nccgroup

NCC Group Whitepaper

My Hash Is My Passport: Understanding Web and Mobile Authentication

June 29, 2016 – Version 1.0

Prepared by
David Schuetz – Senior Security Consultant

Abstract

Andrew Tanenbaum once said, *"The great thing about standards is there are so many to choose from."* That's especially true in the realm of web and mobile application authentication. From Base-64 to OAuth, there are nearly as many ways to send your password to a server as there are ways to store that password.

But how do these work? Is any one system better than another, and if so, why?

This paper explains, with simple examples, how some of the most frequently seen authentication systems work. It identifies the characteristics of an "ideal" authentication system, compares the common methods against that ideal, and demonstrates how to verify that they've been implemented correctly.

Table of Contents

1 Introduction . . . 3

1.1 Background . . . 3

1.2 Goals . . . 3

1.3 Focus . . . 3

2 Evaluating Strengths . . . 5

2.1 Definitions . . . 5

2.2 Real World Example . . . 6

2.3 Application Authentication . . . 7

2.4 Attacks . . . 7

3 Review of Common Systems . . . 9

3.1 Password-Based . . . 9

3.2 Digest . . . 11

3.3 NTLM . . . 13

3.4 OAuth 1.0 . . . 18

3.5 OAuth 2.0 . . . 22

3.6 Other Systems . . . 25

4 Summary of System Capabilities . . . 30

5 Other Considerations . . . 32

5.1 Server . . . 32

5.2 Clients . . . 32

5.3 Developers . . . 33

6 Conclusion . . . 35

1 Introduction

1.1 Background

Much of the content on the World Wide Web is freely available to what are basically anonymous users, who don't need to identify themselves to the remote site. But a large fraction of it requires the reader to somehow identify themselves to the site, such as by using a username and password. Web forums, social media, online shopping, banking, and health care sites – all these require the user to prove who they are.

The average user simply enters their username and password, hit a button, and then they're in. But how exactly this is handled behind the scenes remains a mystery for most people, and can even be confusing for the security experts who test such sites.

This paper, and the ShmooCon talk which accompanies it[Sch16], attempts to explain how such login, or "authentication", systems operate, both for services accessed through a web browser, and those reached via a dedicated mobile application.

> A note about nomenclature: the words "Authentication," "Authorization," and "Identification" each have very specific meanings, especially in information security, but are frequently conflated and interchanged in casual conversation. The focus of this paper is the process of a user proving their identity to a remote server, via a form on a website or some mechanism within a mobile application. It is that process that I'm calling Authentication, though in some cases it may be more properly called Identification, or for some protocols, Authorization. I'm trying not to be too pedantic.

1.2 Goals

There are three primary goals for this paper:

- Explain, in simple terms, how the most popular authentication systems operate. The hope is that this can help provide a better understanding of the risks of online services, to make the system a little bit less like "black magic."
- Where possible, identify the good and bad qualities of these systems. The intent is to help those in security-related positions to be able to better analyze strengths and weaknesses associated with different authentication techniques. This should allow them to make a better assessment of the risks posed by applications using these techniques. This can also be used by development teams to design more secure systems.

1.3 Focus

There is no limit to the number of ways that an application can authenticate a user to a server. This paper will discuss in detail the systems most frequently encountered by the author, but will also touch lightly upon some other methods of historical or increasing importance.

Not all authentication systems reviewed in this paper are designed for both login and session authentication. An application may make use of one system for initial login and use a different one for securely handling session authentication. This paper will consider protocols appropriate to either, or both, of those use cases.

This paper only considers direct communication of authentication data between the application and the server. The world of authentication technology is far wider than that, and includes:

- Federated 3rd-party logins ("Login using your Twitter account")
- Token-based two-factor authentication (Duo, Yubikey, RSA tokens)

- Other two-factor authentication (Duo, Google Authenticator, out-of-band SMS, etc.)
- Technologies generally reserved for specific applications such as email, etc.:
  - CRAM-MD5
  - Kerberos
  - Public key authentication for SSH
  - Secure Remote Password (SRP, TLS-SRP), and other Password-Authenticated Key Agreement (PAKE) systems
  - Opening a secure channel and exchanging credentials through that channel
  - Totally out-of-band authentication systems

Finally, this paper focuses on the client / server relationship versus an external adversary, with the presumption that neither client nor server has been compromised. It will not address attacks against either endpoint, nor intermediate attacks such as phishing. Also, the paper will not directly address cryptographic or other attacks based on weak implementations. All of the systems presented in this paper have had their share of vulnerabilities and problems.

2 Evaluating Strengths

It is not enough to simply catalog frequently used authentication methods; instead, this paper will attempt to present an objective analysis of the relative merits and shortfalls of these methods. Four primary criteria will be submitted as (in the author's opinion) the most important for secure and reliable authentication, and several methods will be assessed using these criteria as a yardstick.

2.1 Definitions

Some terms have already been discussed, and several others will occur with frequency throughout this paper. It may be helpful to define some of them early.

- ***Userid***: The name by which a user identifies themselves to a system. This may be a username, a real name, a phone number, an email address, or even some long, hidden number presented to the server by an application.
- ***Password***: A piece of information which the user presents to prove that they are the person who they claim to be. Generally a string of user-enterable characters, but may also be something derived from a biometric identifier such as a fingerprint, or a one-time code.
- ***Session Token***: A frequently random string that is used during communication to identify requests as part of an active, authenticated session. Tokens should expire after a set time frame, but while still active, are typically sufficient to authenticate any request.
- ***Participant***: One of the systems involved in an authentication transaction. Most transactions will have two participants: The client (the end-user's web browser or mobile application) and the server to which the application or browser is connecting. Some authentication systems have additional participants.
- ***Nonce***: A randomly generated number or string of data, generally used to ensure that a transaction which may otherwise contain the same information every time (userid, password) is unique with each occurrence. May be inserted into the exchange by any participant. Tracking the use of nonces allows a participant to refuse any transaction that reuses a nonce.
- ***Timestamp***: Similar to a nonce, but uses the current time rather than a random string. This can be used in the same manner as a nonce, but has the advantage of being self-expiring, simply by refusing transactions with a timestamp older than some specified time-period. This avoids requiring the server to track an excessive number of nonces.
- ***Hash***: A mathematical process by which a string of data is reduced to a pre-determined number of characters, called a digest. Hashes are a "one-way function," in that a given string can be consistently converted to a single hash, but it is not possible to take a hash and reverse the process to arrive at the original string. (See also the sidebar on "Strong Hashes" below.) Common hash functions discussed here include MD5, SHA1, and SHA-256. Hashes are frequently used as an authenticating code, to verify that a piece of text has not been altered, but in practice this is not entirely a correct use of hashes. Hashes are frequently used for storage of passwords on servers.
- ***Salt***: A short, generally random string, which is combined with a user's password to make their individual password hash unique. Random salts thus prevent hashes from appearing in pre-computed password hash lists. The salt and password are generally hashed together, as a single string, with the result stored in the database. The salt is also stored alongside the password hash, to allow the salted hash to be recreated when authenticating a user.
- ***HMAC***: A Hash-based Message Authenticating Code, the HMAC combines a hash function with a secret key, to create a signed, or authenticated, hash of a given piece of text. The underlying hash function used

by the HMAC function may be varied depending on needs and is generally written alongside the HMAC term, to define the appropriate combined function. HMAC-MD5, HMAC-SHA, and HMAC-SHA256 are all frequently used HMAC forms.

- ***Transport Layer Security (TLS)***: Encryption of the network transmissions between a client and server, usually making use of a Public Key Infrastructure (PKI). Originally handled by the Secure Socket Layer (SSL) specification, SSL has been deprecated in favor of more modern TLS systems.

> ***A Note On Strong Hashes***
>
> Many hash functions are designed to execute as rapidly as possible. This allows systems and protocols to behave at a high level of performance when many hash operations are required. Unfortunately, this also makes it more feasible to "brute force" the original text for some hashes.
>
> This technique is used for cracking passwords and other short strings (with today's technology, generally about a dozen characters or less). Many hash functions can be optimized for use on dedicated hardware processors, which further increases the speed by which passwords can be attacked. Another attack that is frequently undertaken is the pre-computation of many hashes, based on known password lists or using a "rainbow table." Such databases can reduce a password cracking operation to a simple database lookup, or at least drastically reduce the number of guesses required.
>
> To combat this, passwords may be combined with a salt. However, even salted hashes are not a strong deterrent to an attacker intent on cracking a single, particular password. For this reason, several strong hash functions have been developed specifically for the storage and processing of passwords. These include bcrypt, scrypt, and PBKDF2 (Password Based Key Derivation Function, version 2). These functions are much slower to execute than other hashes, and in some cases, are intentionally designed to be difficult to run on dedicated hardware.
>
> It is almost always preferable for servers to store passwords using a strong password hashing function. If such a function is infeasible to use, then a larger, salted hash may be acceptable, especially if other steps are taken to mitigate the risks. The use of salted hashes without additional mitigations, or of unsalted, fast hashes (to say nothing of the plaintext storage of passwords) is to be avoided at all possible costs.

2.2 Real World Example

To begin, imagine authentication in the "real world:"

1. Your friend, "Tim," walks up and says "Hello."
2. You immediately recognize him, and begin a private conversation.

This is probably the most obvious and simple way one can perform authentication, something we've done all of our lives. However, it also exhibits all the key traits of a secure authentication system:

1. It is *repeatable*. As long as Tim's appearance has not changed substantially, your criteria for evaluating the "Timness" of the person in front of you is consistent.
2. It is *non-transferable*. Someone may approach wearing a perfect Tim mask, but by checking for the presence of a mask (perhaps you're from a culture where friends greet with a kiss on the cheeks), the presented face can be verified.
3. It is *revocable*. If one day Tim introduces you to his identical twin brother "Tom," then you will mentally

make a note of the fact that there's a twin out there. Some new criteria beyond simple facial recognition will be employed at future meetings.

4. It is *observable*. The entire transaction can be carried out in the full view of others not party to the process, with no reduction in security.

While this isn't a perfect analogy, it should help to illustrate the concepts.

2.3 Application Authentication

In the "Internet world," authentication is not as simple. Similarly, we use knowledge to identify and authenticate, but that knowledge must be something that can be entered into a computer. Typically, this means a password. If that password is stolen, then the capability to authenticate as that user is given to another. How do we build a system that supports the criteria above, while retaining the ease of use for users and systems?

When discussing the use of authentication to a remote site or service, it is helpful to distinguish between two different phases of the process: Login and session. The login is the first use of a website or application – in some situations, this is the first time the application is launched, while in others, a new login may be required every day, or even more frequently. Session authentication is a continuous proof of authentication, sent with each request made of the remote system.

Each phase presents slightly different risks:

- *Login*: An attacker may steal credentials and reuse within the application, or other related applications. For example, an attacker might steal a password from an active network connection and then use it on the system's website. A password, once stolen, may be valid indefinitely.
- *Session*: An attacker may steal application-specific session credentials, or tokens. These tokens may be used within the current application, or possibly other, related applications. Another session-level attack is the modification of requests in-transit to perform some malicious action. A stolen session token may be valid indefinitely, or for some limited timespan.

2.4 Attacks

Any protocol can be perfectly secure, if it's perfectly hidden. However, this is rarely seen in practice, if it's even technically possible at all. Multiple attacks may lead to the disclosure of passwords, session tokens, and password- equivalents, such as hashes or other strings used as part of authentication.

Passwords: The user's password may be compromised. If an attacker is able to extract a plaintext password from, for example, a browser cache, or crack it from a server's password file, then that attacker can authenticate as that user at any time in the future. Therefore, preventing the disclosure of the password is highly important.

However, in some cases, a user's password may not actually be present in a login request, with the system instead utilizing some other password equivalent (usually derived from the password). A stolen password equivalent may only be useful within the scope of a particular type of application, perhaps even limited to a specific platform such as iOS or Android.

Passwords can usually be changed or reset by the user. In some cases, the mechanism provided to reset or change a password may introduce other weaknesses to the overall security of the system.

Tokens: Stolen session tokens may be useful for maliciously forging requests, and can sometimes be transferred between mobile and browser-based applications. In some cases, tokens can also be revoked by the

user, especially if they are more directly associated with particular applications or devices (as is the case with some OAuth 1 configurations). Depending on the form of session authentication in place, stolen session tokens may be valid:

- For any and all new requests
- For any new request, but one-time only (once used, the authentication token is no longer valid)
- For only a single, specific request (the token is directly tied to a prepared request)

Even "one-time" session tokens may be misused. An attacker may intercept a request, prevent its delivery to the server, and use the token with their own request. The attacker then sends an error to the original user, who simply shrugs it off and re-sends the request.

"It's okay, we're using TLS"

Most of the attacks described here presume the ability to observe the actual client / server communications. In the unlikely event that the service is using plaintext connections with no encryption, this can be trivial, for an attacker located "nearby" their target. In the case of a TLS-secured connection, the contents of the communication may be revealed through:

- Obtaining a certificate and key for the target server by exploiting a publicly trusted Certificate Authority (CA)
- Bypassing CA trust evaluations through bugs in underlying TLS libraries or other system-level vulnerabilities
- Exploiting a corporate-level TLS inspection proxy that decrypts and re-encrypts transmissions to fulfill Data Loss Prevention requirements
- Causing the user to install and trust a malicious CA certificate and proxy configuration, through phishing attacks or system-level vulnerabilities
- Various TLS-level vulnerabilities and weaknesses, such as BEAST or Heartbleed.

An ideal authentication system will take these risks into consideration and be built to withstand complete scrutiny by a well-positioned adversary.

Ideally, a system should be:

- Safe from in-transit observation. Complete observation of protocol should not lead to a compromise of the user's authentication.
- Safe from at-rest observation. No credentials should be stored that can be easily used to begin or continue a session.
- Resistant to replay or modification of requests. Ideally, the authentication should be directly tied to the request, so that requests with valid authentication cannot be forged or modified.

3 Review of Common Systems

With a common set of criteria and definitions established, we can shift our focus to understanding some of the most common systems in use today.

This paper will only focus on the generation and presentation of the user's credentials to the server. The detailed mechanics of the exchange between the server and client (for example, use of "401 Unauthorized" responses, HTML forms, etc.) are implementation details not within the scope of this discussion.

Only a few of the methods described in this paper are natively supported by "all" browsers today: Session Cookies, Passwords (and the related "Authentication: Basic" method), Digest mode, and generally NTLM. Some browsers may support additional methods, but those are not universally supported at this time. A mobile application can support nearly any system the developers wish to include.

3.1 Password-Based

The simplest method of authentication an application can use is to present a user's password directly to the server. This may be hidden or obfuscated in some ways, but generally, they all amount to the same thing: the password.

In the simplest example, an application asks the user for their userid and password, and then sends it to the server as parameters:

```
GET /login?userid=tim&password=Enchanter
```

Listing 1: Simple GET

This approach is bad on many levels:

- The password is provided as a URL parameter, which means that some network devices, which track or list all URLs passing through, may record the password in a log file.
- If the communication is observable, such as over an unencrypted connection or because an attacker has successfully performed a man-in-the-middle (MITM) attack, then the password can immediately be extracted.
- If the password is recovered, it can be reused within the same application on an attacker's device and potentially in other applications. For example, the password extracted from a mobile application's database or active connection could be used to login to a web-based application for the same service.

One way to improve this usage is to move the credentials out of the GET parameter list and into a POST body or the HTTP headers:

```
POST /login HTTP/1.1
[...]

userid=tim&password=Enchanter
```

Listing 2: Password in POST Body

```
GET /login HTTP/1.1
[...]
Userid: tim
Password: Enchanter
```

Listing 3: Password in HTTP Headers

However, though this reduces the chance of the password being stored in a cache of recently accessed URLs, it does very little to improve the security of the password against an attacker who is able to intercept and read network traffic.

Another method is to obfuscate the password, frequently using Base-64 encoding. A web site can automatically trigger a browser-based password field by including the "401 Unauthorized" response code, and can restrict the reuse of passwords by defining a realm:

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Basic realm="caerbannog"

[...]

GET /login HTTP/1.1
Authorization: Basic dGltOkVuY2hhbnRlcg==
```

Listing 4: Base-64 Obfuscation using Authentication: Basic

In this case, the string "tim:Enchanter" is Base-64 encoded as "dGltOkVuY2hhbnRlcg==". However, this is so easily reversed that it might as well be in plaintext anyway. Another method is to provide a hash of the password, but this may be easily cracked (in the case of a weak hash such as MD5) and is not resistant against an attacker simply using the hash of the password instead of the password itself.

If the password itself cannot be recovered from an obfuscated or hashed form, then an attacker may only be able to use it in an identical situation, that is, the same client / server application. For example, an extracted password hash may be sufficient to allow an attacker to access a user's bank account using a hacked copy of a mobile application, but it will not allow the attacker to log in using the bank's website, which requires the actual password text.

Because HTTP-based systems are stateless, some form of authentication credentials must be provided with every request sent to the server. This continuous authentication is generally referred to as a session authenticator, and it can be handled in multiple ways.

In its simplest form, session authentication may simply require presenting the user's password with every single request. This approach, however, puts the user's password at risk for disclosure to an attacker able to sniff their network connection (if they haven't already intercepted the password during the login sequence).

A better solution uses randomly-generated session tokens with a limited lifespan. An attacker who is able to steal the token may be able to re-use it to gain access to a currently-established session, but they will not be able to establish a new session using just the token. However once the session has expired, due to a timeout or the user actively logging out, the stolen session token will no longer be useful.

The vast majority of systems which have been tested by the author used some variant of this approach: plaintext password provided as a POST parameter with subsequent requests to the service using a session-based token.

3.1.1 Assessment

- Weakest of all systems
- Usage sometimes unavoidable especially in web-based systems
- Extensive use for session authentication increases risk of disclosure

3.2 Digest

Digest authentication is somewhat more complicated than simple passwords, but provides protection against a few attacks. Most modern browsers support Digest by default, making it usable for both web and mobile applications. Originally defined in RFC 2609, and expanded in RFC 2617, the discussion here will follow the latter specification[RFC99].

Here is an example of an HTTP Digest authentication exchange:

```
[client]
GET / HTTP/1.1

[server]
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Digest nonce="1450807853.38:E10B:497c0eadca9e962b45e54cd2629399b3",
     realm="caerbannog", algorithm="MD5", opaque="ADAC33E813C0CE930F4744C90E02396E",
    qop="auth", stale="false"

[client]
GET / HTTP/1.1
Authorization: Digest username="tim", realm="caerbannog", nonce="1450807853.38:E10B
    :497c0eadca9e962b45e54cd2629399b3", uri="/", algorithm=MD5, response="
    df529127ed79076430be29b897622ae5", opaque="ADAC33E813C0CE930F4744C90E02396E", qop=
    auth, nc=00000002, cnonce="9cf9d4679b7d83fb"
```

Listing 5: Digest Exchange

Several items are provided by the server:

- A server nonce, unique to the current transaction. The library[pyt10] used when developing this example uses MD5((*timestamp*):(*salt*)(*server-secret*)), but it could really be anything at all. It is helpful to include a timestamp to allow the server to invalidate the nonce, and thus any responses utilizing it, after an adequate delay.
- A realm for which the authentication will be valid. Typically an application name, "API," or something similar.
- The algorithm to be used when calculating the digest. Many browsers support only MD5.
- An "opaque" string. This random string is simply returned by the client in the authentication response.
- The "Quality of Protection", or QOP. This will either be "auth" or "auth-int," and affects how the digest is computed. (see below)

Upon receiving the request the browser prompts the user for their userid and password, and then computes multiple values. First, hash 1 (HA1) is computed using the user's name, digest realm, and password:

```
HA1 = MD5('{username}:{realm}:{password}')

Example:
MD5('tim:caerbannog:Enchanter') = 005a400eaf17492454011ddeb50c4a60
```

Listing 6: Computing HA1

Then, HA2 is computed, using the HTTP method and the URL path, and, optionally, an MD5 hash of the HTTP request body:

```
if qop == 'auth':
  HA2 = MD5(method:url)

if qop == 'auth-int':
  HA2 = MD5(method:uri:MD5(entity-body))

Example:
MD5('GET:/') = 71998c64aea37ae77020c49c00f73fa8
```

Listing 7: Computing HA2

If the QOP was selected as "auth-int," then the request body is also hashed and that hash becomes part of the computation for HA2. Note that this hash is computed against the pre-encoded entity, which is the message body before any Transfer-Encoding is applied. This provides an additional integrity protection, explicitly linking the successful digest authentication with the contents of the request itself. This helps to prevent an attacker from using a properly constructed authentication response with a different, malicious request, or from modifying the request while in-transit. Unfortunately, this feature is not as widely supported.

Finally, the client creates its own random nonce and counter value, which the server tracks to prevent replay of previous authentication responses.

The final response is then calculated as:

```
response = MD5(HA1:server-nonce:counter:client-nonce:qop:HA2)

Example:
MD5('005a400eaf17492454011ddeb50c4a60:
  1450807853.38:E10B:497c0eadca9e962b45e54cd2629399b3:
  00000002:
  9cf9d4679b7d83fb:
  auth:
  71998c64aea37ae77020c49c00f73fa8') = df529127ed79076430be29b897622ae5
```

Listing 8: Computing the Digest Response

The client then provides the calculated response, along with the server and client nonces, the counter, uri, qop, realm, userid, and opaque values. The server makes the same calculation, using the HA1 stored on the server (or calculated real-time using a plaintext copy of the user's password). If the server's calculation matches the response provided by the browser, then the authentication is declared to be successful.

3.2.1 Assessment

- Good against replay attacks (with client / server nonce tracking)
- Does not send plaintext credentials across wire - good against reuse on other platforms
- Prevents modification of request if QOP "auth-int" is used
- Use of MD5 as the hashing algorithm poses a security risk. The user's password may be stored on the server as a hash of '*username:realm:password*', which may be cracked using a brute-force attack.

- Server-side hash compromise allows immediate access to user accounts, without cracking the password
- The completed response may also be vulnerable to a brute-force attack, especially with dedicated hardware
- The auth-int QOP and algorithms other than MD5 may not be broadly or consistently supported.

3.3 NTLM

Part of a larger suite of Microsoft protocols, the NT LAN Manager (NTLM) system can be used by browsers and other applications to provide login authentication. It is not as commonly used for Session authentication, though some servers can support it. The NTLM system is extensive, complicated, and nuanced. A complete discussion of the protocol is beyond the scope of this paper. Detailed documentation is not always clear, as it remains a proprietary protocol. One very useful description of NTLM as a web authentication protocol is included as part of the Samba project[Gla06], while a somewhat older analysis by Ronald Tschalär is useful for filling in some confusing aspects[Tsc03].

When first connecting to an NTLM-controlled server, the system simply tells the browser to reconnect, using NTLM authentication.

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: NTLM
Content-type: text/html
Content-Length: 32

<h1>Authentication Required</h1>
```

Listing 9: Initial NTLM response

The client then responds with an "NTLM Type 1" message, which may include multiple settings flags and a hostname for the client. Here is a hex dump of an example message (with hex dump provided to show the contents of the Base-64 encoded data):

```
Authorization: NTLM TlRMTVNTUAABAAAAB4IIAAAAAAAAAAAAAAAAAAAAAA=

0000000: 4e54 4c4d 5353 5000 0100 0000 0782 0800  NTLMSSP.........
0000010: 0000 0000 0000 0000 0000 0000 0000 0000  ................
```

Listing 10: Type 1 message: Client begins NTLM exchange

This example represents the simplest possible response. The responses from other operating systems and browsers will vary.

The server answers with a challenge, sent as a "server nonce" in the "Type 2" message:

```
WWW-Authenticate: NTLM TlRMTVNTUAACAAAAEAAQADAAAAABAoEAbmT8
M7uSpWcAAAAAAAAAAIAAgABAAAAAbQB5AHQAYQByAGcAZQB0AAIAAAABABQ
AbQB5AGMAbwBtAHAAdQB0AGUAcgAEACgAbQB5AGQAbwBtAGEAaQBuAC4AZQ
B4AGEAbQBwAGwAZQAuAGMAbwBtAAMALABtAHkAYwBvAG0AcAB1AHQAZQByA
C4AZQB4AGEAbQBwAGwAZQAuAGMAbwBtAAAAAAAAAAAAA


0000000: 4e54 4c4d 5353 5000 0200 0000 1000 1000  NTLMSSP.........
```

```
0000010: 3000 0000 0102 8100 6e64 fc33 bb92 a567  0.......nd.3...g
0000020: 0000 0000 0000 0000 8000 8000 4000 0000  ............@...
0000030: 6d00 7900 7400 6100 7200 6700 6500 7400  m.y.t.a.r.g.e.t.
0000040: 0200 0000 0100 1400 6d00 7900 6300 6f00  ........m.y.c.o.
0000050: 6d00 7000 7500 7400 6500 7200 0400 2800  m.p.u.t.e.r...(.
0000060: 6d00 7900 6400 6f00 6d00 6100 6900 6e00  m.y.d.o.m.a.i.n.
0000070: 2e00 6500 7800 6100 6d00 7000 6c00 6500  ..e.x.a.m.p.l.e.
0000080: 2e00 6300 6f00 6d00 0300 2c00 6d00 7900  ..c.o.m...,.m.y.
0000090: 6300 6f00 6d00 7000 7500 7400 6500 7200  c.o.m.p.u.t.e.r.
00000a0: 2e00 6500 7800 6100 6d00 7000 6c00 6500  ..e.x.a.m.p.l.e.
00000b0: 2e00 6300 6f00 6d00 0000 0000 0000 0000  ..c.o.m.........
```

Listing 11: Type 2 message: Server Challenge

This message includes several key points of information:

Type 2 Challenge Contents			
Field Name	Offset	Identifier	Value
Challenge	000018	(n/a)	6e64fc33bb92a567
Target Name	0000030	(n/a)	mytarget
Domain Name	0000040	0200 0000	(null)
Computer Name	0000044	0100	mycomputer
Domain DNS Name	000005c	0400	mydomain.example.com
Computer DNS Name	0000088	0300	mycomputer.example.com

In this example, the user's password is "Enchanter", and the NTLM MD4 hash of this (once expanded to UTF-16 format) is "52dac53d34d998ee55b1137d647e93ce."

3.3.1 NTLM v1

For the original NTLM v1 method, the hash is padded with nulls, broken into three 7-byte strings, and converted into three 8-byte DES keys. This process spreads the 56 bits across all 8 bytes, saving the least significant bit of each byte for a parity check.

Those keys are each used to encrypt the challenge using DES ECB mode. The responses are concatenated to produce the final NTLM response. In this example:

```
NTLM hash, padded:  52dac53d34d998 ee55b1137d647e 93ce0000000000
Final DES key:  526db0a7d3a76731 ef2a6d2337ea91fd 92e6800101010101
Server challenge:  6e64fc33bb92a567

key1 = 526db0a7d3a76731, ciphertext1 = 8f86b036e38ac5b8
key2 = ef2a6d2337ea91fd, ciphertext2 = f5913b338af9912e
key3 = 92e6800101010101, ciphertext3 = e3037866d94f62d7

NTLM V1 response:  8f86b036e38ac5b8 f5913b338af9912e e3037866d94f62d7
```

Listing 12: Computing the Response, NTLM v1

A similar process is followed with the LAN Manager (LM) hash, and both results are included in the response to the server. The actual response, a "Type 3" NTLM message, includes the user name, workstation name,

authentication target name, the responses, and some additional information and framing.

```
Authorization: NTLM TlRMTVNTUAADAAAAGAAYAGQAAAAYABgAfAAAAAA
AAABAAAAABgAGAEAAAAAeAB4ARgAAAAAAAAAAAAAAAQIAAHQAaQBtAGcAbA
BhAG0AZAByAGkAbgBnAC4AbABvAGMAYQBsAI+GsDbjisW49ZE7M4r5kS7jA
3hm2U9i14+GsDbjisW49ZE7M4r5kS7jA3hm2U9i1w==

0000000: 4e54 4c4d 5353 5000 0300 0000 1800 1800  NTLMSSP.........
0000010: 6400 0000 1800 1800 7c00 0000 0000 0000  d.......|.......
0000020: 4000 0000 0600 0600 4000 0000 1e00 1e00  @.......@.......
0000030: 4600 0000 0000 0000 0000 0000 0102 0000  F...............
0000040: 7400 6900 6d00 6700 6c00 6100 6d00 6400  t.i.m.g.l.a.m.d.
0000050: 7200 6900 6e00 6700 2e00 6c00 6f00 6300  r.i.n.g...l.o.c.
0000060: 6100 6c00 8f86 b036 e38a c5b8 f591 3b33  a.l....6......;3
0000070: 8af9 912e e303 7866 d94f 62d7 8f86 b036  ......xf.Ob....6
0000080: e38a c5b8 f591 3b33 8af9 912e e303 7866  ......;3......xf
0000090: d94f 62d7                                .Ob.
```

Listing 13: Complete Type 3 Response

NTLMv1 Type 3 Response Contents		
Field Name	Offset	Value
Domain Name	(n/a)	(null)
User Name	0000040	tim
Computer DNS Name	0000046	glamdring.local
NTLMv1 Response	0000064	8f86 b036 e38a c5b8 f591 3b33 8af9 912e e303 7866 d94f 62d7

Since the server has a copy of the user's NTLM (and LM) hashes, as well as a copy of the original challenge, it can perform the same calculations and compare the results. If they match, then the user entered the correct password, and authentication is successful.

3.3.2 NTLM v2

With NTLM v2 the response is computed very differently. The steps for this response include:

1. Calculate the user's NTLM password hash as before by converting the password to UTF-16 and making an MD4 hash.
2. Convert the username to uppercase and UTF-16, and then append the UTF-16 string of the authentication target (the domain or server name). If the user is authenticating from a non-windows device, this "target" may be blank.
3. Using the NTLM hash as a key and the username+domain string as the text, apply HMAC-MD5 to generate the NTLMv2 hash.
4. Create a binary blob including a special timestamp (tenths of a microsecond since January 1, 1601, as a little-endian 64-bit value), a random client nonce, and the target information block from the Type 2 message.
5. Prepend the server challenge to this blob.

6. Using the NTLMv2 hash computed in step 3 as the key and the (challenge + blob) as the message text, apply HMAC-MD5 to generate the response.
7. Returning to the binary blob (step 4, before prepending the challenge), prepend the response to the blob, to generate the final NTLMv2 response. This is then included in a Type 3 message (as described above) and returned to the server.

```
blob = ‹header› + ‹reserved› + ‹timestamp› + nonce + 0x00000000 + ‹target block›

NTLM_hash = MD4(UTF-16(password))

NTLMv2_hash = HMAC-MD5(text: UTF-16(UPPERCASE(username)+UTF-16(target), key:
    NTLM_hash))

challenge = ‹8-byte string from server›

NTLMv2_response = HMAC-MD5(text: challenge + blob, key: NTLMv2_hash)

final_response = ‹type 3 message› including (NTLMv2_response + blob)
```

Listing 14: Summary of the NTLMv2 Computation

The Type 1 and Type 2 messages are identical in format to the examples above, though for this example the challenge is now: ''115f4b6a06e258e0''. An example Type 3 response follows:

```
Authorization: NTLM TlRMTVNTUAADAAAAGAAYAFwAAACsAKwAdAAAAAA
AAABAAAAABgAGAEAAAAAWABYARgAAAAAAAAAAAAAAAQIAAHQAaQBtAFcATw
BSAEsAUwBUAEEAVABJAE8ATgChSoA1qSciJufTYYDi6uS9AfAPjPaXVXOEq
hhMKfdRRKIl8M33MlErAQEAAAAAAACAMtPKSEPRAaZ4E2XjzU+aAAAAAAIA
AAABABQAbQB5AGMAbwBtAHAAdQB0AGUAcgAEACgAbQB5AGQAbwBtAGEAaQB
uAC4AZQB4AGEAbQBwAGwAZQAuAGMAbwBtAAMALABtAHkAYwBvAG0AcAB1AH
QAZQByAC4AZQB4AGEAbQBwAGwAZQAuAGMAbwBtAAAAAAAAAAAAA

0000000: 4e54 4c4d 5353 5000 0300 0000 1800 1800  NTLMSSP.........
0000010: 5c00 0000 ac00 ac00 7400 0000 0000 0000  \.......t.......
0000020: 4000 0000 0600 0600 4000 0000 1600 1600  @.......@.......
0000030: 4600 0000 0000 0000 0000 0000 0102 0000  F...............
0000040: 7400 6900 6d00 5700 4f00 5200 4b00 5300  t.i.m.W.O.R.K.S.
0000050: 5400 4100 5400 4900 4f00 4e00 a14a 8035  T.A.T.I.O.N..J.5
0000060: a927 2226 e7d3 6180 e2ea e4bd 01f0 0f8c  .'"&..a.........
0000070: f697 5573 84aa 184c 29f7 5144 a225 f0cd  ..Us...L).QD.%..
0000080: f732 512b 0101 0000 0000 0000 8032 d3ca  .2Q+.........2..
0000090: 4843 d101 a678 1365 e3cd 4f9a 0000 0000  HC...x.e..O.....
00000a0: 0200 0000 0100 1400 6d00 7900 6300 6f00  ........m.y.c.o.
00000b0: 6d00 7000 7500 7400 6500 7200 0400 2800  m.p.u.t.e.r...(.
00000c0: 6d00 7900 6400 6f00 6d00 6100 6900 6e00  m.y.d.o.m.a.i.n.
00000d0: 2e00 6500 7800 6100 6d00 7000 6c00 6500  ..e.x.a.m.p.l.e.
00000e0: 2e00 6300 6f00 6d00 0300 2c00 6d00 7900  ..c.o.m...,.m.y.
00000f0: 6300 6f00 6d00 7000 7500 7400 6500 7200  c.o.m.p.u.t.e.r.
0000100: 2e00 6500 7800 6100 6d00 7000 6c00 6500  ..e.x.a.m.p.l.e.
0000110: 2e00 6300 6f00 6d00 0000 0000 0000 0000  ..c.o.m.........
```

Listing 15: NTLMv2 Type 3 Message

NTLMv2 Type 3 Response Contents		
Field Name	Offset	Value
NTLMv2 Response	0000074	84aa 184c 29f7 5144 a225 f0cd f732 512b
Blob header	0000084	01010000
Reserved	0000088	00000000
Timestamp	000008c	01d14348cade3280
Client Nonce	0000094	a678 1365 e3cd 4f9a
Target Block	00000a0	0200 0000 ....

The target block beginning at 00000a0 (''0200 0000...'') should match the target block provided in the server's Type 2 message, seen in the dump above beginning at location 000040.

Decoding the timestamp

1. Convert the timestamp from a little-endian byte string into a hex value: 0x01d14348cade3280
2. Convert to decimal: 130959844090720896
3. Divide by 10,000,000 (tenths of a microsecond / second): 13095984409
4. Subtract 11644473600 (seconds between the 1601 and 1970 epoch points): 1451510809
5. Convert this Unix-style timestamp to a modern date and time: 2015-12-30 16:26:49

We now have enough information to calculate the response.

```
NTLM hash:
52dac53d34d998ee55b1137d647e93ce (as above) Username: tim Domain target:
(null) Username + Target string: TIM NTLMv2 Hash:
ab774da35ccf4d3c9fc19cbb2829a6a8 Challenge: 115f4b6a06e258e0 Target block:
(see above) NTLMv2 response: 84aa184c29f75144a225f0cdf732512b
```

Listing 16: NTLMv2 Calculations

3.3.3 Assessment

- Reasonably strong, especially in the NTLMv2 version
- Use of nonce and timestamp help prevent replay attacks
- Does not send the plaintext password over network
- Not as frequently used for session credentials
- Quite complex and potentially vulnerable to implementation problems
- Not very well publicly documented
- An attacker with a user's NTLM or LAN Manager hash can use that to generate authenticated requests, without requiring the user's actual password[pas]

3.4 OAuth 1.0

Open Authorization (OAuth) began in 2007 as an OpenID implementation for Twitter, but quickly grew into its own standard, releasing Version 1.0 in April of 2010[RFC12a]. Strictly speaking, OAuth is not intended to provide a mechanism for authentication, but instead for authorizing access to a resource by a trusted 3rd party. However, in many applications this distinction is somewhat irrelevant. It is also frequently used for ''Federated Logins'', in which a system leverages the user base of popular social media or other sites, rather than maintaining its own database of usernames and passwords.

OAuth operates on a design involving three, not just two, participants: The end user, who owns the data, the resource provider, where the data resides, and a consumer, which is the application or service that is granted access to the user's data. Again, for many end-users there is very little distinction between ''logging into Twitter'' and simply ''granting the Twitter app access to my account,'' though it really is the former, and not the latter, that occurs using OAuth.

Several good descriptions of OAuth can be found online, including ''OAuth for Dummies''[Tra09] and ''Dancing with OAuth''[Oh12]. These provide very in-depth exploration of the protocol as well as further clarification of the differences between OpenID, OAuth, and the ''OAuth 1.0 Authentication Sandbox''[oau], which allows for easy exploration of the various parameters and signatures.

The proliferation of various sites explaining OAuth is an indication as to how complex and confusing these standards can be. The following example greatly simplifies the specification, but should be enough to understand the most common uses encountered using modern applications.

1. A user needs to access a service: ''Tim wants to connect his Angry Rabbit game with the application's online community.''
2. The Angry Rabbit application (the consumer) connects to rabbit.com (the provider) and asks for a request token. The server responds with the token, and an associated secret, both of which are temporarily retained by the consumer.
3. The consumer gives the token to the user and redirects them to the provider, usually through a web form.
4. The provider asks the user to verify themselves by logging in, then displays the consumer's request to the user ''Do you want to connect the Angry Rabbit game to your online account?''
5. The user confirms the request, thereby *authorizing* the consumer to have access to their account.
6. The consumer then shows the provider the previously received request token and demonstrates possession of the secret through a digital signature. The consumer then receives an access token and secret and may now discard the request token and secret.
7. All future interactions use the access token and secret, allowing the consumer to access the provider's service on the user's behalf.

The following sections provide a detailed example of a full authorization request and subsequent resource access.

3.4.1 Client Setup

When building an application which will authenticate with a remote service using OAuth, the developer must first register that application with the service. This registers the developer or application itself, across multiple versions and platforms, and will be the same for all installations of that application. This registration

process provides a developer with a "Consumer Key" (or "Client ID") and "Consumer Secret".

3.4.2 User Registration: Obtaining Access Token and Secret

A user wishes to authenticate to a service using the application, over OAuth. The application will already possess the Consumer Key and Consumer Secret as well as the proper URL to connect to as defined by the remote server's interface specifications. The application will select a signature method, note the current time, and create a random nonce.

OAuth Example Data		
Description	Field Name	Value
Consumer Key	oauth_consumer_key	QkNDREItQjNDRC00NkNGL
Consumer Secret	(n/a)	OEUyM0MtQUQzNi00Q0M4L
Signature Algorithm	oauth_signature_method	HMAC-SHA1
Timestamp	oauth_timestamp	1453030020
Random Nonce	oauth_nonce	MTZBNDAtMTNBNy00MEQwL
Callback URL	oauth_callback	oob
URL	(n/a)	https://rabbit.com/login

1. The client application asks Rabbit for a request token. The request includes the `oauth_consumer_key`, `oauth_signature_method`, `oauth_timestamp`, `oauth_nonce`, `oauth_version` (optional, must be 1.0), and `oauth_callback`. In this example the callback URL is "oob," indicating that the user must manually visit a web page to complete authentication. Supported digital signature methods include plaintext, HMAC-SHA1, and RSA-SHA1.

2. The request string is normalized (sorted, UTF-8 encoded, URL-encoded, and concatenated into a single string with '&' separators). The request method and URL-encoded URL are added to the beginning.

```
GET&https%3A%2F%2Frabbit.com%2Flogin&oauth_callback%3Doob%26oauth_consumer_key%3
    DQkNDREItQjNDRC00NkNGL%26oauth_nonce%3DMTZBNDAtMTNBNy00MEQwL%26
    oauth_signature_method%3DHMAC-SHA1%26oauth_timestamp%3D1453030020%26
    oauth_version%3D1.0
```

Listing 17: Building the Request String

3. The client application signs the request, using a signing key built from the consumer and token secrets, separated by an ampersand. For the initial request, this is just "(consumer)&", as the client has not yet received an access token or secret.

```
Signing key = OEUyM0MtQUQzNi00Q0M4L&

base64( HMAC-SHA1(request-string, signing-key) ) = xTAwVGGUGjT1ViJH5EQtPKeRn50%3D
```

Listing 18: Signing the Request

4. The request parameters and signature are incorporated into the HTTP request.

```
GET /login HTTP/1.1
Host: rabbit.com:443
Authorization: OAuth realm="https://rabbit.com/login",
```

```
  oauth_callback="oob",
  oauth_consumer_key="QkNDREItQjNDRC00NkNGL",
  oauth_nonce="MTZBNDAtMTNBNy00MEQwL",
  oauth_signature_method="HMAC-SHA1",
  oauth_timestamp="1453030020",
  oauth_version="1.0",
  oauth_signature="PqQOcOG2wgVijjjHtgtxYNDMmZk%3D"
```

Listing 19: OAuth Authorization Header

5. The server responds with the "Unauthorized Request Token and Secret": `oauth_token` (in our example, it returns "NDdDOTUtNzAxRC00MDIzL"and `oauth_token_secret`, as well as `oauth_callback_confirmed=true`, and additional parameters.

6. The application redirects the user to the service provider's authorization URL. A simple HTTP GET is sent with only the `oauth_token` parameter (received in the previous step). The remote service then authenticates the user, such as the user logging into the online site, and obtains consent for the client authorization. If user interaction is required due to an out-of-band callback, they should receive a human-friendly verifier code, frequently a multi-digit number. Otherwise, the code is returned as the `oauth_verifier` parameter.

```
https://auth.rabbit.com/oauth/authenticate?oauth_token=NDdDOTUtNzAxRC00MDIzL

"Please return to the application and enter the pin 6066323 when prompted."
```

Listing 20: URL Enabling User to Authorize Request

7. The service then creates an OAuth request, as before, with the additional `oauth_token` parameter. `oauth_verifier` is provided as a GET or POST variable using the verification code received in the previous step. The browser then is redirected to that URL which will provide the access token. The signature now includes the `oauth_token_secret` (received in step 5) as part of the signing key.

8. The service provider then returns a long-term access token containing new `oauth_token` and `oauth_token_secret`, and any additional parameters. These credentials are retained by the client application for all future requests. The temporary token and secret received in step 5 may now be discarded.

3.4.3 Signing Individual Requests

Here is an example of an individual request signed using OAuth credentials. In this example, the above parameters have been reused, with a different URL and additional request variables:

OAuth Example Data		
Description	Field Name	Value
Client ID	oauth_consumer_key	QkNDREItQjNDRC00NkNGL
Client Secret	(n/a)	OEUyM0MtQUQzNi00Q0M4L
Signature Algorithm	oauth_signature_method	HMAC-SHA1
Timestamp	oauth_timestamp	1452029056
Random Nonce	oauth_nonce	OUIwRDItMzhGMi00MzI4L
URL	(n/a)	https://rabbit.com/lookup
GET variable	record	cave
GET variable	detail	3
Access Token	oauth_token	NDdDOTUtNzAxRC00MDIzL
Access Secret	(n/a)	RTFCOUYtMTU2Ni00N0JDL

As before, the parameters, key, token, nonce, timestamp, signature method, and version strings are UTF-8 encoded, URL-encoded, sorted, and combined into a URL parameter-like string. The HTTP method and URL are prepended, and the entire string URL encoded.

```
GET&https%3A%2F%2Frabbit.com%2Flookup&detail%3D3%26oauth_consumer_key%3
    DQkNDREItQjNDRC00NkNGL%26oauth_nonce%3DOUIwRDItMzhGMi00MzI4L%26
    oauth_signature_method%3DHMAC-SHA1%26oauth_timestamp%3D1452029056%26oauth_token%3
    DNDdDOTUtNzAxRC00MDIzL%26oauth_version%3D1.0%26record%3Dcave
```

Listing 21: Final Request String

A signing key is created by concatenating (after UTF-8 and URL encoding) the consumer secret and access token secret, separated by a single &:

```
OEUyM0MtQUQzNi00Q0M4L&RTFCOUYtMTU2Ni00N0JDL

base64( HMAC-SHA1( request-string, signing-key ) ) = Ntzq3+K1XEyYep1uksdtc/pXWnk=
```

Listing 22: Signing Key

Finally, the parameters are assembled into an OAuth authorization request, which can be given in the URL, as POST parameters, or as an HTTP header:

```
GET /lookup?record=cave&detail=3 HTTP/1.1
Host: rabbit.com:443
Authorization: OAuth realm="https://rabbit.com/lookup",
  detail=3,
  oauth_consumer_key="QkNDREItQjNDRC00NkNGL",
  oauth_nonce="DOUIwRDItMzhGMi00MzI4L",
  oauth_signature_method="HMAC-SHA1",
  oauth_timestamp="1452029056",
  oauth_token="NDdDOTUtNzAxRC00MDIzL",
  oauth_version="1.0",
  record="cave",
  oauth_signature="Ntzq3%2BK1XEyYep1uksdtc%2FpXWnk%3D"
```

Listing 23: OAuth Authorization Header

3.4.4 Securely Storing User Secrets

Careful server-side design may be able to mitigate the effects of a server compromise. A compromise of a server that stores both the user's access token and secret key will directly compromise all users whose keys are stored on the server, as the attacker need only transfer those credentials to their own system. However, if the server does not retain a plain-text copy of the access token, such an attack can be prevented. This can be accomplished by retaining only a hashed copy of the token:

- A user's record includes their secret key and a hash of the access token
- When the user's mobile application connects to the service, it provides the plaintext access token as part of the connection
- The server hashes that token and uses it to look up the appropriate user record (or uses the provided userid, and simply verifies that the hashed value is also present)
- If the hashed value matches the value on record, the server can continue with the authentication as normal

Additional recommendations for securing OAuth servers can be found in the NCC Group paper by Paul Youn[You11].

3.4.5 Assessment

- A very strong system: includes nonces, timestamps, and digital signatures.
- Stores only password-equivalent credentials for each authorized application.
- Once authorization is complete and the client has received their tokens, no further credentials are actually exchanged over the network.
- Can provide integrity controls on individual requests.
- Somewhat complicated, and thus can be difficult to understand.
- Does not actually provide the authentication step – OAuth relies on the service provider's authentication system, which may be weakly implemented.
- Compromise of the server that results in disclosure of the access token and secret gives attackers direct access.

3.5 OAuth 2.0

In 2010, work on a subset of OAuth called the OAuth Web Resources Authorization Protocol (WRAP)[wra10] was re-focused as a new version of OAuth. While still sharing the OAuth name, the new framework is not backward compatible with OAuth 1 and differs in significant ways. The first elements of OAuth 2.0 were formally published in 2012[RFC12b].

In some ways OAuth 2 greatly simplified the OAuth protocol, but it also make some parts more complicated or less secure. It largely drops the signatures used in OAuth 1 and instead relies on the security of the transport layer. Furthermore, where OAuth 1 defined a fairly strict protocol, OAuth 2 is intended as a framework, which services may extend in different ways.

The basic flow is still very similar to OAuth 1:

- The Client requests authorization from a Resource Owner
- The Resource Owner responds with a grant
- The Client sends the grant to the Authorization Server
- The Authorization Server responds with an Access Token
- The Client uses the Access Token to request access to a protected resource on the Resource Server

The Resource Owner responds with an Authorization Grant, the specific type of which is related to the client and overall flow required.

- **Authorization Code**: The flow closest to the OAuth 1 approach, where the Authorizing server and the Resource Server are separate entities.
- **Implicit**: A simplified flow, typically targeted at browser clients. This flow still makes use of separate authentication steps, but avoids the step where the client exchanges a grant for an access token. Instead, the access token is returned to the client immediately after authentication.
- **Resource Owner Password Credentials**: In this flow, the user's credentials are directly exchanged with the resource owner for the access token.
- **Client Credentials**: Similar to Resource Owner Password Credentials, but for authenticating the client directly to the service (as when the client application is itself the resource owner).

The access tokens returned by the server are intended to be of limited duration, though enforcement of a specific time limit is up to the server's implementation. Typically the time limit is measured in hours rather than days. When a request is made using an expired token, the server indicates the error, and the client may use a "refresh token" to request a new access token. The refresh token may have been returned upon completion of the initial authorization or may be provided in response to specific requests to the server. The longevity of the refresh token is undefined. Some services leave them unlimited, others having them expire after a set period of weeks or months, some after a long period of account inactivity, and others after a pre-set number of uses.

As with OAuth 1, a multitude of references are available to explain the various flows and grants. Two references proved useful in the writing of this paper: the first, written by Alex Bilbie[Bil13], and the second, by Mitchell Anicas[Ani].

3.5.1 Obtaining and Using Access Tokens

As an example, the application connection process for GitHub will be described. GitHub supports both web-based and app-based scenarios for OAuth2[git16]. Assuming the application to be connected to GitHub is web-based, the following steps need to be taken to authorize the app on behalf of the GitHub user:

1. Redirect the user's browser to GitHub to get an authorization token.

```
GET https://github.com/login/oauth/authorize

parameters:
  * client_id - Identifier for the application, issued by GitHub
  * redirect_uri - URL to which the user will be redirected after authorization
  * scope - An array of strings that identifies the permissions the application
```

```
    is requesting
  * state - A random string to protect against cross-site request forgeries (CSRF
    )
```

Listing 24: Request Authorization

2. The user then authenticates using their GitHub userid, password, and authorizes the application's access to their account
3. GitHub redirects the browser to the requesting application using the `redirect_uri` parameter provided earlier, resulting in a new authorization code in the `code` parameter.
4. The web application then uses the authorization code to request an access token:

```
POST https://github.com/login/oauth/access_token

parameters:
  * client_id and client_secret: The client identification code and secret key
    assigned by GitHub to this application
  * code - The authorization code just received from GitHub
  * redirect_uri - The page in the local application to which GitHub will
    redirect upon successful issuance of the access token
  * state - As before, the anti-CSRF token
```

Listing 25: Request Access Token

5. GitHub then responds with an access token. The format of the response is dictated by the "Accept:" header provided in the last request:

```
[Accept: application/json]

{"access_token":"e85ecf8fc409714a3c28fbd205f8567de0521e57", "scope":"repo,gist",
   "token_type":"bearer"}
```

Listing 26: Access Token Response

3.5.2 Use of Access Token in Session

The application may now use the `access_token` to request protected resources from the API:

```
GET https://api.github.com/api_command/etc
Authorization: token e85ecf8fc409714a3c28fbd205f8567de0521e57
```

Listing 27: Session Requests

Note that just the session token is enough to get access. Signing of the request to verify its authenticity is not required. Also, the client ID and key are no longer used, which means that the token can be transferred to other clients using the same system. The token must therefore be stored carefully, such as in a secure system keychain.

3.5.3 Assessment

- OAuth 2 is more of a general framework of an authorization system, making it open to interpretation and extension by implementers.

- Because of the openness of the framework, interoperability across systems may be limited.
- The lack of timestamps, nonces, and client signatures means that session tokens are universally transferable and reusable. Compromise of an access token provides the attacker with full control of the account to whatever limits were imposed by the scope at token issuance.
- Despite limitations, OAuth 2 can still be useful for both for client applications and as a sort of ''universal session token'' to allow applications to access other services.

3.6 Other Systems

The authentication systems described thus far are all very commonly used, and supported by most, if not all, popular browsers in use today. This section describes a few other systems, at a high level, which the reader may find of interest. Detailed analysis and assessments of the qualities of these systems is beyond the scope of this paper.

3.6.1 FIDO Alliance U2F

The Fast IDentification Online (FIDO) Alliance has published two specifications: The Universal Authentication Framework (UAF) and the Universal Second-Factor (U2F) protocols[All]. Both protocols are designed to strengthen or replace the usual password-based login process, and neither is suited to ongoing session authentication.

UAF can replace the password-based login experience by integrating access to local mechanisms, such as biometrics (finger prints, facial recognition) or locally entered PINs. U2F augments the password process, making use of strong second factor devices, such as a USB- or NFC-connected key.

U2F was developed by Google and Yubico, but management of the standard has now been transferred to FIDO. Integrated support for U2F is currently present only in the Chrome browser.

At its most basic level, the U2F system allows a remote server to issue a challenge to the local hardware device, which then signs the challenge and returns it to the server. The server then verifies that the signature is legitimate, and the authentication is completed.

In order to provide a variety of additional features and protections, the protocol is slightly more complicated than that. The following outline describes most of the key features:

1. The user first registers a hardware device (such as a Yubikey, we'll call it ''u2fkey'' here) with the remote server
2. Upon subsequently revisiting the server, the user enters their userid and password, and the server validates it (this can be accomplished using any of the traditional authentication systems described throughout this paper)
3. If the password is verified, then the server sends a challenge to the browser
4. In addition to the challenge, the server provides an application ID and a handle, which together specify which identity on the u2fkey is to be challenged. This permits the same u2fkey to be used for multiple accounts on the same service.
5. The browser appends the URI origin, and, optionally, the TLS Channel ID, to the challenge. This helps to minimize the potential for a phishing attack where the origin does not match the legitimate server's origin. The challenge and other data are then sent to the hardware key.

6. The u2fkey locates the correct identity for the given information *(handle, application)*, and retrieves an internal counter specific to the application and handle. This helps to prevent cloning of a hardware key, by permitting the server an opportunity to reject re-used counter values.

7. The u2fkey then combines the application ID, the challenge (server challenge, origin, and channel ID), and the counter, into a single message, and signs it with the private key associated with the *(handle, application)* identity.

8. The u2fkey returns the counter value and the signature to the browser, which then forwards the response to the server. The server verifies that the counter value has not yet been used, and verifies the signature on the response, using the public key stored on the server during u2fkey registration. The u2fkey also increments the locally stored counter for the service.

The specific approach to handling the details of the hardware key are left up to the vendor. Yubico, for example, uses the handle and application ID to dynamically generate a private / public key pair each time the device is challenged, minimizing the requirements for storage of multiple keys on the small device[Yub].

3.6.2 OpenID

The OpenID system bears many similarities to OAuth, which was initially begun as a Twitter-specific OpenID implementation. The protocol permits decentralized 3rd-party servers to authenticate users to individual services. In this way, a user of OpenID may simply use the same OpenID account to register and authenticate themselves to many different systems on the Internet.

OpenID was originally developed in 2005, and became somewhat popular for some time. In recent years, the dominance of the protocol has faded, being largely replaced with direct federated logins against popular social media sites (for example, "Use your Twitter account to log in!"). The most recent version of OpenID, published in early 2014, is called OpenID connect, and is built atop the OAuth 2 framework[Fou15].

3.6.3 CRAM-MD5

Though not usually seen in web or mobile applications, CRAM-MD5 is a part of the Simple Authentication and Security Layer (SASL) specification and has been frequently used within mail protocols, such as SMTP, POP, and IMAP authentication[sas06]. The protocol is a "Challenge Response Authentication Mechanism" (CRAM), and uses HMAC-MD5 to produce the response.

This system has serious weaknesses and should no longer be used, but is presented here for historical interest and as another example of a challenge-response system.

1. The server sends a random string to the client as a challenge. The SASL specification requires that the string be a Message-ID email header, including random digits, timestamp, and the server's domain name. The string is then Base-64 encoded and sent to the client.

```
Server: 220 smtp.server.com Simple Mail Transfer Service Ready
Client: EHLO client.example.com
Server: 250-smtp.server.com Hello client.example.com
Server: 250-SIZE 1000000
Server: 250 AUTH LOGIN PLAIN CRAM-MD5
Client: AUTH CRAM-MD5
Server: 334 PDI2MzcyNzA1MjM2NzI3MC4xNDUyMTc5OTI4LnNtdHAuc2VydmVyLmNvbT4=
```

Listing 28: SMTP Server Challenge

2. The client then decodes the challenge, and computes an HMAC-MD5 hash on the challenge contents,

using the user's password as the key:

```
base64-decode(challenge) = "<263727052367270.1452179928.smtp.server.com>"

HMAC-MD5('Enchanter', challenge-text) = c2fd6616011d71af3fe82c17a3baa309

response = base64('tim c2fd6616011d71af3fe82c17a3baa309')
  ("dGltIGMyZmQ2NjE2MDExZDcxYWYzZmU4MmMxN2EzYmFhMzA5")
```

Listing 29: Computing Response

3. Finally, the client returns this response to the server, completing the authentication process.

```
Client: dGltIGMyZmQ2NjE2MDExZDcxYWYzZmU4MmMxN2EzYmFhMzA5

Server: 235 2.7.0 Authentication successful
```

Listing 30: Completing Challenge

4. The server computes the same response using its copy of the password. If they match, the challenge was successful.

The CRAM-MD5 algorithm has some serious weaknesses, however.

- If the password is used directly in the HMAC-MD5 step, then the server must retain a plaintext copy of that password in order to validate the response (though some workarounds are possible).
- An attacker who is able to observe the transaction may crack the user's password using an offline brute-force attack.
- An attacker may pre-compute the most popular passwords using a known challenge. If that attacker can intercept the authentication transaction and substitute their known challenge to the client, the password may be recovered far more quickly than using brute force methods.

It is generally recommended that CRAM-MD5 be avoided whenever possible.

3.6.4 X.509 and PKI

The use of X.509 certificates and private keys, signed by trusted Certificate Authorities in either a private or public key infrastructure, is another strong method of authenticating a user. However, since the private keys are inappropriate for personal memorization by the user, they're saved on the user's device and presented when connecting to the remote server. In this way, they more technically authenticate the client, and not the person using it.

To add more security (and to make the authentication more personal, and less client-oriented) it may be advisable to store the private key in encrypted form, and use a user-entered password to decrypt it when the application is launched.

This approach has several drawbacks:

- Significant overhead of maintaining large numbers of X.509 certificates on the server side, which may be significantly larger than even the largest password hashes in common use today.
- Every device (and potentially every application) must be individually registered with the remote service, unless a secure method of distributing copies of the private key is used. Such methods are theoretically

possible, but in practice may introduce additional significant vulnerabilities both in the procedure and the technical implementation.

- Initially associating the private / public key pair with a user requires an additional authentication step, much as requesting an OAuth token requires authentication outside the scope of that particular protocol.

The author has rarely seen certificates used for authentication in consumer-level mobile applications, and even then, the use has been to add more layers of security to an already-authenticated channel. For example, a device-specific private key may be established when the device is registered for use with a particular account. This key may then be used to additionally encrypt or authenticate traffic between the device and the server, further frustrating any attempts at man-in-the-middle attacks.

3.6.5 JSON Web Token

The JSON Web Token (JWT) isn't an authentication system itself, but is a way to structure and sign data for use in session tokens or other cookie-like systems.[RFC15]

The token consists of three Base-64 encoded JSON strings, separated by a "." character. The strings are defined as follows:

- Header: Defines the token type and hashing algorithm. In this example, "JWT" and HMAC-"SHA256." Other valid choices for the algorithm include "None," "RSASSA-PKCS1-V1_5 with SHA-256 (RS256)," and "ECDSA P-256 with SHA 256 (ES256)."

```
{"typ":"JWT",
 "alg":"HS256"}

Base-64 encoded: eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9
```

Listing 31: JWT Header

- Payload, or "JWT Claims Set": Includes the specific data being encoded within the token.

```
{"user": "tim",
  "is_wizard": true}

Base-64 encoded: eyJ1c2VyIjoidGltIiwiaXNfd2l6YXJkIjp0cnVlfQ==
```

Listing 32: JWT Payload

- Signature: Uses the chosen algorithm, with a shared secret as the key and the Base-64 encoded payload string as the message.

```
HMAC-SHA256('secret', 'eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.
    eyJ1c2VyIjoidGltIiwiaXNfd2l6YXJkIjp0cnVlfQ==')

  IVliVUihAm1B7ZIX2xk8FaMMlavQNPBbz7y33vSItMg=
```

Listing 33: Computing JWT Signature

The final token is then these three strings concatenated with "." characters:

```
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9 + "." +
```

```
eyJ1c2VyIjoidGltIiwiaXNfd2l6YXJkIjp0cnVlfQ== + "." +
IVliVUihAm1B7ZIX2xk8FaMMlavQNPBbz7y33vSItMg=
```

Listing 34: Concatenating the JWT components

This may then be returned to the server through any available channel, such as a cookie, an HTTP header, or a POST body variable.

4 Summary of System Capabilities

The authentication systems described in this paper do not cover every single possible approach, but they do represent a significant cross-section of methods in common use today. The author has personally tested applications using every method described here, with the exception of OpenID and FIDO systems.

At this point, we have reviewed enough to be able to make some generalizations about such systems, what commonly seen elements they use, and the overall safety and security of each. The following table lists the key elements used by authentication systems today.

Elements of Authentication Systems	
Element	Description
Password	Presented by the user as proof of shared knowledge, which permits the authentication of that user.
Signature	Used with a shared key to authenticate a piece of data. Allows for proof of shared knowledge without requiring the disclosure of the actual shared information.
Nonce	Ensures that each authentication exchange is unique. Helps to prevent replay attacks.
Timestamp	Ensures that old transactions cannot be reused. Helps to prevent replay attacks.
PKI	Public key cryptography can be used for very strong encryption and authentication of data. If private keys are properly secured, can be made extremely difficult to clone and attack.
Integrity Protections	Ensures that individual requests cannot be altered, by tying the contents of the request to an authentication-based signature.

The following tables indicate which of the above elements are used by the authentication systems described in this paper, and overall security-relevant features of each.

Use of Common Elements by Popular Authentication Systems						
System	Password	Signature	Nonce	Timestamp	PKI	Integrity
Password	yes					
Digest	yes	yes	yes	optional		optional
NTLM	yes	yes	yes	yes		
OAuth 1	n/a	yes	yes	yes		yes
OAuth 2	optional					
FIDO U2F	n/a	yes	optional	optional	yes	

Security-Relevant Features					
Feature or Attribute	Password	Digest	NTLM	OAuth 1	OAuth 2
Password Sent at Login	yes	no	no	n/a	optional
Password Required to Re-establish Session	yes	yes	yes	no	no
Password Sent During Session	sometimes	no	no	no	no
Unlimited Session Tokens	yes	no	no	no	yes
One-Time Session Tokens	no	yes	yes	yes	no
Detection of Request Modification	no	optional	no	yes	no

Notes:

- Some systems use passwords instead of session tokens, and send the user's password with every request.
- OAuth (especially OAuth 1) and U2F rely on a separate authentication step that generally includes use of passwords, but may rely on other factors instead.
- Additionally, OAuth 1 can protect the integrity of request contents. That is, it can include a digital signature of the actual request to the server, which protects against an attacker intercepting communication and maliciously modifying the request.
- A password-based system that replays the user's password for each packet (instead of using a dedicated session token) is at risk for password theft during an established session, in addition to during login.
- For OAuth 1, the registered client or application retains a long-lived access token, which, along with the associated secret and client token and secret, functions equivalently to a password. However, this token can be revoked by the user on a per-application basis.
- NTLM is not generally used for session-based authentication.
- U2F provides for one-time-use authentication responses, which cannot be reused. As this occurs only at initial login, it cannot specifically provide integrity protections for the request contents.

5 Other Considerations

It's tempting to analyze authentication protocols at a theoretical level, develop the absolutely most-secure, theoretically unbreakable system, and decree by regulatory fiat that all authentication henceforth must use that approach. However, the real world cannot work that way, and each system may have different drawbacks or limitations that make ubiquitous use impossible. These impacts may be conveniently divided and discussed based on their impact to servers, clients, and developers.

5.1 Server

5.1.1 Performance

The choice of password hashes in use by a system may impact the performance of that system.

For example, a protocol specifically designed for use with mobile applications may choose to have both client and server hash the user's password with 10,000 iterations of a strong hash, producing a result that is, at least in practical terms, uncrackable, and which can be stored on the server with confidence.

However, if the user is to send a plaintext password to the server, and then rely on the server to perform the 10,000 iteration password hash, such a design may impose a significant load on the server when load grows to the tens of thousands, or even millions, of users. This may be mitigated by moving authentication processing to a separate machine, but if this intensive calculation is to be performed while verifying every request, such a division of labor may not be possible. Thus, it is important to balance server requirements when designing an authentication system.

5.1.2 Flexibility

Another important consideration is whether the choice of authentication method locks the system into that method, with no easy way to migrate to a different system. In some cases, migrations may be handled transparently to the user.

For example, if the system is to migrate user password hashes from SHA1 to scrypt, the system (upon receipt of a user's plaintext password) could calculate the SHA1 hash of the password, and upon verification, replace it with the equivalent scrypt-based hash. However, if the system in use does not provide the user's password to the server, then such a migration would require additional steps and, possibly, user interaction.

It may not always be possible to design a strong, secure system with support for potential, unimagined future technology, but it would be advisable to remain open to that possibility when designing systems.

5.2 Clients

Similar, but different, calculations must be made when considering the effect of various authentication systems on clients. In general, performance calculations aren't as important, as the load presented by (for example) exceptionally strong hash functions, is infrequent and may be masked by normal variations in network latency. However, the basic availability of certain technologies on the clients may not always be guaranteed.

5.2.1 Technical Capabilities

Certainly, where a system may depend on biometrics, the capabilities of the client will be a deciding factor. But what may be less obvious is the availability of certain functions on remote devices. For example, web browsers only natively support Basic (passwords), Digest, and NTLM methods (though U2F is gaining acceptance). Therefore, even if the back-end servers support the latest and most powerful systems, any web-based client will frequently fall back to simple password and randomized session tokens.

5.2.2 Differing Feature Scope

Not only, then, is it important to include support for weaker clients such as browsers, it is also important to ensure that such support doesn't diminish the security offered to more full-featured dedicated mobile applications. Tokens issued to a mobile application shouldn't be accepted by a server API aimed at browsers, and the browser tokens shouldn't unlock API functions protected by a better system. In practice, both application and browser APIs will likely be equally full-featured, but there may be considerations where this is not the case. For example, consider a simple web-based interface and a more capable mobile application. If the session tokens issued to the web app are easily compromised, and may be reused on the mobile application, than an attacker can leverage the more easily obtained credential into a more powerful position with the mobile app.

5.3 Developers

When presented with the need to implement a strong protocol, and a client (such as a web browser) which doesn't support that protocol, the first inclination may be to simply implement the protocol as part of the client code. If the missing code is something simple, like a Base-64 encoder, this may be an acceptable plan. If, however, more complex cryptographic primitives are needed, then self-implementation is a bad, and potentially dangerous, plan of action.

Whenever possible, developers should seek out and use known, trusted, and actively maintained libraries for the more advanced cryptographic functions. Pre-built libraries are available for hash generation, symmetric and public key cryptography, HMAC functions, and so forth, for just about every programming language, operating system, and mobile platform currently on the market. In many cases, these libraries are available as part of the operating system itself.

Unfortunately, implementing such functions in a browser-based application, even with known libraries, is dangerously risky. Because the code run within a web application is all part of the web app itself, any incorporated JavaScript code may not be fully trustworthy. Attacks on network infrastructure may be able to easily substitute maliciously modified code within the web page, to subtly (or drastically) alter the results of such operations, making any protections afforded by the protocol totally moot. It is far better, for web-based applications, to rely on primitives and functions offered directly by the browser, however limited those options may be.[Pta11]

However, it remains important for developers to avoid writing all aspects of their systems to the lowest common denominator. If a system is to be supported by both browser based and native mobile applications, there is no reason that the mobile apps should be restricted to whatever authentication system the web client uses. In fact, the dedicated mobile application offers the greatest potential for integrated security, as the system owns both sides of the conversation: server and client software.

A mobile application is the ideal place to implement the strongest possible security measures – strong initial login, one-time session tokens, request integrity controls. Unfortunately, the vast majority of iOS applications reviewed by this author use simple passwords for login and basic tokens (frequently OAuth 2 based) for session authentication.[Sch15] The majority of those applications also insecurely stored session tokens (and nearly half insecurely stored passwords).

Results of iOS Application Authentication Survey				
Credential Type	Example	Unsafe Count	App Count	% Unsafe
Login Passwords	Sent at login in cleartext	35	38	92%
Session Credentials	Password sent with each request	2	38	5%
	Static token sent with each request	28	38	74%
Passwords	Stored outside secure keychain	4	9	44%
Tokens		27	40	66%

Note: Of the 38 applications covered in the referenced survey, only one used unencrypted communications, and only one failed to notice a man-in-the-middle attempt using an untrusted certificate. Two applications (plus another two discarded from the survey), for a total of 4 out of 40 applications, made use of certificate pinning.

Clearly, the security of the credentials in transit is not the only problem we face today.

6 Conclusion

A multitude of different options are available for web and mobile application client / server authentication. The vast majority of systems in general consumer use today select from only five systems: Password or HTTP Basic Authentication, Digest, NTLM, OAuth 1, and OAuth 2. Some of these systems are inherently unsafe in the absence of a reliably encrypted transport layer. Others can offer protections even over totally unencrypted channels, but with significant limitations such as initial login security or implementation complexity.

The most secure systems (OAuth 1 and newer systems such as U2F) are not natively supported by browsers today, and so the choices available to web-based applications are generally limited when compared to the capabilities of dedicated, native mobile applications. However, experience has shown that even these mobile applications, more often than not, choose easier, and less secure, techniques.

The situation can be improved somewhat with better browser support for stronger authentication protocols, and even for dedicated implementation of advanced cryptographic primitives. The speculation and design of such an improved state of affairs is beyond the scope of this paper, however, the recent addition of the FIDO Alliance U2F protocol as a natively implemented authentication system in Chrome browsers gives some hope for the future.

It is hoped that this paper may be useful as a reference for developers and security professionals alike. The five protocols discussed in-depth are in use by the vast majority of applications today, and being able to properly understand these protocols is vital to the development and deployment of secure systems.

However, the examples here, more often than not, only scratch the surface. For the most authoritative reference, original specification documents should always be consulted. A listing of RFC and other specification documents, as well as helpful explanations and descriptions written for development and security audiences, is provided in the References section.

References

[All] FIDO Alliance. Specifications overview. URL: https://fidoalliance.org/specifications/overview/. 25

[Ani] Mitchell Anicas. An introduction to oauth 2. URL: https://www.digitalocean.com/community/tutorials/an-introduction-to-oauth-2. 23

[Bil13] Alex Bilbie. A guide to oauth 2 grants, 2013. URL: http://alexbilbie.com/2013/02/a-guide-to-oauth-2-grants/. 23

[Fou15] OpenID Foundation. Specifications and developer information, 2015. URL: http://openid.net/developers/specs/. 26

[git16] Oauth - github developer guide, 2016. URL: https://developer.github.com/v3/oauth/. 23

[Gla06] Eric Glass. The ntlm authentication protocol and security support provider, 2006. URL: http://davenport.sourceforge.net/ntlm.html. 13

[oau] Oauth 1.0 authentication sandbox. URL: http://nouncer.com/oauth/authentication.html. 18

[Oh12] Changhun Oh. Dancing with oauth: Understanding how authorization works, 2012. URL: http://www.cubrid.org/blog/dev-platform/dancing-with-oauth-understanding-how-authorization-works/. 18

[pas] Pass the hash. URL: https://en.wikipedia.org/wiki/Pass_the_hash. 17

[Pta11] Thomas Ptacek. Javascript cryptography considered harmful, 2011. URL: https://www.nccgroup.trust/us/about-us/newsroom-and-events/blog/2011/august/javascript-cryptography-considered-harmful/. 33

[pyt10] python-digest, 2010. URL: https://bitbucket.org/akoha/python-digest/wiki/Home. 11

[RFC99] Http authentication: Basic and digest access authentication, 1999. URL: http://tools.ietf.org/html/rfc2617. 11

[RFC12a] The oauth 1.0 protocol, 2012. URL: https://tools.ietf.org/html/rfc5849. 18

[RFC12b] The oauth 2.0 authorization framework, 2012. URL: https://tools.ietf.org/html/rfc6749. 22

[RFC15] Json web token (jwt), 2015. URL: https://tools.ietf.org/html/rfc7519. 28

[sas06] Simple authentication and security layer (sasl), 2006. URL: https://tools.ietf.org/html/rfc4422. 26

[Sch15] David Schuetz. Knock knock: A survey of ios authentication methods, 2015. URL: http://darthnull.org/2015/01/23/shmoocon-ios-auth. 33

[Sch16] David Schuetz. My hash is my passport - shmoocon talk, 2016. URL: http://darthnull.org/2016/01/17/hashpassport. 3

[Tra09] Mark Trapp. Oauth for dummies, 2009. URL: https://marktrapp.com/blog/2009/09/17/oauth-dummies. 18

[Tsc03] Ronald Tschalär. Ntlm authentication scheme for http, 2003. URL: http://www.innovation.ch/personal/ronald/ntlm.html. 13

[wra10] Oauth web resource authorization protocol (wrap), 2010. URL: http://wiki.oauth.net/w/page/12238537/OAuth+WRAP. 22

[You11] Paul Youn. Creating a safer oauth user experience, 2011. URL: https://www.nccgroup.trust/us/our-research/creating-a-safer-oauth-user-experience/. 22

[Yub] Yubico. U2f key generation. URL: https://developers.yubico.com/U2F/Protocol_details/Key_generation.html. 26