httpauth
Internet-Draft
Intended status: Experimental
Expires: August 31, 2017

J. Woodworth
D. Ballew
CenturyLink, Inc.

March 05, 2017

HTTP Authentication - |JSON| Scheme
draft-woodworth-json-http-auth-01

Abstract

The |JSON| authentication scheme provides a mechanism for exchanging authentication challenges and credentials as objects in the form of JavaScript Object Notation (JSON). This scheme offers a secure mechanism of providing authenticated access to a set of protected HTTP resources which may be handled by scripting utility framework as in XMLHttpRequest calls (AJAX) or directly by the client's user agent. This chaining feature is unique to this scheme.

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.

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-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."

Copyright Notice

(http://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents

Table of Contents:

1. Introduction

The |JSON| authentication scheme offers a number of new concepts used to extend or enhance current HTTP-Authentication facilities.

New concepts include; JSON Based Challenge/ Response Payloads, Chained Authentication Cursor (CAC), Extensible Authentication Indicators (EAI) and One-Time Password (OTP) Authentication Support.

1.1. Background and Related Documents

This document assumes the reader is familiar with the basic HTTP and HTTP-Authentication concepts described in [RFC7230] and [RFC7235].

The reader is also assumed to be familiar with the JSON data interchange format and associated terminology described in [RFC7159] as well as the Base64 encoding methods as described in [RFC4648].

1.2. Reserved Words

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].

2. The |JSON| HTTP-Authentication Scheme

The |JSON| scheme provides an extensible mechanism of providing primary HTTP authentication to a web-based application. This section will cover a number of new concepts intended to extend the flexibility and capabilities of existing HTTP authentication schemes.

This scheme follows the methods defined in [RFC7235] to provide authentication challenge and response via the HTTP-Headers "WWW-Authenticate" and "Authorization" respectively. For the moment at least, this document hereby defines proxy authentication as out of scope.

2.1. JSON Based Payloads

A primary motivation for this document is to offer the ability to leverage scripting capabilities within the client application. While this does fall outside the HTTP protocol itself, modern HTTP clients have advanced to a point where scripting and asynchronous calls to back-end web services have become quite common. These advancements offer an integration between the HTTP protocol and client side logic processing (i.e. scripting).

One such advancement along this path is the ubiquity of JSON encoded objects. These objects can be quickly processed and passed over-the-wire to almost any programming language and are native to the default scripting engine of modern browsers.

By leveraging the JSON format for passing challenges and responses back and forth a type of protocol-logic bridge can be easily established.

This document defines a single JSON object containing zero or more elements where the set of elements is defined by its authentication "type" and whether it is a challenge or response. This document defines 3 types, "password", "challenge", and a hybrid one-off variant each described in detail in sections to follow. This object is Base64 encoded and assigned to the "data" auth-param as defined in [RFC7235], Section 2.1. Additionally, while many examples show JSON object definitions in this document are shown in a "pretty" format to improve readability the over-the-wire encoding is expected to be in a condensed form (minimum whitespace) prior to Base64 encoding.

Below is a simple example demonstrating the "password" type:

```
WWW-Authenticate: |JSON| realm="Test Realm",
                  data="eyAidHlwZSIgOiAicGFzc3dvcmQiIH0="
```

2.2. Chained Authentication Cursor (CAC)

Beyond the utilization of the JSON format to exchange and validate authentication credentials, this document establishes the concept of a Chained Authentication Cursor (CAC).

The logic used to implement CAC is simpler than the name may imply. In essence, the client's user-agent is currently aware of several authentication schemes, each with their own handler. For example, a handler is defined for the "Basic" scheme, another for the "Digest" scheme, etc. When a challenge is received from the server as a result of a "401 Unauthorized" response, a cursor is assigned and attached to this challenge. This cursor follows the challenge throughout its short life-cycle "Chained" in succession to each of the available user-agent authentication handlers in order to provide the client's response. Each handler may either accept or reject the authentication cursor based entirely upon the scheme passed to it. If the cursor is accepted, it MUST provide a response to be used for the follow-up request.

Some handlers in the chain MAY be defined as Extensible Authentication Handlers (EAH) others as Native Authentication Handlers (NAH). EAHs are extensions to the built-in NAHs and are generally provided as either a client extension or scripted handler and can be used to provide custom or interim authentication solutions. Additional recommendations are provided in this document to avoid "Chicken and Egg" scenarios where EAHs are concerned.

2.3. Extensible Authentication Indicators (EAI)

Schemes used by Extensible Authentication Handlers (EAH) MUST have their intent formally identified by an Extensible Authentication Indicator (EAI). The [US-ASCII] pipe character (|) is reserved for this purpose and if the HTTP-Authentication scheme is surrounded by this character (e.g. "|JSON|", "|Basic|", "|example|", "|pdmk|", "|random|", etc.) an EAH MUST be assumed. If no EAH are defined for an EAI indicated scheme, the indicators MUST be removed and the CAC passed to a handler for the new scheme as next in the chain. For example, if the scheme "|Basic|" is provided in the challenge and no EAH is defined for "|Basic|", then an attempt to pass the CAC to a "Basic" NAH MUST be attempted. This logic MUST be attempted for each EAH in the chain for the active CAC.

2.4. One-Time Password (OTP) Authentication Capability

This document defines provisions for One-Time Password (OTP) Authentication. This concept, while simple to grasp, causes difficulty in a stateless protocol such as HTTP. Recommendations for

overcoming such difficulties are provided in sections which follow.

3. |JSON| Authentication Scheme Types

The following three authentication types are explicitly defined by this document but others may exist under its umbrella so long as none of the defined requirements are violated.

Additionally, the objects and internal elements defined are to be considered a super-set of those implemented and extension or "enrichment" is to be expected and are considered OPTIONAL. However, in order to ensure compatibility all objects and internal elements defined by a type and not defined as OPTIONAL MUST be implemented as defined.

3.1. The "password" |JSON| Authentication Type

This is the simplest authentication type offered. This type is provided in order to offer scripted capability without a lot of knowledge in cryptography. Due to its clear-text nature it is highly recommended to only be used in an encrypted environment (i.e. SSL/TLS, etc.).

The HTTP-Authentication challenge for this type includes the following elements (with definitions):

type: REQUIRED

The |JSON| Authentication Scheme type of "password"

cookie: OPTIONAL

This is the name of the HTTP-Cookie as defined by [RFC6265] which will be used by the server for continuing the session after initial HTTP based authentication has completed

version: OPTIONAL

The |JSON| Authentication Scheme version. This document defines this as 1.0 and version 1.0 MUST be assumed if no version is provided

An example challenge for this type is simply:

```
{ "type" : "password" }
```

The HTTP-Authentication challenge MUST be Base64 encoded and applied to the "WWW-Authenticate" HTTP-Header as defined by [RFC7235], Section 4.1.

The Base64-encoded header with associated realm appears as:

```
WWW-Authenticate: |JSON| realm="Test Realm",
                  data="eyAidHlwZSIgOiAicGFzc3dvcmQiIH0="
```

The HTTP-Authentication response for this type includes the following elements (with definitions):

type: REQUIRED

The |JSON| Authentication Scheme type of "password"

username: REQUIRED

The user associated with this protected resource

password: REQUIRED

The password associated with this protected resource

version: OPTIONAL

The |JSON| Authentication Scheme version. This document defines this as 1.0 and version 1.0 MUST be assumed if no version is provided

NOTE: Since no protection of credentials is offered by the "password" type, the need for other protections such as replay-prevention is unnecessary and therefore not offered as part of this type.

For an example username of "MyUser" and password of "MyPassword" the JSON representation of the response would be:

```
{
   "type"     : "password"
  ,"username" : "MyUser"
  ,"password" : "MyPassword"
}
```

The Base64-encoded header with associated realm appears as:

```
Authorization: |JSON| realm="Test Realm",
               data="eyAidHlwZSIgOiAicGFzc3dvcmQiLCAi
               dXNlcm5hbWUiIDogIk15VXNlciIsICJwYXNzd2
               9yZCIgOiAiTXlQYXNzd29yZCIgfQ=="
```

NOTE: The "data" element has been expanded to multiple lines for readability purposes only

3.2. The "challenge" |JSON| Authentication Type

The challenge authentication type provides features to protect the credentials against replay-prevention and over-the-wire interception.

The HTTP-Authentication challenge for this type includes the following elements (with definitions). More details are provided in the implementation considerations sections below.

type: REQUIRED

The |JSON| Authentication Scheme type of "challenge"

algorithms: REQUIRED

A comma separated list of accepted algorithms as defined by [FIPS-180-4] and [FIPS-202] in order of preference (OPTIONAL whitespace MUST be ignored during selection described below).

NOTE: This list of algorithms represents all algorithms the server is capable of and willing to verify hashed passwords received by a client against. In other words, hashed passwords MUST either be known to the server or generated by the server by way of hashing known clear-text passwords for each provided algorithm

NOTE: Use of the SHA-1 algorithm is highly discouraged at the time this document is being written and SHOULD NOT be used.

nonce: REQUIRED

A one-time-only value calculated by the server to be used in client calculations

cookie: OPTIONAL

This is the name of the HTTP-Cookie as defined by [RFC6265] which will be used by the server for continuing the session after initial HTTP based authentication has completed

message: OPTIONAL

A string value which MAY be presented to the user as additional information about the authentication challenge or previous attempt. An example would be "Unable to authenticate credentials at this time, please try again later."

opaque: OPTIONAL

A value calculated by the server to provide additional

protection against tampering and session management

path: OPTIONAL

A value representing the protected resource path. The provided path does not need to be the literal path but can be a symbolic path or hash which can be used by the client and server to limit a set of resources beyond what has been provided by the realm

version: OPTIONAL

The |JSON| Authentication Scheme version. This document defines this as 1.0 and version 1.0 MUST be assumed if no version is provided

window: OPTIONAL

An integer value (in seconds) the challenge will be valid for. This value is informational and MAY be used by the client to determine why a previous authentication attempt failed (challenge is no longer valid)

An example challenge for this type is:

```
{
   "type"       : "challenge"
  ,"algorithms" : "SHA-384,SHA-256,SHA-224"
  ,"nonce"      : "1488442706.13154/
                   339158aa-2504-44a4-bd7a-c86a85c4c7a8,
                   320afaed21f1827383194b49c02008909cf28
                   3ca2f3dca190c2ab958ea580a28"
}
```

NOTE: Elements in the object above have been expanded to multiple lines for readability purposes only

The Base64-encoded header with associated realm appears as:

```
WWW-Authenticate: |JSON| realm="Test Realm",
                  data="eyJ0eXBlIjoiY2hhbGxlbmdlIiwiYWxnb3JpdG
                        htcyI6IlNIQS0yNTYsU0hBLTEiLCJub25jZSI6
                        IjE0ODg0NDI3MDYuMTMxNTQvMzM5MTU4YWEtMj
                        UwNC00NGE0LWJkN2EtYzg2YTg1YzRjN2E4LDMy
                        MGFmYWVkMjFmMTgyNzM4MzE5NGI0OWMwMjAwOD
                        kwOWNmMjgzY2EyZjNkY2ExOTBjMmFiOTU4ZWE1
                        ODBhMjgifQ=="
```

The HTTP-Authentication response for this type includes the following elements (with definitions). More details are provided in the

implementation considerations sections below.

type: REQUIRED

The |JSON| Authentication Scheme type of "challenge"

username: REQUIRED

The user associated with this protected resource

algorithm: REQUIRED

The algorithm selected from the challenge's algorithms element. If no compatible algorithm can be established, the client MUST fail and if applicable provide additional details to the user

nonce: REQUIRED

This value MUST match the nonce value provided by the server's challenge verbatim

token: REQUIRED

This value MUST be generated by the client according to the following rules:

- A hash of the client provided password MUST be generated using the cryptographic algorithm specified as "algorithm" above.

```
password_hash = ALGORITHM ( password )
```

- The following values (including quoted colon values ":") MUST be concatenated into a single string value. Optional values which are not defined MUST be replaced by an empty string. Any required field which is not defined MUST fail and if applicable provide additional details to the user

```
pre_token = username + ":" + password_hash + ":" +
            nonce + ":" + opaque + ":" +
            algorithm + ":" + cnonce + ":" +
            message
```

- A hash of the string provided in the previous step MUST be generated using the cryptographic algorithm specified as "algorithm" above.

```
token = ALGORITHM ( pre_token )
```

NOTE: The output of the ALGORITHM function MUST be encoded as a lowercase hexadecimal value

NOTE: Upon receipt of a client's response to a "challenge" type server challenge, the server MUST follow the identical process as above to generate the token on its side of the conversation for an identical match. All necessary elements are expected to be already known, derived by known data or received in the client's response. Nonce, Opaque and other verifiable elements MUST be verified prior to recreating the token in order to verify data against potential tampering

cnonce: OPTIONAL

A one-time-only value calculated by the client to be used for further limiting replay-attacks

message: OPTIONAL

A string value which MAY be presented to the server as additional information about the authentication response for logging and debugging purposes. An example would be "CoolAuth-Client/1.0"

opaque: OPTIONAL

This value MUST match the opaque value provided by the server's challenge verbatim. If no opaque value was provided by the server's challenge, the client MUST NOT provide one in the response

version: OPTIONAL

The |JSON| Authentication Scheme version. This document defines this as 1.0 and version 1.0 MUST be assumed if no version is provided

For an example username of "MyUser", a password of "MyPassword" and the "SHA-256" hashing algorithm the JSON representation of the response would be:

```
{
   "type"       : "challenge"
  ,"algorithm"  : "SHA-256"
  ,"username"   : "MyUser"
  ,"nonce"      : "1488442706.13154/
                   339158aa-2504-44a4-bd7a-c86a85c4c7a8,
                   320afaed21f1827383194b49c02008909cf28
                   3ca2f3dca190c2ab958ea580a28"
  ,"token"      : "03066bdf1244be4c458fd6ef46af52acceea2
                   0d90ee979b10231018a52d92e66"
}
```

NOTE: Elements in the object above have been expanded to multiple

lines for readability purposes only

The Base64-encoded header with associated realm appears as:

```
Authorization: |JSON| realm="Test Realm",
               data="eyJ0eXBlIjoiY2hhbGxlbmdlIiwiYWxnb3JpdG
                     htIjoiU0hBLTI1NiIsInVzZXJuYW1lIjoiTXlV
                     c2VyIiwibm9uY2UiOiIxNDg4NDQyNzA2LjEzMT
                     U0LzMzOTE1OGFhLTI1MDQtNDRhNC1iZDdhLWM4
                     NmE4NWM0YzdhOCwzMjBhZmFlZDIxZjE4MjczOD
                     MxOTRiNDljMDIwMDg5MDljZjI4M2NhMmYzZGNh
                     MTkwYzJhYjk1OGVhNTgwYTI4IiwidG9rZW4iOi
                     IwMzA2NmJkZjEyNDRiZTRjNDU4ZmQ2ZWY0NmFm
                     NTJhY2NlZWEyMGQ5MGVlOTc5YjEwMjMxMDE4YT
                     UyZDkyZTY2In0="
```

NOTE: The "data" element has been expanded to multiple lines for readability purposes only

3.3. The Hybrid One-Off |JSON| Authentication Type Variant

This document defines a "One-Off" feature which informs the client's user agent the intent of the server's challenge is to fulfill a One-Time-Password (OTP) and client credentials MUST NOT be cached and reused for multiple responses.

It must be noted this is a deviation from the standard client behavior in that caching and reusing credentials expected as OTP can have the side-effect of locking the account upon reuse of successful credentials. It is highly recommended a cookie authentication mechanism be used to continue a successfully authenticated session. More details are provided in the implementation considerations sections below.

This feature is available for either the "password" or "challenge" types explained in the sections above and is enabled by setting the type to the type prepended by a single exclamation character (!).

For an example username of "MyUser" and password of "MyPassword" the JSON representation of the response would be:

```
{
   "type"     : "!password"
  ,"username" : "MyUser"
  ,"password" : "MyPassword"
}
```

4. Implementation Considerations

This document provides the following recommendations to improve

overall compatibility between implementations.

4.1 Nonce Generation

The authors of this document highly recommended two factors when implementing the generation of nonce values.

1) Time Component

The two main goals for the nonce field is to; a) provide protection against replays of captured credential payloads; and b) make cryptographic analysis more difficult. A simple implementation to meet both of these goals is to provide a time component to the nonce string.

The examples in this document use a hi-resolution "epoch" (or unix time) at the start of each nonce followed by a [US-ASCII] forward-slash character "/" and a generated portion.

> The string "1488442706.13154" from the examples above represents "Thu Mar 2 08:18:26 2017" referenced to the GMT timezone.

Using a component like this greatly reduces the risk of nonce reuse and allows for a validity window to easily be established. Since this is generated at the server, no time synchronization needs to be performed with the client and when used for the response, can be directly compared with the same time source.

2) Validation Component

Since the nonce is used as a component of the client's response and the HTTP protocol is stateless, it is highly recommended the validity of the nonce be confirmed prior to authenticating the request. If, for example, a malicious user was to modify the nonce and use this modified nonce in the response, the security of the requested resource may be jeopardized.

This document recommends adding a component which can be easily self-validated and offers protection against such tampering. This can be done by using a secret known only to the server (or set of servers).

The examples in this document use a Universally Unique Identifier UUID generator to provide a bit of uniqueness beyond simply the time component discussed above. Most UUID implementations leverage a source of entropy to nearly eliminate the risk of collisions and this along with the time component should make this safe within a reasonable validation window.

The UUID is combined with the time component, optional opaque value and secret key to form the full nonce string using the following pattern:

```
time_component + ":" + uuid_component + ":" +
                 opaque + ":" + secret
```

The examples in this document use the following values for this:

```
time_component = "1488442706.13154"

uuid_component = "339158aa-2504-44a4-bd7a-c86a85c4c7a8"

opaque         = ""

secret         = "MyKey"
```

This equates to the string:

```
1488442706.13154:339158aa-2504-44a4-bd7a-c86a85c4c7a8::MyKey
```

The SHA-256 output of this is:

```
320afaed21f1827383194b49c02008909cf283ca2f3dca19
0c2ab958ea580a28
```

NOTE: The above example has been expanded to multiple lines for readability purposes only

The final string for this nonce is:

```
1488442706.13154/339158aa-2504-44a4-bd7a-c86a85c4c7a8,
320afaed21f1827383194b49c02008909cf283ca2f3dca190c2ab
958ea580a28
```

NOTE: The above example has been expanded to multiple lines for readability purposes only

4.2 User-Agent Scripted Authentication

A highlighted feature of this document is its ability to pass authentication to a scripting facility within the client. This section provides recommendations for implementing this feature.

1) Chicken and Egg avoidance:

A complication to tying scripts to authentication is the scripts must be loaded prior to the authentication challenge in order for this to work. One way to avoid this is to provide

the authentication handler in a script above other scripts in

the main document. The main document will need to be able to load without initial authentication for this to work. This solution will not work where Asynchronous Module Definition (AMD) is deployed.

Another solution would be to use an authentication entry-point page which would be in an unauthenticated "zone" of the application and once successfully authenticated redirect to the authenticated "zone" leveraging cookies for the transition.

2) Where do the Scripted Handlers Live?

Assuming this document has found a large adoption, it is recommended the user agent hosts this as a function of the root of in its HTML DOM tree (e.g. "window.AuthHandler();").

This function could provide:

a) A test for the existence of an EAH handler for a particular scheme

b) Invoke an EAH handler for a particular

c) Install an EAH handler for a particular scheme

Until interest for such an adoption, we recommended this to be where developers host their EAHs and invoke such EAHs where appropriate.

5. Security Considerations

This document provides methods for transmitting credentials (or the implicit knowledge of these credentials) from an HTTP client agent (e.g. "browser") to an HTTP server. Methods have been provided in some instances to protect the actual credentials from tampering between this connection but steps must be taken on each side to protect the credential collection and validation points. Any weak point in a security system makes that system in its entirety just as weak.

6. IANA Considerations

IANA is requested to update their http-authschemes registry <http://www.iana.org/assignments/http-authschemes> to include the "|JSON|" and "|*|" schemes where "|*|" represents any valid token surrounded by [US-ASCII] pipe characters "|" (e.g. "|pdmk|").

7. Acknowledgments

This document leans heavily on on the work of others, specifically

those responsible for the RFCs listed below. The authors of this document wish to thank each author involved to help get us here. The authors also extend a special thanks to Kathleen Moriarty, for encouraging us to finish this draft.

8. References

8.1. Normative References

[US-ASCII] American National Standards Institute, "Coded Character Set -- 7-bit American Standard Code for Information Interchange", ANSI X3.4, 1986.

[FIPS-180-4] National Institute of Standards and Technology (NIST), United States of America, "Secure Hash Standard (SHS)", FIPS PUB 180-4, August 2015 , <http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf>.

[FIPS-202] National Institute of Standards and Technology (NIST), United States of America, "SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions", FIPS PUB 202, August 2015 , <http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf>.

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

[RFC4648] S. Josefsson, SJD, "The Base16, Base32, and Base64 Data Encodings", RFC 4648, October 2006.

[RFC6265] A. Barth, "HTTP State Management Mechanism", RFC 6265, April 2011.

[RFC7159] T. Bray, "The JavaScript Object Notation (JSON) Data Interchange Format", RFC 7159, March 2014.

[RFC7230] R. Fielding, J. Reschke, "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", RFC 7230, June 2014.

[RFC7235] R. Fielding, J. Reschke, "Hypertext Transfer Protocol (HTTP/1.1): Authentication", RFC 7235, June 2014.

8.2. Informative References

Authors' Addresses

John Woodworth
4250 North Fairfax Drive
Arlington, VA 22203
USA

EMail: John.Woodworth@CenturyLink.com

Dean Ballew
2355 Dulles Corner Boulevard Suite 200 300
Herndon, VA 20171
USA

EMail: Dean.Ballew@CenturyLink.com

Acknowledgement

Funding for the RFC Editor function is currently provided by the Internet Society.