFC 4422, June 2006.

[RFC5802] Newman, C., Menon-Sen, A., Melnikov, A., and N. Williams, "Salted Challenge Response Authentication Mechanism (SCRAM) SASL and GSS-API Mechanisms", RFC 5802, July 2010.

[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.

[RFC3748] Aboba, B., Blunk, L., Vollbrecht, J., Carlson, J., and H. Levkowetz, "Extensible Authentication Protocol (EAP)", RFC 3748, June 2004.

[RFC2743] Linn, J., "Generic Security Service Application Program Interface Version 2, Update 1", RFC 2743, January 2000.

[RFC4559] Jaganathan, K., Zhu, L., and J. Brezak, "SPNEGO-based Kerberos and NTLM HTTP Authentication in Microsoft Windows", RFC 4559, June 2006.

[RFC4121] Zhu, L., Jaganathan, K., and S. Hartman, "The Kerberos Version 5 Generic Security Service Application Program Interface (GSS-API) Mechanism: Version 2", RFC 4121, July 2005.

Author's Address

Nicolas Williams
Cryptonector, LLC

Email: nico@cryptonector.com