1 Attacking the challenge response protocol

By looking into Authorization headers passed to the server on several HTTP GET request over time, we can confirm that HTTPd's standard for nonce lifetime is set to 300 seconds. When the nonce lifetime has run out, the client is initially unaware. It will try using the old nonce but is notified by the server after the first failure. When an old nonce appears at a server, it responds with a new 401 Authorization Required with a fresh nonce value and the stale-flag set to true, indicating to the client that the nonce was aged and has been replaced. This method is specified in the RFC. If the user agent has the credentials remembered, the transition to a fresh nonce will go automatically and a valid response value is calculated, re-authenticating the user.

The only value changing between each HTTP GET within one such *nonce lifetime* (five minutes) is the nonce counter, nc, which is incremented on every client request. Thus is the only value responsible for the change of the response value between requests.

This situation is analogous to the classic problem of cracking a hashed and salted password: In the HA1 calculation, the static values (username, realm and colons) are analoguous to *salt*. In the response calculation we consider the HA1 value and the static values (nonce, nc, cnonce, qop and HA2, separated by colons) analogue to *password* and *salt* respectively.

We have the following scenario:

1. One or more HTTP GET requests containing an Authorization header are *snooped*, i.e. read off the network cable or wireless channel by a *Man In The Middle*.

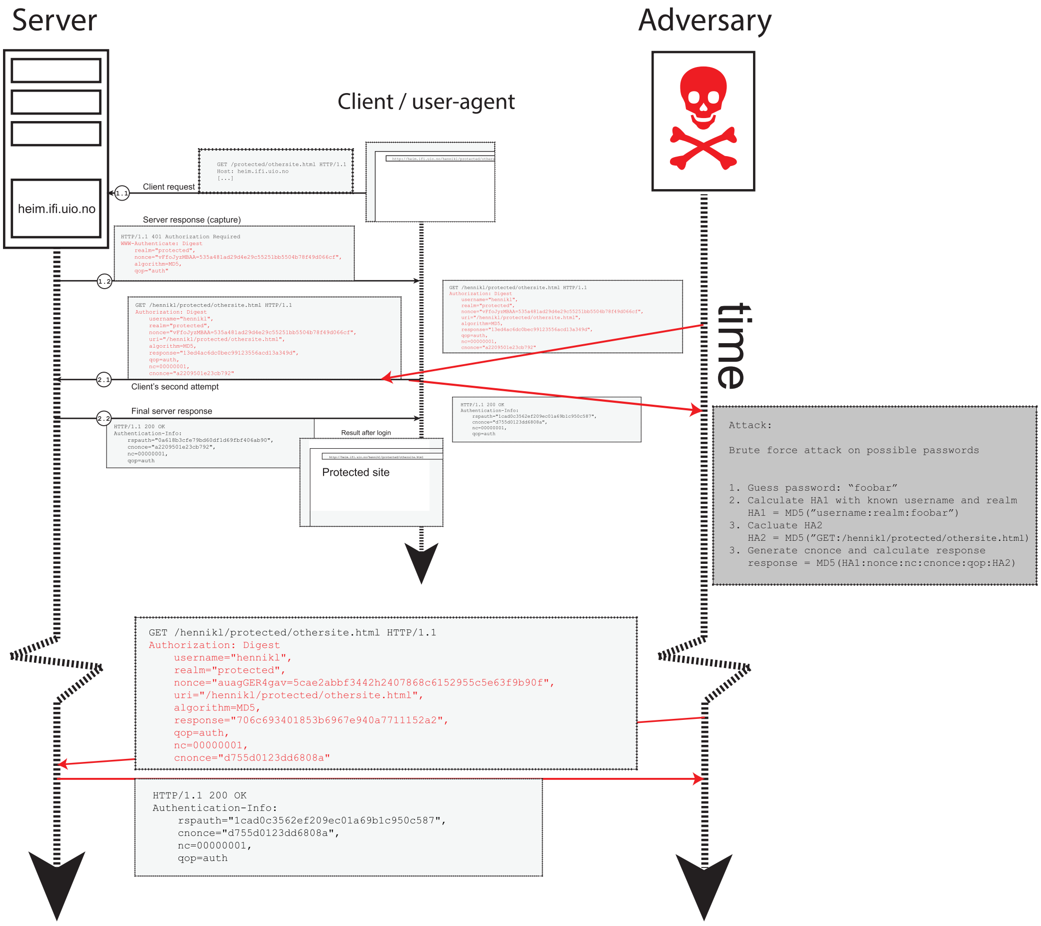
2. A possibly high number of different response values that are hashes of the same combination of header data and a different, known salt (nc) each time.

An Authorization header's response value is an expression on the form:

$$HA1 = MD5(s_1||password)$$
$$response = MD5(HA1||s_2)$$

Where || is string concatenation, s_1 and s_2 are the static values, of the format "username:realm:" and "nonce:nc:cnonce:qop:HA2" respectively.

A successful brute force attack on the password will reveal the static secret *HA1* value, that in turn can be validated by the *response* calculation above. Attempting to break the one-way property of MD5 is not practical at the time of writing. The most effective known preimage attack has a



```
GET /protected/othersite.html HTTP/1.1
Host: heim.ifi.uio.no
[...]
```

```
HTTP/1.1 401 Authorization Required
WWW-Authenticate: Digest
    realm="protected",
    nonce="vFfoJyzMBAA=535a481ad29d4e29c55251bb5504b78f49d066cf",
    algorithm=MD5,
    qop="auth"
```

```
GET /hennikl/protected/othersite.html HTTP/1.1
Authorization: Digest
    username="hennikl",
    realm="protected",
    nonce="vFfoJyzMBAA=535a481ad29d4e29c55251bb5504b78f49d066cf",
    uri="/hennikl/protected/othersite.html",
    algorithm=MD5,
    response="13ed4ac6dc0bec99123556acd13a349d",
    qop=auth,
    nc=00000001,
    cnonce="a2209501e23cb792"
```

```
GET /hennikl/protected/othersite.html HTTP/1.1
Authorization: Digest
    username="hennikl",
    realm="protected",
    nonce="vFfoJyzMBAA=535a481ad29d4e29c55251bb5504b78f49d066cf",
    uri="/hennikl/protected/othersite.html",
    algorithm=MD5,
    response="13ed4ac6dc0bec99123556acd13a349d",
    qop=auth,
    nc=00000001,
    cnonce="a2209501e23cb792"
```

```
HTTP/1.1 200 OK
Authentication-Info:
    rspauth="0a618b3cfe79bd60df1d69fbf406ab90",
    cnonce="a2209501e23cb792",
    nc=00000001,
    qop=auth
```

```
HTTP/1.1 200 OK
Authentication-Info:
    rspauth="1cad0c3562ef209ec01a69b1c950c587",
    cnonce="d755d0123dd6808a",
    nc=00000001,
    qop=auth
```

```
Attack:

Brute force attack on possible passwords


1. Guess password: “foobar”
2. Calculate HA1 with known username and realm
   HA1 = MD5(”username:realm:foobar”)
3. Cacluate HA2
   HA2 = MD5(”GET:/hennikl/protected/othersite.html)
3. Generate cnonce and calculate response
   response = MD5(HA1:nonce:nc:cnonce:qop:HA2)
```

```
GET /hennikl/protected/othersite.html HTTP/1.1
Authorization: Digest
    username="hennikl",
    realm="protected",
    nonce="auagGER4gav=5cae2abbf3442h2407868c6152955c5e63f9b90f",
    uri="/hennikl/protected/othersite.html",
    algorithm=MD5,
    response="706c693401853b6967e940a7711152a2",
    qop=auth,
    nc=00000001,
    cnonce="d755d0123dd6808a"
```

```
HTTP/1.1 200 OK
Authentication-Info:
    rspauth="1cad0c3562ef209ec01a69b1c950c587",
    cnonce="d755d0123dd6808a",
    nc=00000001,
    qop=auth
```

Figure 1: An attack on digest authentication

password	`HA1`	`response`
hennikl:realm:a	7a29f74992 . . . a0fdda	8f2a65080a8761d4ee6da59544ccc186
hennikl:realm:b	e3e5f66d00 . . . de36f6	61f33b9a185989d9f215c916e67afa68
hennikl:realm:c	53d83d97af . . . 05404e	10bb7eca0fcf8876b3cf8a44fa98a185
hennikl:realm:d	9ef81d5334 . . . 769b8a	beaa89ecc4c7f7863982ff1f65efd65e
hennikl:realm:e	a0b1fae479 . . . 397e1d	d17b3e2a0ddc857329c3aa4df232736d
hennikl:realm:f	5a991fb4a4 . . . d5fc0c	49f275d192e250e8a5787284298f8e05
hennikl:realm:g	4f71d53117 . . . 84d22d	3cc9c20d9a0006cb8cc371b3c873aec4
hennikl:realm:h	dabeeba885 . . . 78a195	d3c91f4d2fe3ca8ba28f9bb3cbefd4b9
hennikl:realm:i	115dd27a08 . . . b633ca	1183e7df1779e43473e26e0febd6148d
. . .		
hennikl:realm:passwor5	9684b080fa . . . 9901f4	b815adac2c1045b40ed621c347b661a8
hennikl:realm:passwor6	3caa6da3aa . . . 69b735	31380b94e595f449ccafc2bf9063ff2b
hennikl:realm:passwor7	14fe235f79 . . . abb63c	35127fbc6b025ccc8540b690d7958013
hennikl:realm:passwor8	b1a7fe3369 . . . f0966c	26794bc1025236342aee2c653d52a7c9
hennikl:realm:passwor9	7f8f6ef704 . . . 7c0ff3	c7b18e356f24ed141dbd9e0cf3722a89
hennikl:realm:passwora	d356381226 . . . 9cd4e3	28aabbb0f847fdabc899ae0f949caae7
hennikl:realm:passworb	5b4415f182 . . . 7b8b95	9a816243210af42e50767f61ba0fea7c
hennikl:realm:passworc	e281050e8f . . . 50f93a	d62ef054edaa5b216db454e565e36a0a
hennikl:realm:password	f19b0a03ee . . . 471dcf	f15370722fb0a84b799c669cdb4b35d6

Table 1: An example of an exhaustive search

computational complexity of $2^{123.4}$. To find a usable password[1], however, we must find the preimage of the `HA1` value, which itself is hashed.

However, attacking from another angle is possible. As we collected the `Authorization` header, we collected all the values needed to calculate the `HA1` except the password. Actually, all values making up the entire final `response` are accessible should we find the correct password. Exhaustively searching the available preimages' character space is a usual approach to *password cracking* on hashed passwords. In this scenario we must customize the password cracking algorithm to first hash the guessed password together with the rest of the `A1` parameters to recreate a suggestion for `HA1`. Second, we must use that `HA1` value in the `response` calculation (using the retrieved nonces and other collected parameters) to produce a possible response value. Finally, in the (unlikely) event that the `response` value equals the one collected, we have collected a password that is usable in any session with the same system. We have decoupled the password from the nonce- and client nonces. In table 1 and figure 1 below, we show how this approach is possible, using these example values:

[1] Although this is a fully usable password within the system, it is only so because the calculations yield the same `response` value. There is no certainty as to whether the password retrieved is the password the user originally selected, so it is not guaranteed transferable to other systems.

```
username: hennikl
realm:    realm
nonce:    aGVsbG8=feffda0520707fc331d9be9eff74eab1eb7cafe4
cnonce:   SHZlbSBoYWRkZSB0cm9kZCBhdCBkZXQgc3RvZCBub2UgaGVyPw==
nc:       00000001
uri:      /foo/
method:   GET

correct password: password
correct HA1:      f19b0a03eead15d687986227dc471dcf
                  MD5("hennikl:realm:password");
correct HA2:      2e18ba280b7f2a4e2785f9d88fc7aa72
                  MD5("GET:/foo/");
correct response: f15370722fb0a84b799c669cdb4b35d6
                  MD5(HA1:nonce:nc:cnonce:auth:HA2);
```

Each of the text values are to be interpreted as strings of bytes. The presented hashes are strings of bytes in hexadecimal representation. If multiple `Authorization` headers are collected, each have different `nc` values, but may be attacked in parallel processes, with equal probability of recovering the password.

1.1 Pseudocode

This section contains pseudocode showing the execution of the exhaustive attack on the digest authentication scheme. It follows the same general recipe as any brute force hash cracking algorithms, namely going through an entire dictionary until the attack succeds. Algorithm 1 shows the calculation of the `response` value. Algorithm 2 iterates through a stack of suspected passwords in a dictionary, passing each password suggestion to the response calculator. It requires the data from one intercepted `Authorization` header.

The procedures `DigestCalcHA1`, `DigestCalcHA2` and `DigestCalcResponse` are equal to those specified in RFC2617. The one-way function, however, which is `MD5` in the specification, may be regarded as any other one-way function as long as it is used equally on both endpoints of the authentication. Synchronization of the selected one-way function is done using the `algorithm` field.

```
Procedure: CalculateResponse
Data: pszMethod, pszDigestUri, pszQop,
      pszUsername, pszRealm, pszPassword,
      pszNonce,pszCNonce, pszNonceCount
Result: pszResponse
$ha1 \leftarrow DigestCalcHA1(pszUsername, pszRealm, pszPassword)$
$ha2 \leftarrow DigestCalcHA2(pszMethod, pszDigestUri, pszQop)$
$response \leftarrow$
$DigestCalcResponse(ha1, ha2, pszMethod, pszDigestUri,$
      $pszQop, pszNonce, pszNonceCount, pszCNonce)$
return $null$
```

Algorithm 1: This procedure calculates the `response` value

```
Procedure: DigestDictionaryAttack
Data: pszMethod, pszDigestUri, pszQop,
      pszUsername, pszRealm, pszNonce,
      pszCNonce, pszNonceCount, pszTargetResponse
Result: pszPassword or null
while dictionary is not empty do
    $pszPassword \leftarrow dictionary.pop()$
    $res \leftarrow CalculateResponse(pszMethod, pszDigestUri,$
          $pszQop, pszUsername, pszRealm, pszPassword,$
          $pszNonce, pszCNonce, pszNonceCount)$
    if $res == pszTargetResponse$ then
        return $pszPassword$
    end
end
return $null$
```

Algorithm 2: An approach to retrieving a password used in HTTP Digest Access Authentication