

## Using Digest Authentication as a SASL Mechanism

### Status of this Memo

This document specifies an Internet standards track protocol for the Internet community, and requests discussion and suggestions for improvements. Please refer to the current edition of the "Internet Official Protocol Standards" (STD 1) for the standardization state and status of this protocol. Distribution of this memo is unlimited.

### Copyright Notice

Copyright (C) The Internet Society (2000). All Rights Reserved.

### Abstract

This specification defines how HTTP Digest Authentication [Digest] can be used as a SASL [RFC 2222] mechanism for any protocol that has a SASL profile. It is intended both as an improvement over CRAM-MD5 [RFC 2195] and as a convenient way to support a single authentication mechanism for web, mail, LDAP, and other protocols.

### Table of Contents

1	INTRODUCTION.....	2
1.1	CONVENTIONS AND NOTATION.....	2
1.2	REQUIREMENTS.....	3
2	AUTHENTICATION.....	3
2.1	INITIAL AUTHENTICATION.....	3
2.1.1	Step One.....	3
2.1.2	Step Two.....	6
2.1.3	Step Three.....	12
2.2	SUBSEQUENT AUTHENTICATION.....	12
2.2.1	Step one.....	13
2.2.2	Step Two.....	13
2.3	INTEGRITY PROTECTION.....	13
2.4	CONFIDENTIALITY PROTECTION.....	14
3	SECURITY CONSIDERATIONS.....	15
3.1	AUTHENTICATION OF CLIENTS USING DIGEST AUTHENTICATION.....	15
3.2	COMPARISON OF DIGEST WITH PLAINTEXT PASSWORDS.....	16
3.3	REPLAY ATTACKS.....	16

3.4	ONLINE DICTIONARY ATTACKS.....	16
3.5	OFFLINE DICTIONARY ATTACKS.....	16
3.6	MAN IN THE MIDDLE.....	17
3.7	CHOSEN PLAINTEXT ATTACKS.....	17
3.8	SPOOFING BY COUNTERFEIT SERVERS.....	17
3.9	STORING PASSWORDS.....	17
3.10	MULTIPLE REALMS.....	18
3.11	SUMMARY.....	18
4	EXAMPLE.....	18
5	REFERENCES.....	20
6	AUTHORS' ADDRESSES.....	21
7	ABNF.....	21
7.1	AUGMENTED BNF.....	21
7.2	BASIC RULES.....	23
8	SAMPLE CODE.....	25
9	FULL COPYRIGHT STATEMENT.....	27

## 1 Introduction

This specification describes the use of HTTP Digest Access Authentication as a SASL mechanism. The authentication type associated with the Digest SASL mechanism is "DIGEST-MD5".

This specification is intended to be upward compatible with the "md5-sess" algorithm of HTTP/1.1 Digest Access Authentication specified in [Digest]. The only difference in the "md5-sess" algorithm is that some directives not needed in a SASL mechanism have had their values defaulted.

There is one new feature for use as a SASL mechanism: integrity protection on application protocol messages after an authentication exchange.

Also, compared to CRAM-MD5, DIGEST-MD5 prevents chosen plaintext attacks, and permits the use of third party authentication servers, mutual authentication, and optimized reauthentication if a client has recently authenticated to a server.

### 1.1 Conventions and Notation

This specification uses the same ABNF notation and lexical conventions as HTTP/1.1 specification; see appendix A.

Let { a, b, ... } be the concatenation of the octet strings a, b, ...

Let H(s) be the 16 octet MD5 hash [RFC 1321] of the octet string s.

Let  $KD(k, s)$  be  $H(\{k, ":", s\})$ , i.e., the 16 octet hash of the string  $k$ , a colon and the string  $s$ .

Let  $HEX(n)$  be the representation of the 16 octet MD5 hash  $n$  as a string of 32 hex digits (with alphabetic characters always in lower case, since MD5 is case sensitive).

Let  $HMAC(k, s)$  be the 16 octet HMAC-MD5 [RFC 2104] of the octet string  $s$  using the octet string  $k$  as a key.

The value of a quoted string constant as an octet string does not include any terminating null character.

## 1.2 Requirements

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 RFC 2119 [RFC 2119].

An implementation is not compliant if it fails to satisfy one or more of the MUST level requirements for the protocols it implements. An implementation that satisfies all the MUST level and all the SHOULD level requirements for its protocols is said to be "unconditionally compliant"; one that satisfies all the MUST level requirements but not all the SHOULD level requirements for its protocols is said to be "conditionally compliant."

## 2 Authentication

The following sections describe how to use Digest as a SASL authentication mechanism.

### 2.1 Initial Authentication

If the client has not recently authenticated to the server, then it must perform "initial authentication", as defined in this section. If it has recently authenticated, then a more efficient form is available, defined in the next section.

#### 2.1.1 Step One

The server starts by sending a challenge. The data encoded in the challenge contains a string formatted according to the rules for a "digest-challenge" defined as follows:

```

digest-challenge =
    1#( realm | nonce | qop-options | stale | maxbuf | charset
        algorithm | cipher-opts | auth-param )

realm            = "realm" "=" <"> realm-value <">
realm-value      = qdstr-val
nonce            = "nonce" "=" <"> nonce-value <">
nonce-value      = qdstr-val
qop-options       = "qop" "=" <"> qop-list <">
qop-list         = 1#qop-value
qop-value        = "auth" | "auth-int" | "auth-conf" |
                    token
stale             = "stale" "=" "true"
maxbuf           = "maxbuf" "=" maxbuf-value
maxbuf-value      = 1*DIGIT
charset          = "charset" "=" "utf-8"
algorithm        = "algorithm" "=" "md5-sess"
cipher-opts       = "cipher" "=" <"> 1#cipher-value <">
cipher-value      = "3des" | "des" | "rc4-40" | "rc4" |
                    "rc4-56" | token
auth-param        = token "=" ( token | quoted-string )

```

The meanings of the values of the directives used above are as follows:

#### realm

Mechanistically, a string which can enable users to know which username and password to use, in case they might have different ones for different servers. Conceptually, it is the name of a collection of accounts that might include the user's account. This string should contain at least the name of the host performing the authentication and might additionally indicate the collection of users who might have access. An example might be "registered\_users@gotham.news.example.com". This directive is optional; if not present, the client SHOULD solicit it from the user or be able to compute a default; a plausible default might be the realm supplied by the user when they logged in to the client system. Multiple realm directives are allowed, in which case the user or client must choose one as the realm for which to supply to username and password.

#### nonce

A server-specified data string which MUST be different each time a digest-challenge is sent as part of initial authentication. It is recommended that this string be base64 or hexadecimal data. Note that since the string is passed as a quoted string, the double-quote character is not allowed unless escaped (see section 7.2). The contents of the nonce are implementation dependent. The

security of the implementation depends on a good choice. It is RECOMMENDED that it contain at least 64 bits of entropy. The nonce is opaque to the client. This directive is required and MUST appear exactly once; if not present, or if multiple instances are present, the client should abort the authentication exchange.

#### qop-options

A quoted string of one or more tokens indicating the "quality of protection" values supported by the server. The value "auth" indicates authentication; the value "auth-int" indicates authentication with integrity protection; the value "auth-conf" indicates authentication with integrity protection and encryption. This directive is optional; if not present it defaults to "auth". The client MUST ignore unrecognized options; if the client recognizes no option, it should abort the authentication exchange.

#### stale

The "stale" directive is not used in initial authentication. See the next section for its use in subsequent authentications. This directive may appear at most once; if multiple instances are present, the client should abort the authentication exchange.

#### maxbuf

A number indicating the size of the largest buffer the server is able to receive when using "auth-int" or "auth-conf". If this directive is missing, the default value is 65536. This directive may appear at most once; if multiple instances are present, the client should abort the authentication exchange.

#### charset

This directive, if present, specifies that the server supports UTF-8 encoding for the username and password. If not present, the username and password must be encoded in ISO 8859-1 (of which US-ASCII is a subset). The directive is needed for backwards compatibility with HTTP Digest, which only supports ISO 8859-1. This directive may appear at most once; if multiple instances are present, the client should abort the authentication exchange.

#### algorithm

This directive is required for backwards compatibility with HTTP Digest., which supports other algorithms. . This directive is required and MUST appear exactly once; if not present, or if multiple instances are present, the client should abort the authentication exchange.

#### cipher-opts

A list of ciphers that the server supports. This directive must be present exactly once if "auth-conf" is offered in the "qop-options" directive, in which case the "3des" and "des" modes are mandatory-to-implement. The client MUST ignore unrecognized options; if the client recognizes no option, it should abort the authentication exchange.

#### des

the Data Encryption Standard (DES) cipher [FIPS] in cipher block chaining (CBC) mode with a 56 bit key.

#### 3des

the "triple DES" cipher in CBC mode with EDE with the same key for each E stage (aka "two keys mode") for a total key length of 112 bits.

#### rc4, rc4-40, rc4-56

the RC4 cipher with a 128 bit, 40 bit, and 56 bit key, respectively.

**auth-param** This construct allows for future extensions; it may appear more than once. The client MUST ignore any unrecognized directives.

For use as a SASL mechanism, note that the following changes are made to "digest-challenge" from HTTP: the following Digest options (called "directives" in HTTP terminology) are unused (i.e., MUST NOT be sent, and MUST be ignored if received):

opaque  
domain

The size of a digest-challenge MUST be less than 2048 bytes.

#### 2.1.2 Step Two

The client makes note of the "digest-challenge" and then responds with a string formatted and computed according to the rules for a "digest-response" defined as follows:

```

digest-response = 1#( username | realm | nonce | cnonce |
                        nonce-count | qop | digest-uri | response |
                        maxbuf | charset | cipher | authzid |
                        auth-param )

username         = "username" "=" <"> username-value <">
username-value   = qdstr-val
cnonce           = "cnonce" "=" <"> cnonce-value <">
cnonce-value     = qdstr-val
nonce-count      = "nc" "=" nc-value
nc-value         = 8LHEX
qop              = "qop" "=" qop-value
digest-uri       = "digest-uri" "=" <"> digest-uri-value <">
digest-uri-value = serv-type "/" host [ "/" serv-name ]
serv-type        = 1*ALPHA
host             = 1*( ALPHA | DIGIT | "-" | "." )
serv-name        = host
response         = "response" "=" response-value
response-value   = 32LHEX
LHEX             = "0" | "1" | "2" | "3" |
                  "4" | "5" | "6" | "7" |
                  "8" | "9" | "a" | "b" |
                  "c" | "d" | "e" | "f"
cipher           = "cipher" "=" cipher-value
authzid          = "authzid" "=" <"> authzid-value <">
authzid-value    = qdstr-val

```

#### username

The user's name in the specified realm, encoded according to the value of the "charset" directive. This directive is required and MUST be present exactly once; otherwise, authentication fails.

#### realm

The realm containing the user's account. This directive is required if the server provided any realms in the "digest-challenge", in which case it may appear exactly once and its value SHOULD be one of those realms. If the directive is missing, "realm-value" will set to the empty string when computing A1 (see below for details).

#### nonce

The server-specified data string received in the preceding digest-challenge. This directive is required and MUST be present exactly once; otherwise, authentication fails.

**cnonce**

A client-specified data string which MUST be different each time a digest-response is sent as part of initial authentication. The cnonce-value is an opaque quoted string value provided by the client and used by both client and server to avoid chosen plaintext attacks, and to provide mutual authentication. The security of the implementation depends on a good choice. It is RECOMMENDED that it contain at least 64 bits of entropy. This directive is required and MUST be present exactly once; otherwise, authentication fails.

**nonce-count**

The nc-value is the hexadecimal count of the number of requests (including the current request) that the client has sent with the nonce value in this request. For example, in the first request sent in response to a given nonce value, the client sends "nc=00000001". The purpose of this directive is to allow the server to detect request replays by maintaining its own copy of this count - if the same nc-value is seen twice, then the request is a replay. See the description below of the construction of the response value. This directive may appear at most once; if multiple instances are present, the client should abort the authentication exchange.

**qop**

Indicates what "quality of protection" the client accepted. If present, it may appear exactly once and its value MUST be one of the alternatives in qop-options. If not present, it defaults to "auth". These values affect the computation of the response. Note that this is a single token, not a quoted list of alternatives.

**serv-type**

Indicates the type of service, such as "www" for web service, "ftp" for FTP service, "smtp" for mail delivery service, etc. The service name as defined in the SASL profile for the protocol see section 4 of [RFC 2222], registered in the IANA registry of "service" elements for the GSSAPI host-based service name form [RFC 2078].

**host**

The DNS host name or IP address for the service requested. The DNS host name must be the fully-qualified canonical name of the host. The DNS host name is the preferred form; see notes on server processing of the digest-uri.



**serv-name**

Indicates the name of the service if it is replicated. The service is considered to be replicated if the client's service-location process involves resolution using standard DNS lookup operations, and if these operations involve DNS records (such as SRV, or MX) which resolve one DNS name into a set of other DNS names. In this case, the initial name used by the client is the "serv-name", and the final name is the "host" component. For example, the incoming mail service for "example.com" may be replicated through the use of MX records stored in the DNS, one of which points at an SMTP server called "mail3.example.com"; it's "serv-name" would be "example.com", it's "host" would be "mail3.example.com". If the service is not replicated, or the serv-name is identical to the host, then the serv-name component **MUST** be omitted.

**digest-uri**

Indicates the principal name of the service with which the client wishes to connect, formed from the serv-type, host, and serv-name. For example, the FTP service on "ftp.example.com" would have a "digest-uri" value of "ftp/ftp.example.com"; the SMTP server from the example above would have a "digest-uri" value of "smtp/mail3.example.com/example.com".

Servers **SHOULD** check that the supplied value is correct. This will detect accidental connection to the incorrect server. It is also so that clients will be trained to provide values that will work with implementations that use a shared back-end authentication service that can provide server authentication.

The serv-type component should match the service being offered. The host component should match one of the host names of the host on which the service is running, or it's IP address. Servers **SHOULD NOT** normally support the IP address form, because server authentication by IP address is not very useful; they should only do so if the DNS is unavailable or unreliable. The serv-name component should match one of the service's configured service names.

This directive may appear at most once; if multiple instances are present, the client should abort the authentication exchange.

Note: In the HTTP use of Digest authentication, the digest-uri is the URI (usually a URL) of the resource requested -- hence the name of the directive.

**response**

A string of 32 hex digits computed as defined below, which proves that the user knows a password. This directive is required and **MUST** be present exactly once; otherwise, authentication fails.

**maxbuf**

A number indicating the size of the largest buffer the client is able to receive. If this directive is missing, the default value is 65536. This directive may appear at most once; if multiple instances are present, the server should abort the authentication exchange.

**charset**

This directive, if present, specifies that the client has used UTF-8 encoding for the username and password. If not present, the username and password must be encoded in ISO 8859-1 (of which US-ASCII is a subset). The client should send this directive only if the server has indicated it supports UTF-8. The directive is needed for backwards compatibility with HTTP Digest, which only supports ISO 8859-1.

**LHEX**

32 hex digits, where the alphabetic characters MUST be lower case, because MD5 is not case insensitive.

**cipher**

The cipher chosen by the client. This directive MUST appear exactly once if "auth-conf" is negotiated; if required and not present, authentication fails.

**authzid**

The "authorization ID" as per RFC 2222, encoded in UTF-8. This directive is optional. If present, and the authenticating user has sufficient privilege, and the server supports it, then after authentication the server will use this identity for making all accesses and access checks. If the client specifies it, and the server does not support it, then the response-value will be incorrect, and authentication will fail.

The size of a digest-response MUST be less than 4096 bytes.

**2.1.2.1 Response-value**

The definition of "response-value" above indicates the encoding for its value -- 32 lower case hex characters. The following definitions show how the value is computed.

Although qop-value and components of digest-uri-value may be case-insensitive, the case which the client supplies in step two is preserved for the purpose of computing and verifying the response-value.

response-value =

```

    HEX( KD ( HEX(H(A1)),
              { nonce-value, ":" nc-value, ":",
                cnonce-value, ":", qop-value, ":", HEX(H(A2)) } ))

```

If authzid is specified, then A1 is

```

A1 = { H( { username-value, ":", realm-value, ":", passwd } ),
      ":", nonce-value, ":", cnonce-value, ":", authzid-value }

```

If authzid is not specified, then A1 is

```

A1 = { H( { username-value, ":", realm-value, ":", passwd } ),
      ":", nonce-value, ":", cnonce-value }

```

where

```

passwd = *OCTET

```

The "username-value", "realm-value" and "passwd" are encoded according to the value of the "charset" directive. If "charset=UTF-8" is present, and all the characters of either "username-value" or "passwd" are in the ISO 8859-1 character set, then it must be converted to ISO 8859-1 before being hashed. This is so that authentication databases that store the hashed username, realm and password (which is common) can be shared compatibly with HTTP, which specifies ISO 8859-1. A sample implementation of this conversion is in section 8.

If the "qop" directive's value is "auth", then A2 is:

```

A2 = { "AUTHENTICATE:", digest-uri-value }

```

If the "qop" value is "auth-int" or "auth-conf" then A2 is:

```

A2 = { "AUTHENTICATE:", digest-uri-value,
      ":00000000000000000000000000000000" }

```

Note that "AUTHENTICATE:" must be in upper case, and the second string constant is a string with a colon followed by 32 zeros.

These apparently strange values of A2 are for compatibility with HTTP; they were arrived at by setting "Method" to "AUTHENTICATE" and the hash of the entity body to zero in the HTTP digest calculation of A2.

Also, in the HTTP usage of Digest, several directives in the

"digest-challenge" sent by the server have to be returned by the client in the "digest-response". These are:

```
opaque
algorithm
```

These directives are not needed when Digest is used as a SASL mechanism (i.e., MUST NOT be sent, and MUST be ignored if received).

### 2.1.3 Step Three

The server receives and validates the "digest-response". The server checks that the nonce-count is "00000001". If it supports subsequent authentication (see section 2.2), it saves the value of the nonce and the nonce-count. It sends a message formatted as follows:

```
response-auth = "rspauth" "=" response-value
```

where response-value is calculated as above, using the values sent in step two, except that if qop is "auth", then A2 is

```
A2 = { ":", digest-uri-value }
```

And if qop is "auth-int" or "auth-conf" then A2 is

```
A2 = { ":", digest-uri-value, ":00000000000000000000000000000000" }
```

Compared to its use in HTTP, the following Digest directives in the "digest-response" are unused:

```
nextnonce
qop
cnonce
nonce-count
```

## 2.2 Subsequent Authentication

If the client has previously authenticated to the server, and remembers the values of username, realm, nonce, nonce-count, cnonce, and qop that it used in that authentication, and the SASL profile for a protocol permits an initial client response, then it MAY perform "subsequent authentication", as defined in this section.

### 2.2.1 Step one

The client uses the values from the previous authentication and sends an initial response with a string formatted and computed according to the rules for a "digest-response", as defined above, but with a nonce-count one greater than used in the last "digest-response".

### 2.2.2 Step Two

The server receives the "digest-response". If the server does not support subsequent authentication, then it sends a "digest-challenge", and authentication proceeds as in initial authentication. If the server has no saved nonce and nonce-count from a previous authentication, then it sends a "digest-challenge", and authentication proceeds as in initial authentication. Otherwise, the server validates the "digest-response", checks that the nonce-count is one greater than that used in the previous authentication using that nonce, and saves the new value of nonce-count.

If the response is invalid, then the server sends a "digest-challenge", and authentication proceeds as in initial authentication (and should be configurable to log an authentication failure in some sort of security audit log, since the failure may be a symptom of an attack). The nonce-count MUST NOT be incremented in this case: to do so would allow a denial of service attack by sending an out-of-order nonce-count.

If the response is valid, the server MAY choose to deem that authentication has succeeded. However, if it has been too long since the previous authentication, or for any other reason, the server MAY send a new "digest-challenge" with a new value for nonce. The challenge MAY contain a "stale" directive with value "true", which says that the client may respond to the challenge using the password it used in the previous response; otherwise, the client must solicit the password anew from the user. This permits the server to make sure that the user has presented their password recently. (The directive name refers to the previous nonce being stale, not to the last use of the password.) Except for the handling of "stale", after sending the "digest-challenge" authentication proceeds as in the case of initial authentication.

## 2.3 Integrity Protection

If the server offered "qop=auth-int" and the client responded "qop=auth-int", then subsequent messages, up to but not including the next subsequent authentication, between the client and the server

MUST be integrity protected. Using as a base session key the value of  $H(A1)$  as defined above the client and server calculate a pair of message integrity keys as follows.

The key for integrity protecting messages from client to server is:

```
Kic = MD5({H(A1),  
"Digest session key to client-to-server signing key magic constant"})
```

The key for integrity protecting messages from server to client is:

```
Kis = MD5({H(A1),  
"Digest session key to server-to-client signing key magic constant"})
```

where MD5 is as specified in [RFC 1321]. If message integrity is negotiated, a MAC block for each message is appended to the message. The MAC block is 16 bytes: the first 10 bytes of the HMAC-MD5 [RFC 2104] of the message, a 2-byte message type number in network byte order with value 1, and the 4-byte sequence number in network byte order. The message type is to allow for future extensions such as rekeying.

```
MAC(Ki, SeqNum, msg) = (HMAC(Ki, {SeqNum, msg})[0..9], 0x0001,  
SeqNum)
```

where  $K_i$  is  $K_{ic}$  for messages sent by the client and  $K_{is}$  for those sent by the server. The sequence number is initialized to zero, and incremented by one for each message sent.

Upon receipt,  $MAC(K_i, SeqNum, msg)$  is computed and compared with the received value; the message is discarded if they differ.

## 2.4 Confidentiality Protection

If the server sent a "cipher-opts" directive and the client responded with a "cipher" directive, then subsequent messages between the client and the server MUST be confidentiality protected. Using as a base session key the value of  $H(A1)$  as defined above the client and server calculate a pair of message integrity keys as follows.

The key for confidentiality protecting messages from client to server is:

```
Kcc = MD5({H(A1)[0..n],  
"Digest H(A1) to client-to-server sealing key magic constant"})
```

The key for confidentiality protecting messages from server to client is:

```
Kcs = MD5({H(A1)[0..n],  
"Digest H(A1) to server-to-client sealing key magic constant"})
```

where MD5 is as specified in [RFC 1321]. For cipher "rc4-40" n is 5; for "rc4-56" n is 7; for the rest n is 16. The key for the "rc-\*" ciphers is all 16 bytes of Kcc or Kcs; the key for "des" is the first 7 bytes; the key for "3des" is the first 14 bytes. The IV for "des" and "3des" is the last 8 bytes of Kcc or Kcs.

If message confidentiality is negotiated, each message is encrypted with the chosen cipher and a MAC block is appended to the message.

The MAC block is a variable length padding prefix followed by 16 bytes formatted as follows: the first 10 bytes of the HMAC-MD5 [RFC 2104] of the message, a 2-byte message type number in network byte order with value 1, and the 4-byte sequence number in network byte order. If the blocksize of the chosen cipher is not 1 byte, the padding prefix is one or more octets each containing the number of padding bytes, such that total length of the encrypted part of the message is a multiple of the blocksize. The padding and first 10 bytes of the MAC block are encrypted along with the message.

```
SEAL(Ki, Kc, SeqNum, msg) =  
    {CIPHER(Kc, {msg, pad, HMAC(Ki, {SeqNum, msg})[0..9])}), 0x0001,  
    SeqNum}
```

where CIPHER is the chosen cipher, Ki and Kc are Kic and Kcc for messages sent by the client and Kis and Kcs for those sent by the server. The sequence number is initialized to zero, and incremented by one for each message sent.

Upon receipt, the message is decrypted, HMAC(Ki, {SeqNum, msg}) is computed and compared with the received value; the message is discarded if they differ.

### 3 Security Considerations

#### 3.1 Authentication of Clients using Digest Authentication

Digest Authentication does not provide a strong authentication mechanism, when compared to public key based mechanisms, for example. However, since it prevents chosen plaintext attacks, it is stronger than (e.g.) CRAM-MD5, which has been proposed for use with LDAP [10], POP and IMAP (see RFC 2195 [9]). It is intended to replace the much weaker and even more dangerous use of plaintext passwords; however, since it is still a password based mechanism it avoids some of the potential deployability issues with public-key, OTP or similar mechanisms.

Digest Authentication offers no confidentiality protection beyond protecting the actual password. All of the rest of the challenge and response are available to an eavesdropper, including the user's name and authentication realm.

### 3.2 Comparison of Digest with Plaintext Passwords

The greatest threat to the type of transactions for which these protocols are used is network snooping. This kind of transaction might involve, for example, online access to a mail service whose use is restricted to paying subscribers. With plaintext password authentication an eavesdropper can obtain the password of the user. This not only permits him to access anything in the database, but, often worse, will permit access to anything else the user protects with the same password.

### 3.3 Replay Attacks

Replay attacks are defeated if the client or the server chooses a fresh nonce for each authentication, as this specification requires.

### 3.4 Online dictionary attacks

If the attacker can eavesdrop, then it can test any overheard nonce/response pairs against a (potentially very large) list of common words. Such a list is usually much smaller than the total number of possible passwords. The cost of computing the response for each password on the list is paid once for each challenge.

The server can mitigate this attack by not allowing users to select passwords that are in a dictionary.

### 3.5 Offline dictionary attacks

If the attacker can choose the challenge, then it can precompute the possible responses to that challenge for a list of common words. Such a list is usually much smaller than the total number of possible passwords. The cost of computing the response for each password on the list is paid just once.

Offline dictionary attacks are defeated if the client chooses a fresh nonce for each authentication, as this specification requires.



### 3.6 Man in the Middle

Digest authentication is vulnerable to "man in the middle" (MITM) attacks. Clearly, a MITM would present all the problems of eavesdropping. But it also offers some additional opportunities to the attacker.

A possible man-in-the-middle attack would be to substitute a weaker qop scheme for the one(s) sent by the server; the server will not be able to detect this attack. For this reason, the client should always use the strongest scheme that it understands from the choices offered, and should never choose a scheme that does not meet its minimum requirements.

### 3.7 Chosen plaintext attacks

A chosen plaintext attack is where a MITM or a malicious server can arbitrarily choose the challenge that the client will use to compute the response. The ability to choose the challenge is known to make cryptanalysis much easier [8].

However, Digest does not permit the attack to choose the challenge as long as the client chooses a fresh nonce for each authentication, as this specification requires.

### 3.8 Spoofing by Counterfeit Servers

If a user can be led to believe that she is connecting to a host containing information protected by a password she knows, when in fact she is connecting to a hostile server, then the hostile server can obtain challenge/response pairs where it was able to partly choose the challenge. There is no known way that this can be exploited.

### 3.9 Storing passwords

Digest authentication requires that the authenticating agent (usually the server) store some data derived from the user's name and password in a "password file" associated with a given realm. Normally this might contain pairs consisting of username and  $H(\{ \text{username-value}, \text{":"}, \text{realm-value}, \text{":"}, \text{passwd} \})$ , which is adequate to compute  $H(A1)$  as described above without directly exposing the user's password.

The security implications of this are that if this password file is compromised, then an attacker gains immediate access to documents on the server using this realm. Unlike, say a standard UNIX password file, this information need not be decrypted in order to access documents in the server realm associated with this file. On the other

hand, decryption, or more likely a brute force attack, would be necessary to obtain the user's password. This is the reason that the realm is part of the digested data stored in the password file. It means that if one Digest authentication password file is compromised, it does not automatically compromise others with the same username and password (though it does expose them to brute force attack).

There are two important security consequences of this. First the password file must be protected as if it contained plaintext passwords, because for the purpose of accessing documents in its realm, it effectively does.

A second consequence of this is that the realm string should be unique among all realms that any single user is likely to use. In particular a realm string should include the name of the host doing the authentication.

### 3.10 Multiple realms

Use of multiple realms may mean both that compromise of a the security database for a single realm does not compromise all security, and that there are more things to protect in order to keep the whole system secure.

### 3.11 Summary

By modern cryptographic standards Digest Authentication is weak, compared to (say) public key based mechanisms. But for a large range of purposes it is valuable as a replacement for plaintext passwords. Its strength may vary depending on the implementation.

## 4 Example

This example shows the use of the Digest SASL mechanism with the IMAP4 AUTHENTICATE command [RFC 2060].

In this example, "C:" and "S:" represent a line sent by the client or server respectively including a CRLF at the end. Linebreaks and indentation within a "C:" or "S:" are editorial and not part of the protocol. The password in this example was "secret". Note that the base64 encoding of the challenges and responses is part of the IMAP4 AUTHENTICATE command, not part of the Digest specification itself.

```
S: * OK elwood.innosoft.com PMDF IMAP4rev1 V6.0-9
C: c CAPABILITY
S: * CAPABILITY IMAP4 IMAP4rev1 ACL LITERAL+ NAMESPACE QUOTA
    UIDPLUS AUTH=CRAM-MD5 AUTH=DIGEST-MD5 AUTH=PLAIN
S: c OK Completed
```

```

C: a AUTHENTICATE DIGEST-MD5
S: + cmVhbG09ImVsd29vZC5pbm5vc29mdC5jb20iLG5vbmNlPSJPQTZNRz10
    RVFHBtJoaCIscW9wPSJhdXRoIixhbGdvcml0aG09bWQ1LXNlc3MsY2hh
    cnNldD1ldGYtOA==
C: Y2hhcnNldD1ldGYtOCxlc2VybmFtZT0iY2hyaXMiLHJlYWxtPSJlbHdvb2
    QuaW5ub3NvZnQuY29tIixub25jZT0iT0E2TUC5dEVRR20yaGgiLG5jPTAw
    MDAwMDAxLGNub25jZT0iT0E2TUhYaDZwcVRyUmsiLGRpZ2VzdC11cmk9Im
    ltYXAuZWx3b29kLmlubm9zb2Z0LmNvbSIscmVzcG9uc2U9ZDM4OGRhZDkw
    ZDRiYmQ3NjBhMTUyMzIxZjIxNDNhZjcscW9wPWFldGg=
S: + cnNwYXV0aD1lYTQwZjYwMzM1YzQyN2I1NTI3Yjg0ZGJhYmNkZmZmZA==
C:
S: a OK User logged in
---
```

The base64-decoded version of the SASL exchange is:

```

S: realm="elwood.innosoft.com",nonce="OA6MG9tEQGm2hh",qop="auth",
    algorithm=md5-sess,charset=utf-8
C: charset=utf-8,username="chris",realm="elwood.innosoft.com",
    nonce="OA6MG9tEQGm2hh",nc=00000001,cnonce="OA6MHXh6VqTrRk",
    digest-uri="imap/elwood.innosoft.com",
    response=d388dad90d4bbd760a152321f2143af7,qop=auth
S: rspauth=ea40f60335c427b5527b84dbabdcfffd
```

The password in this example was "secret".

This example shows the use of the Digest SASL mechanism with the ACAP, using the same notational conventions and password as in the previous example. Note that ACAP does not base64 encode and uses fewer round trips than IMAP4.

```

S: * ACAP (IMPLEMENTATION "Test ACAP server") (SASL "CRAM-MD5"
    "DIGEST-MD5" "PLAIN")
C: a AUTHENTICATE "DIGEST-MD5"
S: + {94}
S: realm="elwood.innosoft.com",nonce="OA9BSXrbuRhWay",qop="auth",
    algorithm=md5-sess,charset=utf-8
C: {206}
C: charset=utf-8,username="chris",realm="elwood.innosoft.com",
    nonce="OA9BSXrbuRhWay",nc=00000001,cnonce="OA9BSuZWMSpW8m",
    digest-uri="acap/elwood.innosoft.com",
    response=6084c6db3fede7352c551284490fd0fc,qop=auth
S: a OK (SASL {40})
S: rspauth=2f0b3d7c3c2e486600ef710726aa2eae) "AUTHENTICATE
    Completed"
---
```

The server uses the values of all the directives, plus knowledge of the users password (or the hash of the user's name, server's realm and the user's password) to verify the computations above. If they check, then the user has authenticated.

## 5 References

- [Digest] Franks, J., et al., "HTTP Authentication: Basic and Digest Access Authentication", RFC 2617, June 1999.
- [ISO-8859] ISO-8859. International Standard--Information Processing--8-bit Single-Byte Coded Graphic Character Sets --
  - Part 1: Latin alphabet No. 1, ISO-8859-1:1987.
  - Part 2: Latin alphabet No. 2, ISO-8859-2, 1987.
  - Part 3: Latin alphabet No. 3, ISO-8859-3, 1988.
  - Part 4: Latin alphabet No. 4, ISO-8859-4, 1988.
  - Part 5: Latin/Cyrillic alphabet, ISO-8859-5, 1988.
  - Part 6: Latin/Arabic alphabet, ISO-8859-6, 1987.
  - Part 7: Latin/Greek alphabet, ISO-8859-7, 1987.
  - Part 8: Latin/Hebrew alphabet, ISO-8859-8, 1988.
  - Part 9: Latin alphabet No. 5, ISO-8859-9, 1990.
- [RFC 822] Crocker, D., "Standard for The Format of ARPA Internet Text Messages," STD 11, RFC 822, August 1982.
- [RFC 1321] Rivest, R., "The MD5 Message-Digest Algorithm", RFC 1321, April 1992.
- [RFC 2047] Moore, K., "MIME (Multipurpose Internet Mail Extensions) Part Three: Message Header Extensions for Non-ASCII Text", RFC 2047, November 1996.
- [RFC 2052] Gulbrandsen, A. and P. Vixie, "A DNS RR for specifying the location of services (DNS SRV)", RFC 2052, October 1996.
- [RFC 2060] Crispin, M., "Internet Message Access Protocol - Version 4rev1", RFC 2060, December 1996.
- [RFC 2104] Krawczyk, H., Bellare, M. and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", RFC 2104, February 1997.
- [RFC 2195] Klensin, J., Catoe, R. and P. Krumviede, "IMAP/POP AUTHorize Extension for Simple Challenge/Response", RFC 2195, September 1997.

- [RFC 2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.
- [RFC 2222] Myers, J., "Simple Authentication and Security Layer (SASL)", RFC 2222, October 1997.
- [USASCII] US-ASCII. Coded Character Set - 7-Bit American Standard Code for Information Interchange. Standard ANSI X3.4-1986, ANSI, 1986.

## 6 Authors' Addresses

Paul Leach  
Microsoft  
1 Microsoft Way  
Redmond, WA 98052

EEmail: paulle@microsoft.com

Chris Newman  
Innosoft International, Inc.  
1050 Lakes Drive  
West Covina, CA 91790 USA

EEmail: chris.newman@innosoft.com

## 7 ABNF

What follows is the definition of the notation as is used in the HTTP/1.1 specification (RFC 2616) and the HTTP authentication specification (RFC 2617); it is reproduced here for ease of reference. Since it is intended that a single Digest implementation can support both HTTP and SASL-based protocols, the same notation is used in both to facilitate comparison and prevention of unwanted differences. Since it is cut-and-paste from the HTTP specifications, not all productions may be used in this specification. It is also not quite legal ABNF; again, the errors were copied from the HTTP specifications.

### 7.1 Augmented BNF

All of the mechanisms specified in this document are described in both prose and an augmented Backus-Naur Form (BNF) similar to that used by RFC 822 [RFC 822]. Implementers will need to be familiar with the notation in order to understand this specification.

The augmented BNF includes the following constructs:

**name = definition**

The name of a rule is simply the name itself (without any enclosing "<" and ">") and is separated from its definition by the equal "=" character. White space is only significant in that indentation of continuation lines is used to indicate a rule definition that spans more than one line. Certain basic rules are in uppercase, such as SP, LWS, HT, CRLF, DIGIT, ALPHA, etc. Angle brackets are used within definitions whenever their presence will facilitate discerning the use of rule names.

**"literal"**

Quotation marks surround literal text. Unless stated otherwise, the text is case-insensitive.

**rule1 | rule2**

Elements separated by a bar ("|") are alternatives, e.g., "yes | no" will accept yes or no.

**(rule1 rule2)**

Elements enclosed in parentheses are treated as a single element. Thus, "(elem (foo | bar) elem)" allows the token sequences "elem foo elem" and "elem bar elem".

**\*rule**

The character "\*" preceding an element indicates repetition. The full form is "<n>\*<m>element" indicating at least <n> and at most <m> occurrences of element. Default values are 0 and infinity so that "(element)" allows any number, including zero; "1\*element" requires at least one; and "1\*2element" allows one or two.

**[rule]**

Square brackets enclose optional elements; "[foo bar]" is equivalent to "1(foo bar)".

**N rule**

Specific repetition: "<n>(element)" is equivalent to "<n>\*<n>(element)"; that is, exactly <n> occurrences of (element). Thus 2DIGIT is a 2-digit number, and 3ALPHA is a string of three alphabetic characters.

**#rule**

A construct "#" is defined, similar to "\*", for defining lists of elements. The full form is "<n>#<m>element" indicating at least <n> and at most <m> elements, each separated by one or more commas (",") and OPTIONAL linear white space (LWS). This makes the usual form of lists very easy; a rule such as

```
( *LWS element *( *LWS "," *LWS element ))
can be shown as
1#element
```

Wherever this construct is used, null elements are allowed, but do not contribute to the count of elements present. That is, "(element), , (element) " is permitted, but counts as only two elements. Therefore, where at least one element is required, at least one non-null element MUST be present. Default values are 0 and infinity so that "#element" allows any number, including zero; "1#element" requires at least one; and "1#2element" allows one or two.

; comment

A semi-colon, set off some distance to the right of rule text, starts a comment that continues to the end of line. This is a simple way of including useful notes in parallel with the specifications.

implied \*LWS

The grammar described by this specification is word-based. Except where noted otherwise, linear white space (LWS) can be included between any two adjacent words (token or quoted-string), and between adjacent words and separators, without changing the interpretation of a field. At least one delimiter (LWS and/or separators) MUST exist between any two tokens (for the definition of "token" below), since they would otherwise be interpreted as a single token.

## 7.2 Basic Rules

The following rules are used throughout this specification to describe basic parsing constructs. The US-ASCII coded character set is defined by ANSI X3.4-1986 [USASCII].

OCTET	= <any 8-bit sequence of data>
CHAR	= <any US-ASCII character (octets 0 - 127)>
UPALPHA	= <any US-ASCII uppercase letter "A".."Z">
LOALPHA	= <any US-ASCII lowercase letter "a".."z">
ALPHA	= UPALPHA   LOALPHA
DIGIT	= <any US-ASCII digit "0".."9">
CTL	= <any US-ASCII control character (octets 0 - 31) and DEL (127)>
CR	= <US-ASCII CR, carriage return (13)>
LF	= <US-ASCII LF, linefeed (10)>
SP	= <US-ASCII SP, space (32)>
HT	= <US-ASCII HT, horizontal-tab (9)>
<">	= <US-ASCII double-quote mark (34)>
CRLF	= CR LF

All linear white space, including folding, has the same semantics as SP. A recipient MAY replace any linear white space with a single SP before interpreting the field value or forwarding the message downstream.

LWS = [CRLF] 1\*( SP | HT )

The TEXT rule is only used for descriptive field contents and values that are not intended to be interpreted by the message parser. Words of \*TEXT MAY contain characters from character sets other than ISO-8859-1 [ISO 8859] only when encoded according to the rules of RFC 2047 [RFC 2047].

TEXT = <any OCTET except CTLs,  
but including LWS>

A CRLF is allowed in the definition of TEXT only as part of a header field continuation. It is expected that the folding LWS will be replaced with a single SP before interpretation of the TEXT value.

Hexadecimal numeric characters are used in several protocol elements.

HEX = "A" | "B" | "C" | "D" | "E" | "F"  
| "a" | "b" | "c" | "d" | "e" | "f" | DIGIT

Many HTTP/1.1 header field values consist of words separated by LWS or special characters. These special characters MUST be in a quoted string to be used within a parameter value.

token = 1\*<any CHAR except CTLs or separators>  
separators = "(" | ")" | "<" | ">" | "@"  
| "," | ";" | ":" | "\" | "<">  
| "/" | "[" | "]" | "?" | "="  
| "{" | "}" | SP | HT

A string of text is parsed as a single word if it is quoted using double-quote marks.

quoted-string = ( "<"> qdstr-val "<"> )  
qdstr-val = \*( qdtext | quoted-pair )  
qdtext = <any TEXT except "<">>

Note that LWS is NOT implicit between the double-quote marks (<">) surrounding a qdstr-val and the qdstr-val; any LWS will be considered part of the qdstr-val. This is also the case for quotation marks surrounding any other construct.



The backslash character ("\") MAY be used as a single-character quoting mechanism only within qdstr-val and comment constructs.

quoted-pair = "\" CHAR

The value of this construct is CHAR. Note that an effect of this rule is that backslash must be quoted.

## 8 Sample Code

The sample implementation in [Digest] also applies to DIGEST-MD5.

The following code implements the conversion from UTF-8 to 8859-1 if necessary.

```
/* if the string is entirely in the 8859-1 subset of UTF-8, then
 * translate to 8859-1 prior to MD5
 */
void MD5_UTF8_8859_1(MD5_CTX *ctx, const unsigned char *base,
                     int len)
{
    const unsigned char *scan, *end;
    unsigned char cbuf;

    end = base + len;
    for (scan = base; scan < end; ++scan) {
        if (*scan > 0xC3) break; /* abort if outside 8859-1 */
        if (*scan >= 0xC0 && *scan <= 0xC3) {
            if (++scan == end || *scan < 0x80 || *scan > 0xBF)
                break;
        }
    }
    /* if we found a character outside 8859-1, don't alter string
     */
    if (scan < end) {
        MD5Update(ctx, base, len);
        return;
    }

    /* convert to 8859-1 prior to applying hash
     */
    do {
        for (scan = base; scan < end && *scan < 0xC0; ++scan)
            ;
        if (scan != base) MD5Update(ctx, base, scan - base);
        if (scan + 1 >= end) break;
        cbuf = ((scan[0] & 0x3) << 6) | (scan[1] & 0x3f);
        MD5Update(ctx, &cbuf, 1);
    } while (scan < end);
}
```

```
        base = scan + 2;  
    } while (base < end);  
}
```

## 9 Full Copyright Statement

Copyright (C) The Internet Society (2000). All Rights Reserved.

This document and translations of it may be copied and furnished to others, and derivative works that comment on or otherwise explain it or assist in its implementation may be prepared, copied, published and distributed, in whole or in part, without restriction of any kind, provided that the above copyright notice and this paragraph are included on all such copies and derivative works. However, this document itself may not be modified in any way, such as by removing the copyright notice or references to the Internet Society or other Internet organizations, except as needed for the purpose of developing Internet standards in which case the procedures for copyrights defined in the Internet Standards process must be followed, or as required to translate it into languages other than English.

The limited permissions granted above are perpetual and will not be revoked by the Internet Society or its successors or assigns.

This document and the information contained herein is provided on an "AS IS" basis and THE INTERNET SOCIETY AND THE INTERNET ENGINEERING TASK FORCE DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

## Acknowledgement

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

