My Hash Is My Passport

Understanding Web and Mobile Authentication

ShmooCon - January 17, 2016

David Schuetz

Senior Consultant, NCC Group

@DarthNull

Introduction

Agenda

- Background
- Definitions and Goals
- Commonly Used Authentication Systems
- Other Systems
- Overall Evaluation
- Response Builder Tool
- Conclusion

Background

Background

- You enter your password in a web application.
- What happens next?

Authentication Happens

- The server decides whether you are you
- How does your password get to the server?

- “That Depends”

- Many different methods in use today
- Close look at the five most common ones

Why do we care?

- There's a right way, and a wrong way
- Explain the systems to three different audiences:
  - Testers: need to understand to evaluate the app
  - Risk acceptors: need to understand test reports
  - Developers: need to implement it properly
- Also, nice for users to get a peek behind the scenes

Definitions and Goals

Definitions and Such

- Authentication vs Identification vs Authorization
- Focus is on what the client application does
  - Sometimes it actually is Authentication
  - Sometimes it’s technically Authorization
- We’ll just call it all Authentication for now

Focus

- Looking at authentication “protocol” or system
- Strictly how it’s “supposed to work”
- Assuming client and server are legit
  - Not hacked
  - Not a phishing site
- Not a formal security analysis
  - Almost all of these have had some issues
- What can an attacker do?

Real World Authentication

- Your friend Tim walks up and says “Hi.”
- You look at him, say “Hi,” and begin talking.

- That was easy!!

How'd that work?

- Someone simply recognizes you
  - (or compares your face to a trusted ID)
- It's hard to spoof that appearance
  - Especially if you factor in other cues
- Anyone can watch, without weakening security

Not as easy in “Cyberspace”

- Server doesn’t have years of personal experience
- Or brains evolved for facial and behavioral recognition
- So it relies on some kind of “factor.”
- Usually a proof of knowledge that only the user has

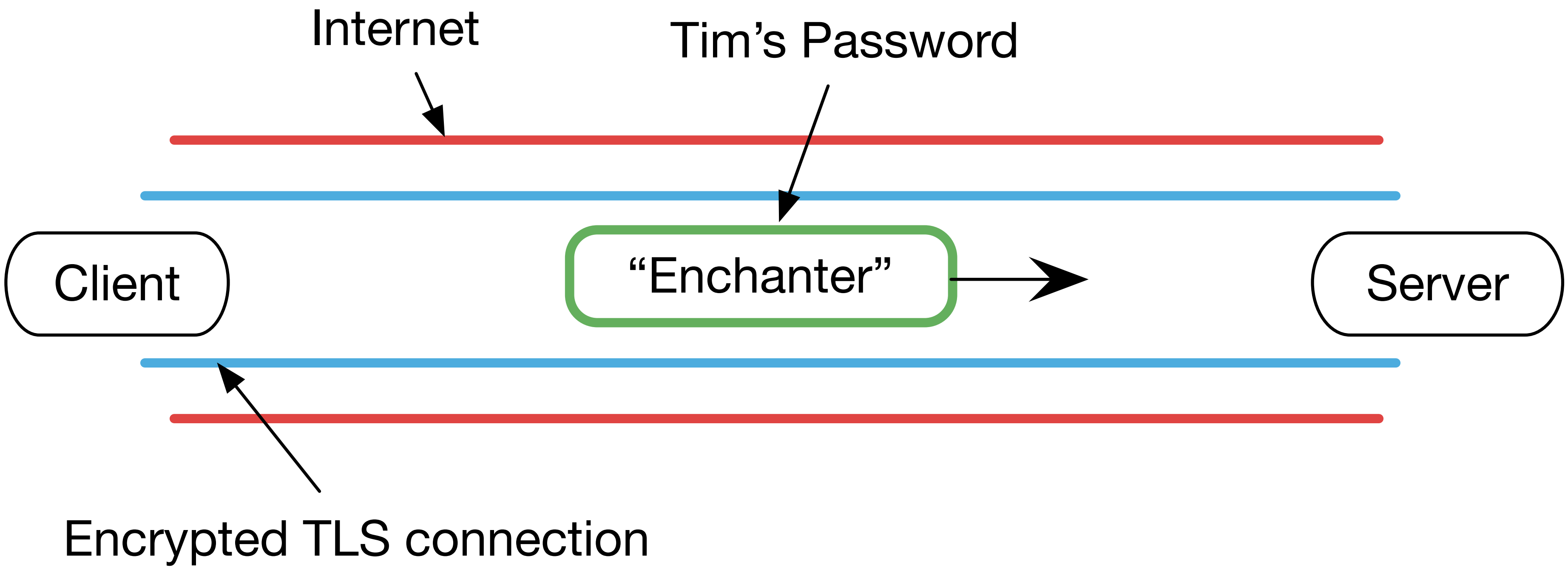
- Like a password

The Crux of the Problem

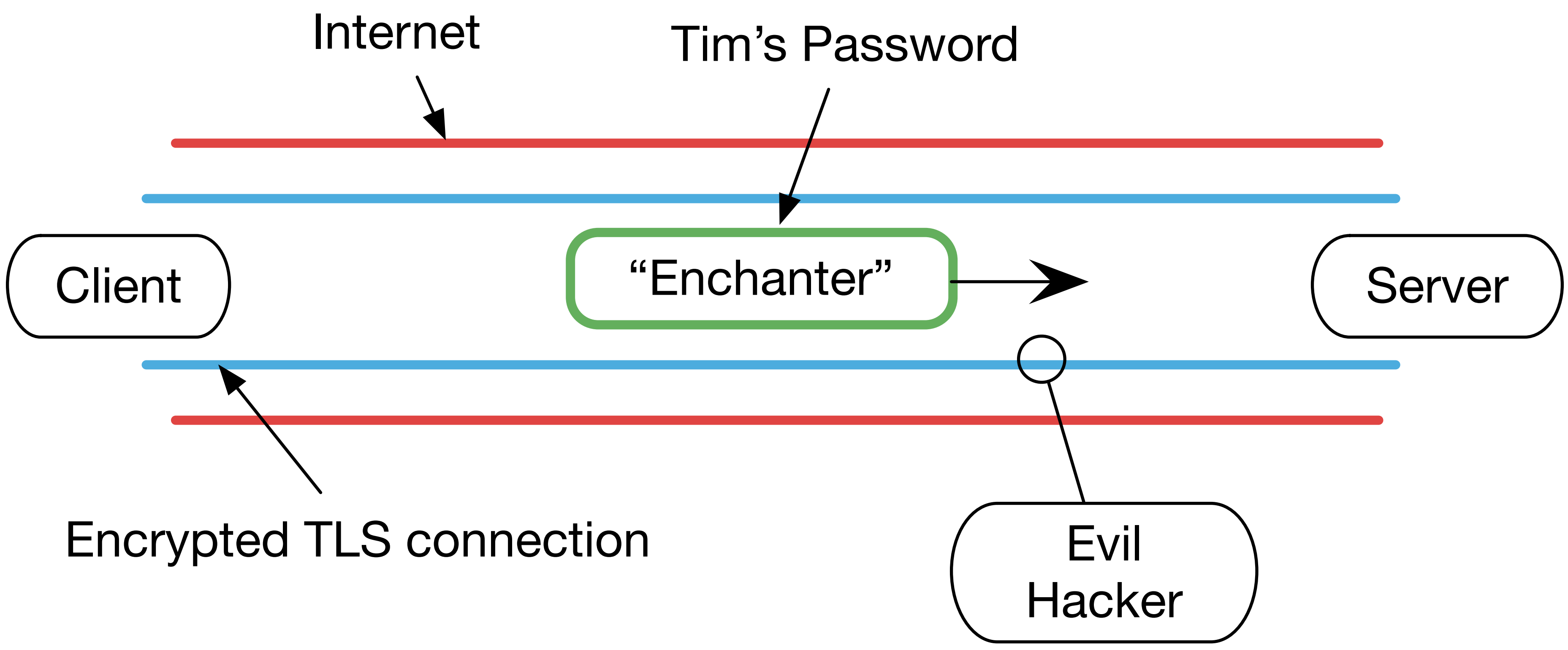
- How do you securely show that password to the server?

- In real life, you wouldn’t shout it out… You’d whisper.
- On the network, you hide it with encryption

It’s a series of tubes



(Hopefully, opaque tubes)



“We’re safe — we use TLS!”

- 2011: Certificate Authority DigiNotar hacked
  - Fake certificates issued
- 2011: iOS “Basic Constraints” bug
  - Real certs can sign fake certs
- 2012: Trustwave issues globally trusted wildcard cert
- 2014: Heartbleed bug - can expose private keys
- 2015: Superfish Adware
  - Self-signed certs on Lenovo laptops can MITM anything
- ? - present: Self-trusted Corporate DLP Proxies

Attack Surfaces

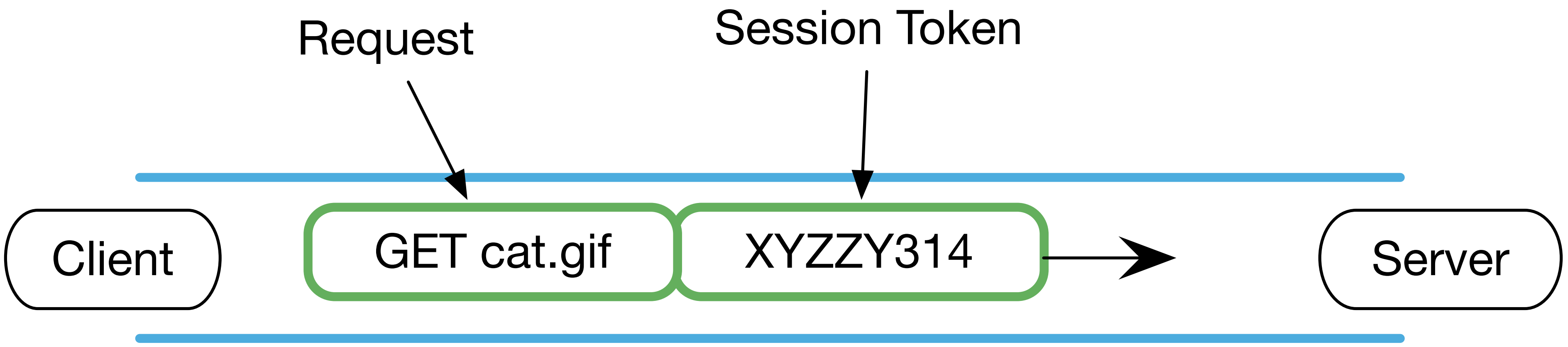
- Login
  - When the user first connects to a service
- Session
  - Continuous authentication
  - While using the application
- Restart (login again)
  - “Remember me” functionality
  - After application has been unused for a time

Attacks - Stealing Credentials

- Network: Disable TLS
- Transit: Server, server network
- Use: Server or client
  - Target application software
  - Intercept keystrokes, search memory
- Storage: Server or client
  - Server - steal and crack passwords
  - Client - steal locally stored passwords and tokens
  - Client - steal improperly cached credentials

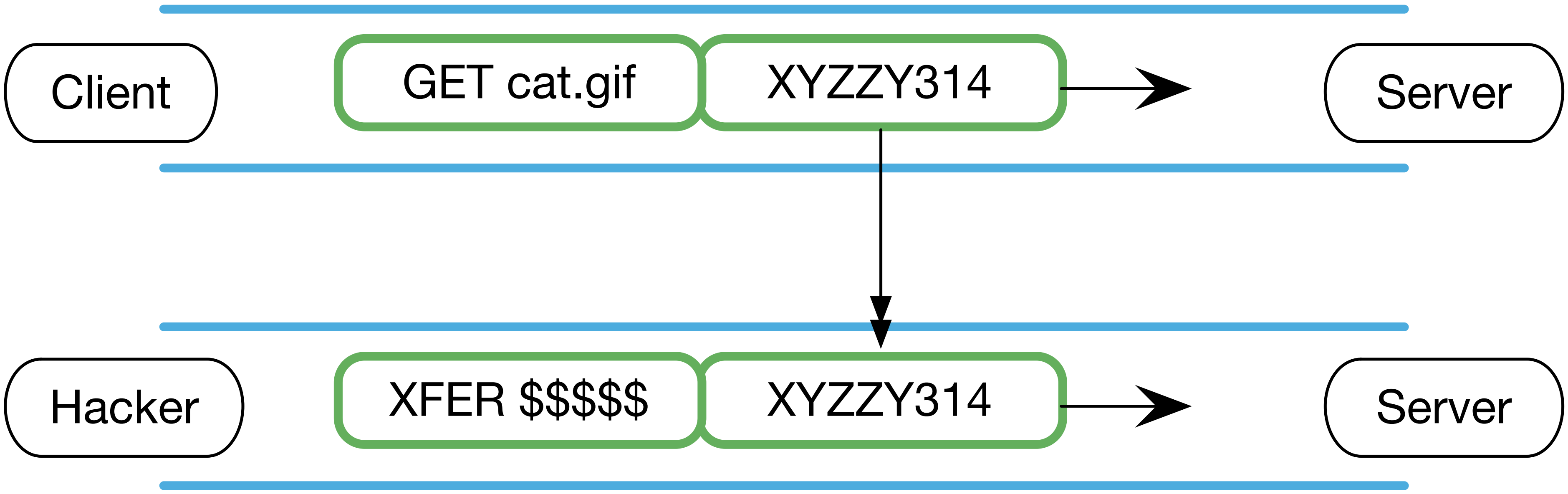
Session Tokens

- Could send your password with every request…
- …or could send a special code that only lasts a day



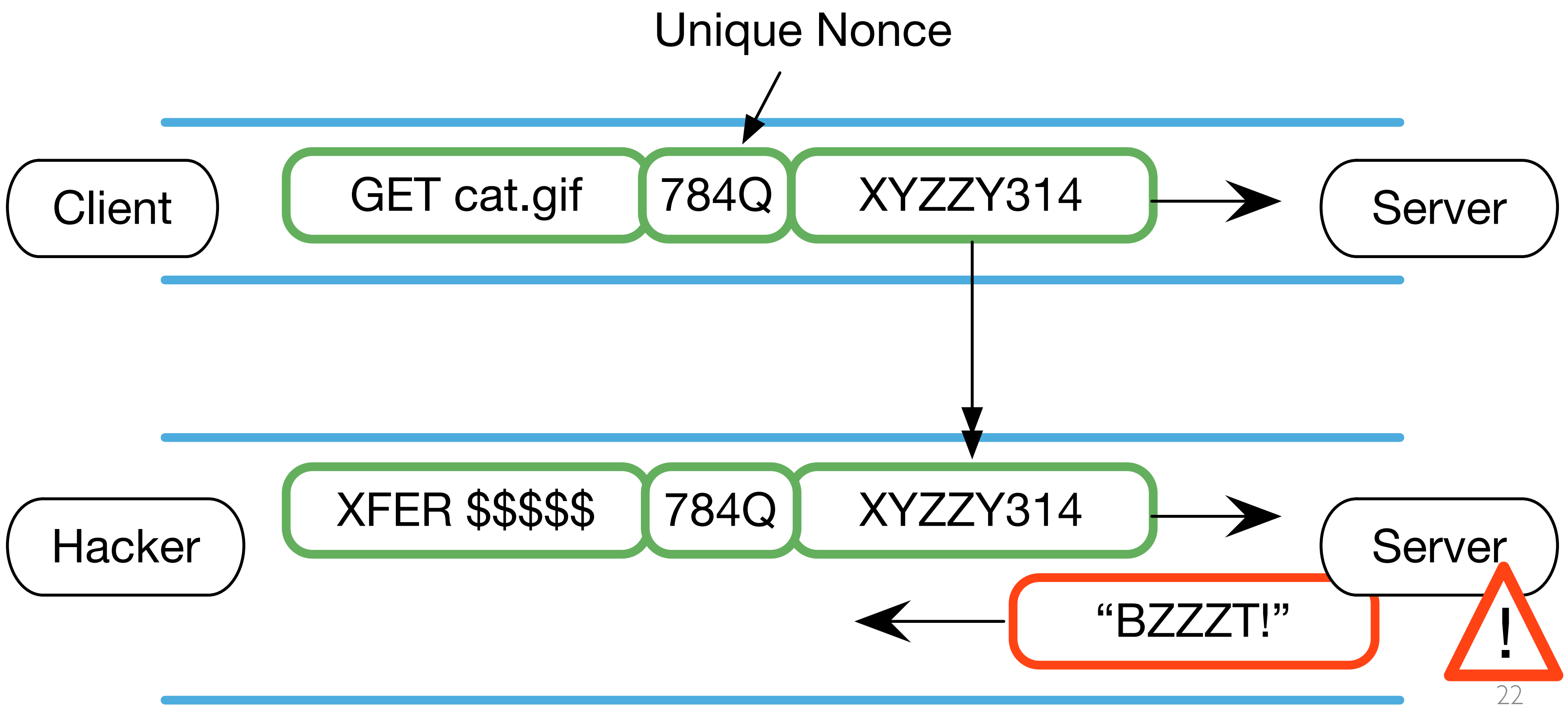
Token Compromise

- If an attacker can steal the session token
  - Break encryption, attack server, attack client
- They can make requests while pretending to be you



Make the Tokens Expire Faster

- Using a "Nonce" - random string - tied to the token
- If the nonce is repeated, server refuses request



Nonces and Timestamps and Sigs

- Server tracks all the used nonces
- Token has to be ***cryptographically tied*** to nonce
- Otherwise hacker could just change the nonce
- Include timestamps - server can discard old nonces

- Still doesn't cover all attacks, but enough for now

Commonly Used Systems

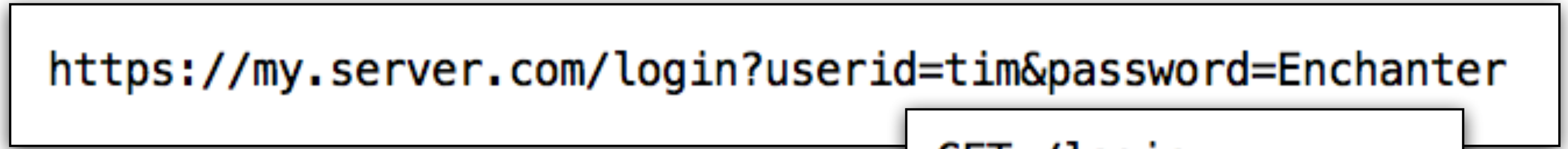
Commonly Used Systems

- Dozens of “mainstream” systems available today
- Only a few in common use
- Will describe five in detail:
  - Password or “Basic”
  - Digest
  - NTLM
  - OAuth 1
  - OAuth 2
- Basic, Digest, and NTLM generally built into browsers

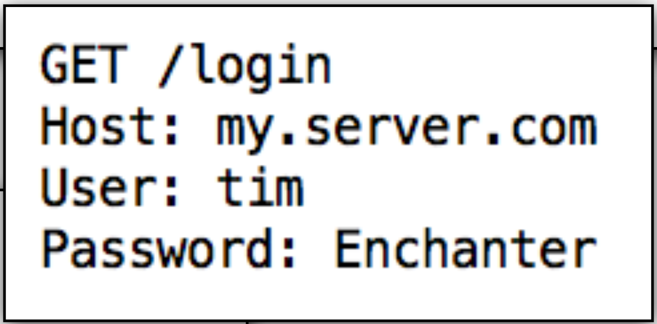
Authorization: Basic

Authorization: Basic

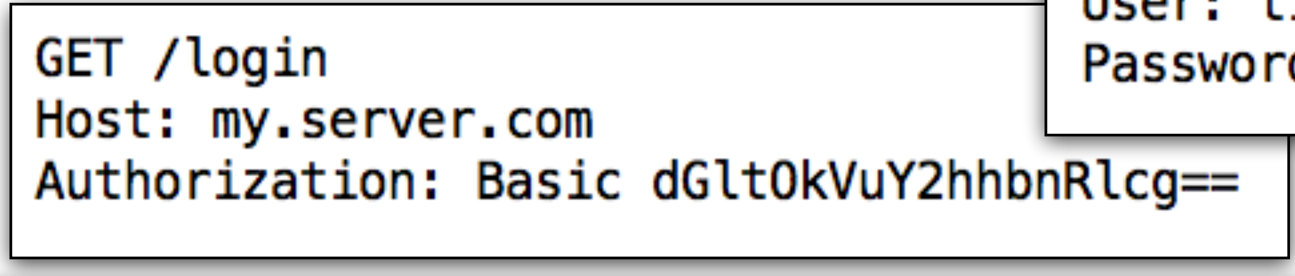
- Simply send your userid and password to the server



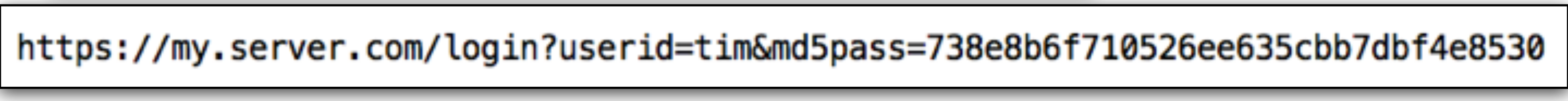
```
https://my.server.com/login?userid=tim&password=Enchanter
```



```
GET /login
Host: my.server.com
User: tim
Password: Enchanter
```



```
GET /login
Host: my.server.com
Authorization: Basic dGltOkVuY2hhbnRlcg==
```



```
https://my.server.com/login?userid=tim&md5pass=738e8b6f710526ee635cbb7dbf4e8530
```

Assessment: Basic

- Good:
  - Simplest method
  - Can use built-in browser form
    - Trigger with “WWW-Authenticate:” header
  - Or use HTML or Javascript form within a page
- Bad:
  - Perfectly fine… IF you trust TLS
  - Use for sessions increases disclosure risk
  - (also, don’t ever put it in the URL…even hashed)

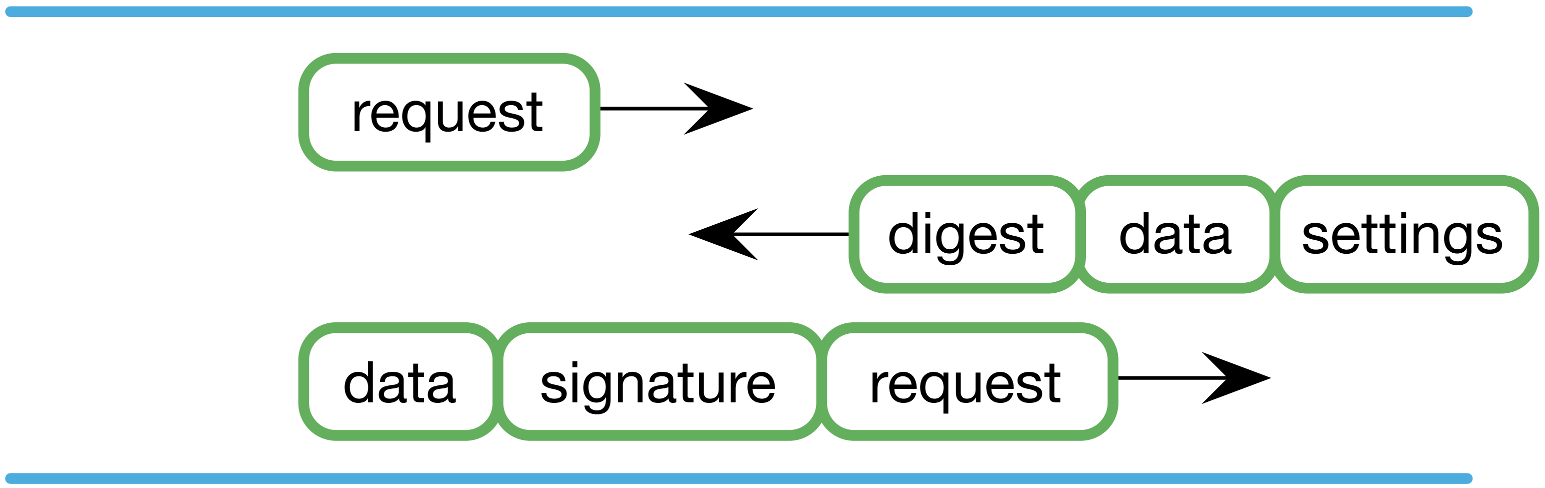
Authorization: Digest

Authorization: Digest

- More complicated
- Server sends unique parameters
- Client combines these with the user’s password
- Creates token that’s uniquely tied to the nonce
- The password is never sent over the network
- Optionally includes a signature of the request

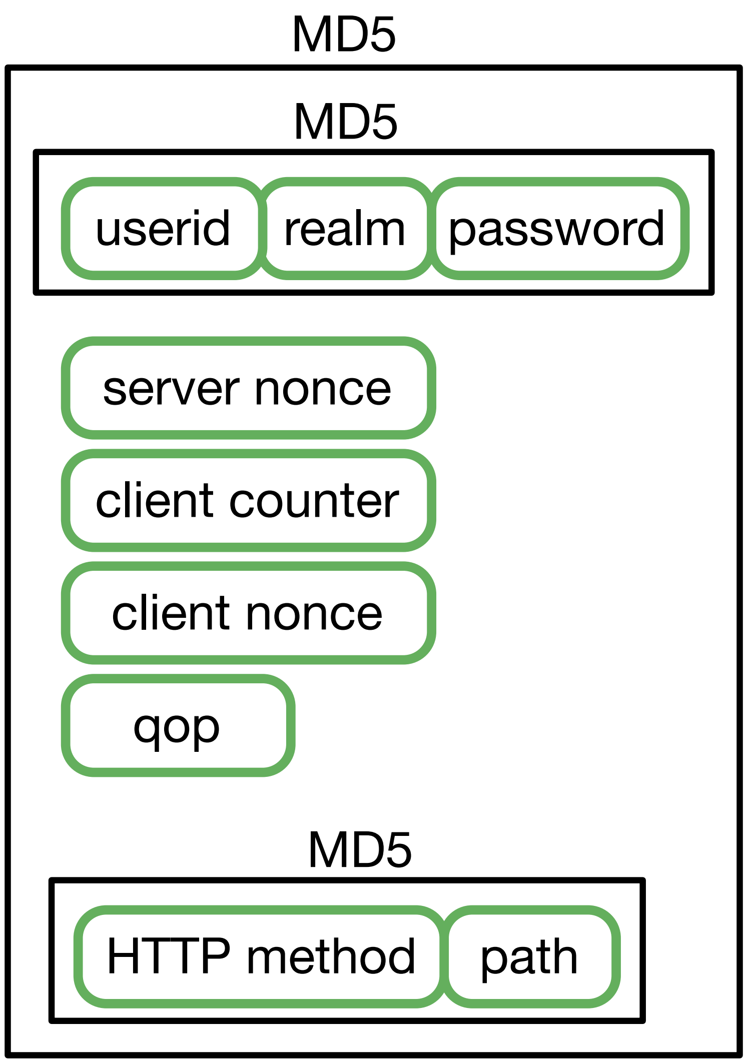
Authentication Flow

- Client requests a resource
- Server replies "Authenticate with Digest, please"
  - Provides realm, nonce, algorithm choice, QOP
- Client builds response, based on:
  - Server data, client data, user credentials



Computing Response

- Server nonce: random string
- Client nonce: random string
- Client counter: number
- QOP: Quality of Protection
  - "auth" or "auth-int"
- Server selects algorithm
  - MD5
- All are strings (hashes in hex)
- Joined with ":"



Server challenge

```
WWW-Authenticate: Digest
 nonce="1450807853.38:E10B:
  497c0eadca9e962b45e54cd2629399b3",
 realm="caerbannog",
 algorithm="MD5",
opaque="ADAC33E813C0CE930F4744C90E02396E",
qop="auth",
stale="false"
```

Hashes and Nonces

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

MD5('GET:/') =
   71998c64aea37ae77020c49c00f73fa8

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

Complete Response

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

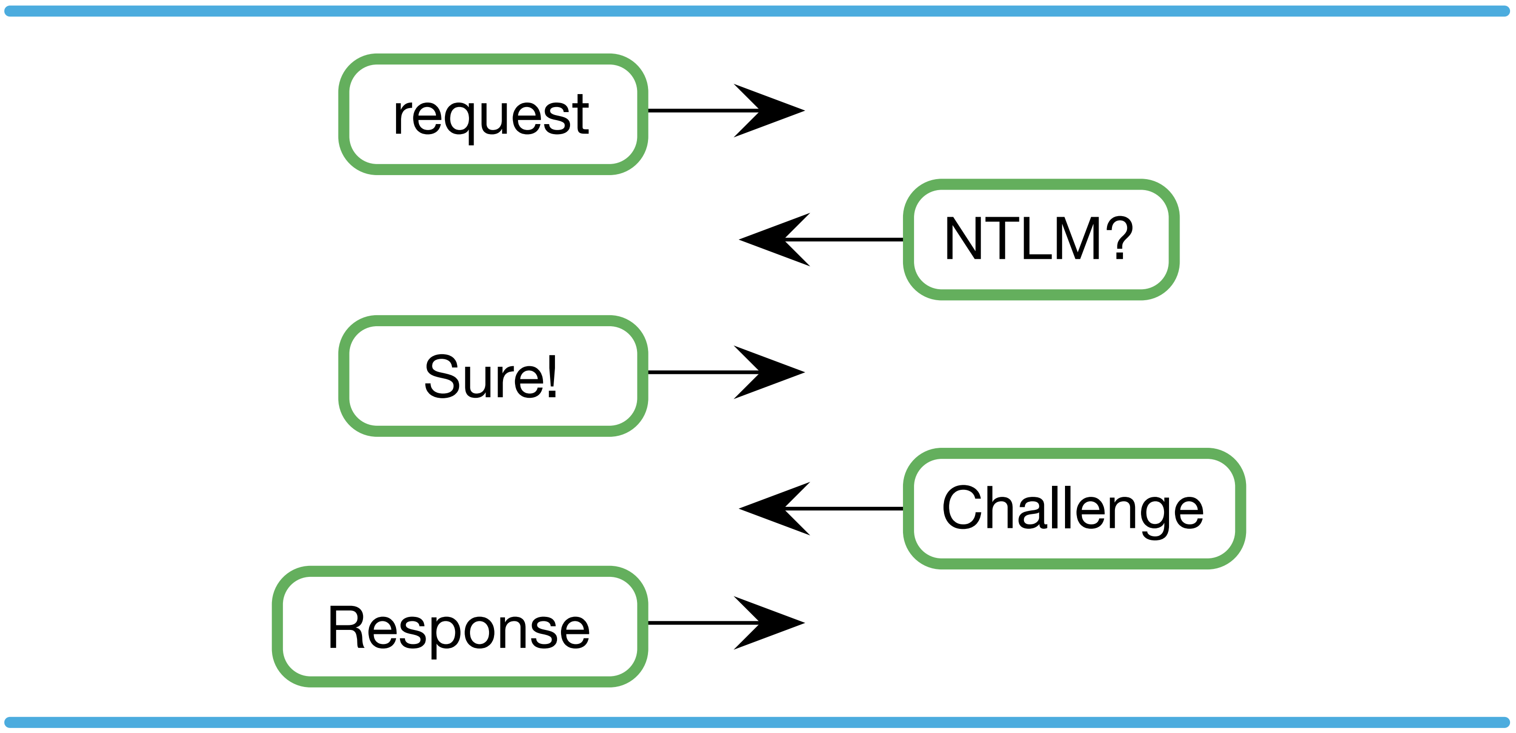
Assessment: Digest

- Not as frequently used for session credentials
- Good:
  - Client / sever nonce help prevent replay attacks
  - Password not sent over the network
  - QOP auth-int detects modification of request
- Bad:
  - Relies on MD5(username:realm:password)
  - Hash compromise allows immediate account access
  - Completed response may be vulnerable to brute-force
  - Uncertain support for auth-int

NTLM

NTLM

- Windows NT LAN Manager authentication
- Proprietary, not very well publicly documented
- Binary protocol, very complicated



Type 1 - Client Begins

- Server requests NTLM
- Client responds with "Type 1" message

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: NTLM


Authorization: NTLM
TlRMTVNTUAABAAAAB4IIAAAAAAAAAAAAAAAAAAAAAAA=


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

Type 2 - Server Challenge

- Server responds with challenge

```
WWW-Authenticate: NTLM [long base64 string]


4e54 4c4d 5353 5000 0200 0000 1000 1000  NTLMSSP.........
3000 0000 0102 8100 6e64 fc33 bb92 a567  0.......nd.3...g
0000 0000 0000 0000 8000           00    ............@...
6d00 7900 7400 6100 7200 6700 6500 7400  m.y.t.a.r.g.e.t.
0200 0000 0100 1400 6d00 7900 6300 6f00  ........m.y.c.o.
6d00 7000 7500 7400 6500 7200 0400 2800  m.p.u.t.e.r...(.
6d00 7900 6400 6f00 6d00 6100 6900 6e00  m.y.d.o.m.a.i.n.
2e00 6500 7800 6100 6d00 7000 6c00 6500  ..e.x.a.m.p.l.e.
2e00 6300 6f00 6d00 0300 2c00 6d00 7900  ..c.o.m...,.m.y.
6300 6f00 6d00 7000 7500 7400 6500 7200  c.o.m.p.u.t.e.r.
2e00 6500 7800 6100 6d00 7000 6c00 6500  ..e.x.a.m.p.l.e.
2e00 6300 6f00 6d00 0000          ) 0000  ..c.o.m.........
```

Server Challenge

Target Block

Response - NTLMv1

- Calculate NTLM hash of user's password
- Pad with 5 bytes of zeroes to make 21 bytes
- Convert to 3 DES keys, 64-bits each
  - Spread bits out in 7-bit blocks
  - Add parity bits in between and at end
  - Read back as 8-byte, 64 bit key (x3)
- Encrypt the challenge with each key in turn
- Append all the results into one long response

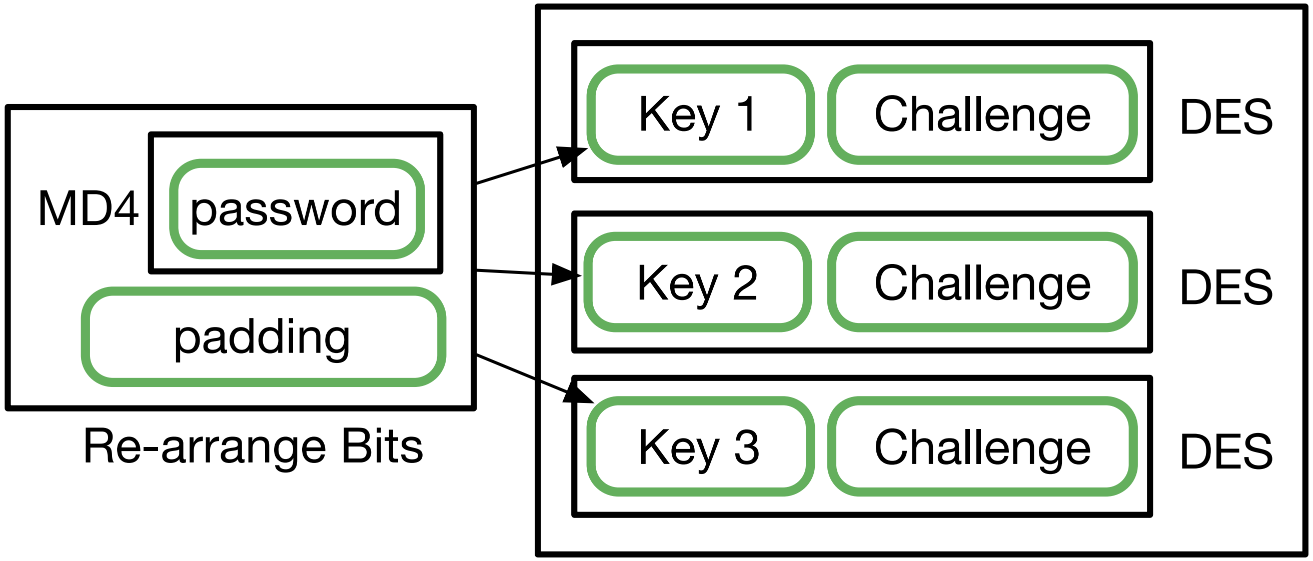
- Repeat with LM hash

Building DES Keys

```
MD4(UTF-16(“Enchanter”)) = 52dac53d34d998ee55b1137d647e93ce

Hash:    5    2    d    a    c    5
Binary:  0101 0010 1101 1010 1100 0101 ...
Spread:  0101 001x 0110 110x 1011 000x 101 ...
Parity:  xxxx xxx0 xxxx xxx1 xxxx xxx0 ....
Keys:    5    2    d    6    b    0    ....

Final keys:
  526db0a7d3a76731, ef2a6d2337ea91fd, 92e6800101010101
```



Encrypting Response

```
Server challenge:  6e64fc33bb92a567

key1 = 526db0a7d3a76731, c1 = 8f86b036e38ac5b8
key2 = ef2a6d2337ea91fd, c2 = f5913b338af9912e
key3 = 92e68001010101, c3 = e3037866d94f62d7

NTLM V1 response:
8f86b036e38ac5b8f5913b338af9912ee3037866d94f62d7
```

Type 3 Message

- Header, reserved bits, etc.
- Computer's Windows domain name
- User name
- Computer DNS name
- Responses
  - Response built from NTLM hash
  - Response built from LM hash

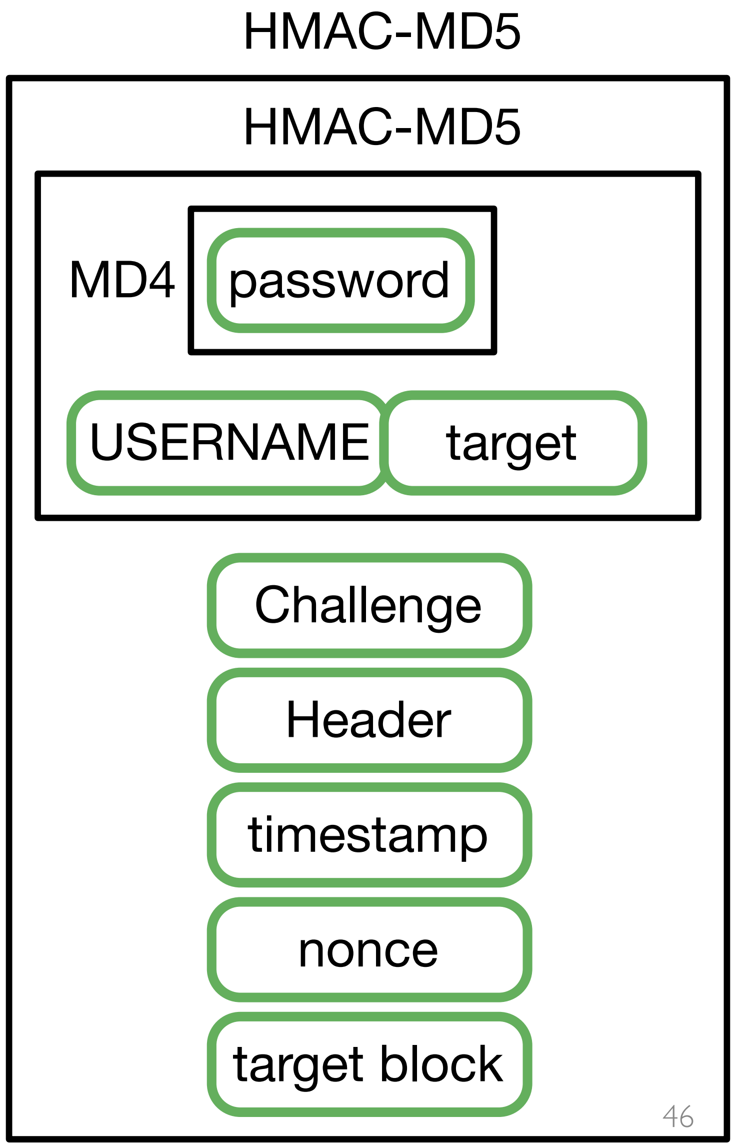
Final Type 3 Message

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

(highlighted: NTLM-based Response)

NTLMv2

- MD4(password) = NTLMv1 hash
- Concatenate:
  - Uppercase username
  - Lowercase target domain
- HMAC using v1 hash as key
  - This is the NTLMv2 hash
- Concatenate challenge, etc.
- HMAC using v2 hash as key



Computing NTLMv2 Hash

```
NTLMv1 hash: 52dac53d34d998ee55b1137d647e93ce

Username: tim
Domain target: (null)

Username + Target string: “TIM”

NTLMv2 Hash: ab774da35ccf4d3c9fc19cbb2829a6a8

Challenge: 115f4b6a06e258e0
Target block: (from Type 2 message)

NTLMv2 response:
84aa184c29f75144a225f0cdf732512b
```

Type 3 Response

```
Authorization: NTLM Tl[... long base64 string ...]AAAAAAA

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

Response

Timestamp

Nonce

Target Block

Assessment: NTLM

- Not as frequently used for session credentials

- Good:
  - Reasonably strong, at least in NTLMv2
  - Client / sever nonce help prevent replay attacks
  - Password not sent over the network
- Bad:
  - Complex, vulnerable to implementation bugs
  - Hash compromise allows immediate account access

OAuth Version 1

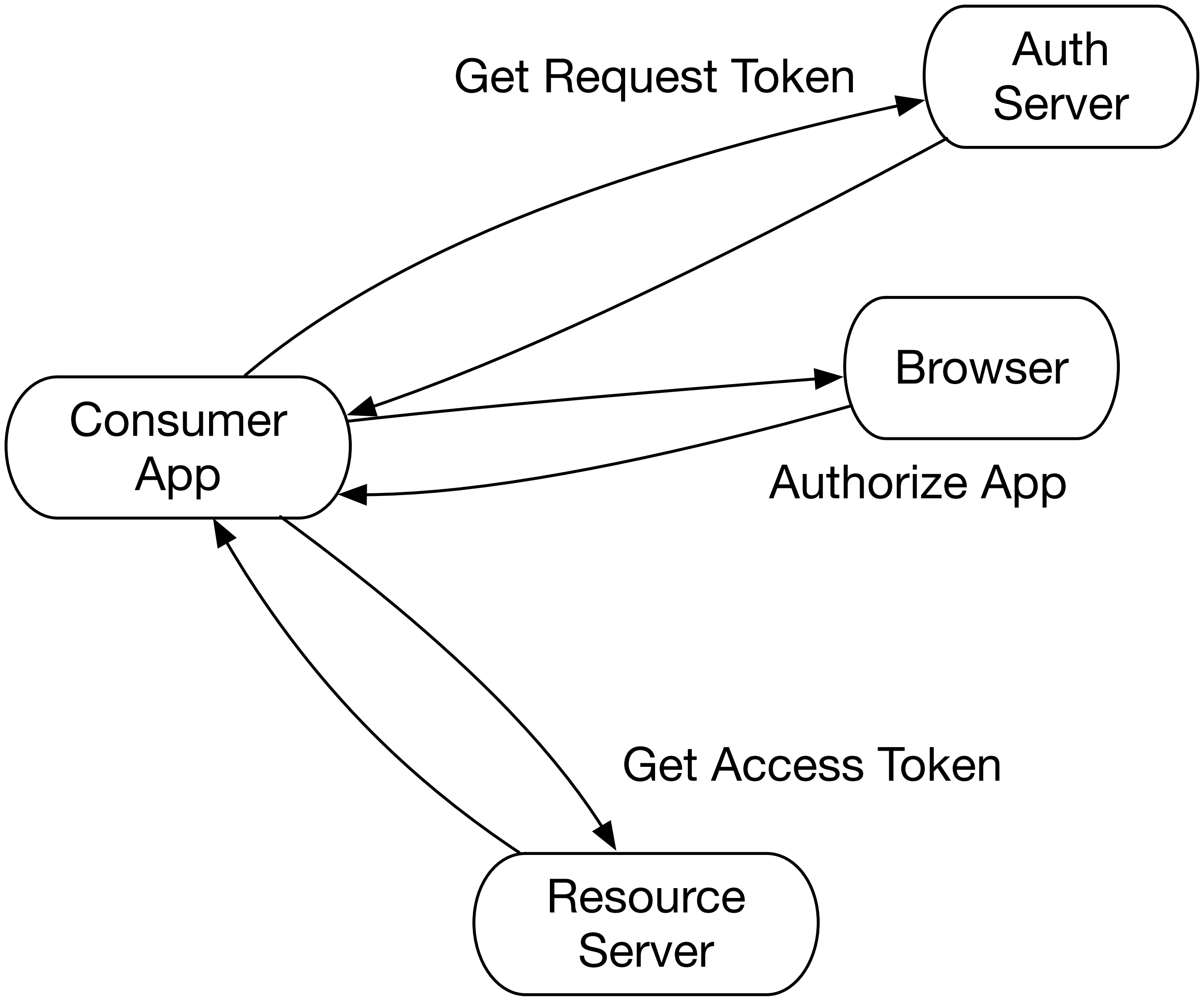
OAuth 1

- Began as “OpenID for Twitter” in 2007
- Not actually an authentication system
- Brokers a user’s *Authorization* from a 3rd party system
- In practice, distinction is sometimes kind of blurred
- Makes extensive use of shared secrets and signatures

Typical Authorization Flow

- Tim wants to let GrailTweet access his Twitter account
- GrailTweet connects to Twitter: “Give me a request token”
- User logs in to Twitter
  - Shows request token, authorizes GrailTweet
  - Receives activation code
  - Provides code to GrailTweet
- GrailTweet shows request token and code to Twitter
- Twitter gives GrailTweet an Access Token
- Now GrailTweet is connected to Tim’s Twitter account

Auth Flow Diagram

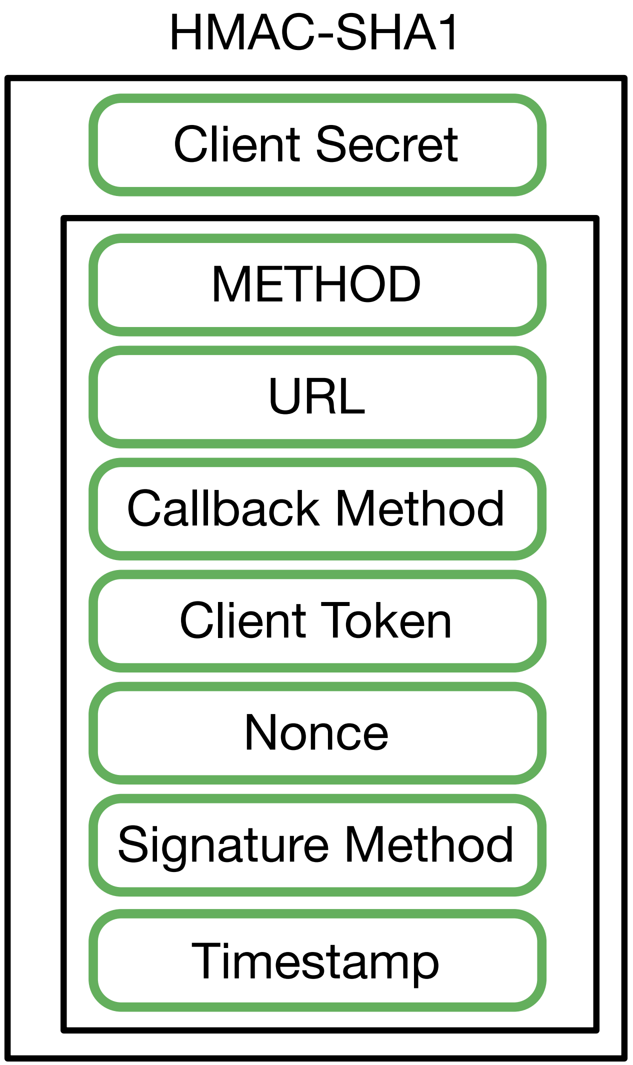


Norm Says "SETEC"

- Client has an application-specific token and secret key
  - Assigned to app (or developer) by provider (Twitter)
  - Long-lived, often compiled into application
- Authorization Request has specific token and secret key
  - Received during authorization setup
  - Discarded once client receives Access Token
- User has account-specific token and secret key
  - Provided when authorization complete
  - Long-lived, should be stored securely on device

Computing Signature

- Build request string
  - Sort parameters by name
  - Build as UTF-8, URL-encoded string
- Prepend:
  - Method (GET) in uppercase
  - URL (url-encoded)
- Use client secret and "&" as key
- Signature is HMAC-SHA1 of string
  - Optionally, RSA-SHA1



Example Authorization String

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

HMAC-SHA1("OEUyM0MtQUQzNi00Q0M4L&",
authorization_string):
   xTAwVGGUGjT1ViJH5EQtPKeRn50%3D
```

Submitting 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=
"xTAwVGGUGjT1ViJH5EQtPKeRn50%3D"
```

Permission to Ask: Granted

- Client now receives a temporary request token and key
- Client opens a browser using the token (no key):
  - https://api.myservice.com/oauth/authenticate?oauth_token=NDdDOTUtNzAxRCooMDIzL
  - User authenticates to service, authorizes app
  - Services displays an authorization code to the user
- Client creates a new OAuth request (as above), including:
  - oauth_token = (request token)
  - oauth_verifier = (authorization code)
- Uses request key as part of signing key:
  - “<client key> & <request key>”

Authorization is complete

- Client (finally) receives access token and key
  - Stores these securely
- Discards request key
- Future requests use the new access token
  - oauth_token parameter
  - access token key as second half of signing key

Future Requests are Signed

- Request parameters may be added to authorization string
  - UTF-8, url-encoded, sorted in with rest of "oauth_" vars
- Request signed:
  - Using client (consumer) key and user (access token) key

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

Server Security

- Compromise of (token, key) pairs grants full access
  - No password cracking necessary
- Suggestion:
  - Store access token in hashed form
  - Application sends token in request
  - Server applies hash to token
  - Uses hash to look up user, then verify using key
  - Account compromise now limited to device only

Assessment: OAuth 1

- Good:
  - Very strong, with nonces, timestamps, and signatures.
  - Client stores only access tokens
  - Once authorized, password never sent
  - Tokens may be individually revoked by the end user
  - Can provide integrity controls on individual requests.
- Bad:
  - Quite complicated, can be difficult to understand.
  - Depends on remote service for actual authentication

OAuth Version 2

OAuth 2

- Starts as OAuth Web Resources Authorization Protocol
- Published as OAuth 2 in 2012
- Simplified in many ways
- Made more complicated in others

- Not a strictly defined protocol
- More of a framework — different systems may build / expand

(almost) No More Secrets

- User secrets and HMAC-SHA1 signing from OAuth 1 - Gone!
- Relies on transport and storage security
  - TLS to protect exchange of data
- Also local (and server) storage to protect tokens
- Client key and secret remain
  - But sent in clear during part of process

Typical Authorization Flow

- Client requests authorization from Resource Owner
- Resource owner responds with a Grant
- Client sends Grant to Authorization Server
- Server responds with an Access Token
- Client uses Token for future requests

Flow Varies by Grant Type

- Authorization Code Grant — closest to OAuth 1
  - Different Authorizing and Resource servers
- Implicit - typically aimed at browser clients
  - Avoids separate grant, returns token immediately
- Resource Owner Password Credentials
  - The user’s password is directly exchanged for a token
- Client Credentials - similar to Resource Owner
  - Authenticates client itself - like a system-level acct
  - Client acts as the resource owner

Example - WebApp to GitHub

- A user is already logged into a web application
- They want to connect that account to GitHub
- App redirects them to GitHub to get authorization token
- Includes web-app URL for GitHub to send token to

```
GET https://github.com/login/oauth/authorize?
  client_id=oozZYMUdsEtt&
  scope=repo&
  redirect_uri=https://myapp.com/oauth/code&
```

User Authorizes the Application

- The user authenticates to GitHub
  - Using GitHub's system, 2FA, etc.
- Authorizes the application to connect to their account
- GitHub connects to the callback URI and provides code

```
https://myapp.com/oauth/code&code=KLSqKzkta1
```

Application Requests Connection

- The browser is already authenticated to the user's app
- The app sends code back to GitHub
  - This time, including the app's client_id and secret

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

client_id=oozZYMUdsEtt&
client_secret=TKgx1h7Kq0W&
code=KLSqKzkta1&
redirect_uri=https://myapp.com/oauth/newtoken
```

GitHub Sends Token to App

- GitHub validates the client_id and secret
- Generates an access token and adds to the user’s account
- Connects to the user’s web app, providing token

```
POST https://myapp.com/oauth/newtoken

{"access_token":"e85ecf8fc409714a3c28fbd205f856
7de0521e57", "scope":"repo,gist",
"token_type":"bearer"}
```

Accounts Are Now Linked

- Whenever web app wants to access user’s account…
- ….simply send request to GitHub
- And include the user’s unique access_token

Refresh Token

- Some systems have expiration times on tokens
- So they provide a long-lived “refresh token” as well
- Client uses access token for normal requests
- When access token expires
  - Use refresh token to get new one
- Need to ensure both tokens are securely stored on client

Assessment: OAuth 2

- Good:
  - Generalized framework, open to extension and changes
  - Can provide direct authentication
  - Essentially, a standard “universal session token” format
- Bad:
  - May limit Interoperability with other systems
  - Does not include timestamp, nonce, signatures
    - Tokens are universally accepted
    - Must be carefully protected

Some Other Systems

FIDO U2F

- Fast ID Online - Universal Second Factor
- Allows website to directly challenge USB or NFC key
  - Built into Chrome, others in progress
- Still relies on “traditional” password or other authentication

U2F Overview

- User enters userid and password on remote site
- Site validates password, and sends a challenge to key
  - Application ID - tied to remote app
  - Handle - tied to user, allows more than one ID on key
- Browser appends data URI origin - helps prevent phishing
- Key locates correct identity, using {application, handle} pair
- Key combines app ID, challenge, and a counter, into message
- Signs message with private key stored on device
- Browser forwards response to server
- Server verifies signature using the account's public key

JSON Web Token

- Not a system or protocol, but a format for signed data
- Consists of three parts:
    - Header — defines token type
    - Claims — the contents of the token
    - Signature — a cryptographic signature of the claims

JSON Web Token

- Structure for data signed with shared secret
- Header — defines token type
- Claims — the contents of the token
- Signature — a cryptographic signature of the claims

Header	`{"typ":"JWT", "alg":"HS256"}`	`eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9`
Claims	`{"user": "tim", "is_wizard": true}`	`eyJ1c2VyIjoidGltIiwiaXNfd2l6YXJkIjp0cnVlfQ==`
Signature	`HMAC-SHA256('secret', '<header>.<claims>')`	`IVliVUihAm1B7ZIX2xk8FaMMlavQNPBbz7y33vSItMg`
Token	`eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c2VyIjoidGltIiwiaXNfd2l6YXJkIjp0cnVlfQ==.IVliVUihAm1B7ZIX2xk8FaMMlavQNPBbz7y33vSItMg=`	

OpenID

- Distributed Authentication / Identity system
- Essentially, user has an account associated with some server
- Then can use that account to sign into multiple services
- Was vaguely popular for a while, but not as much so today
- Most recent version published in 2014
  - OpenID Connect
  - Layered atop OAuth 2 framework

CRAM-MD5

- Challenge-Response Authentication Method, with MD5
- Frequently seen with email systems (SMTP, POP, IMAP)
- Server sends challenge string
- Client responds with HMAC-MD5(password, challenge)
- Several issues:
  - Server may require a plaintext copy of password
  - A sniffed transaction can be brute-forced
  - Attacker may respond with known challenge
    - Find response in list of pre-computed passwords
- System is now basically deprecated

X.509 and Public / Private Keys

- Benefits:
  - Very strong (library bugs and CA attacks notwithstanding)
  - Provides additional encryption and authentication
- Drawbacks:
  - Need to retain DB of large public keys
  - Client needs to securely store and manage private key
  - Need to authenticate new devices before key exchange

- Rarely seen, except as additional device-binding feature

Capability Review

Common Elements

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	optional	yes
OAuth 2	optional					

Strengths

- **Basic**: Simplicity
- **Digest**: Resistance to replay, optional request integrity
- **NTLM**: Resistance to replay
- **OAuth 1**: Replay, request integrity, revocable tokens
- **OAuth 2**: Simplicity, revocable tokens

Weaknesses

- **Basic**: Password sent over wire.
- **Digest**: Potential brute force attacks, server-side storage questions, support for non-MD5 and auth-int, hash use
- **NTLM**: Complex, binary. Issues with hash, esp. NTLMv1. Not often used for sessions. Hash use.
- **OAuth 1**: Complicated. Can't do direct authentication.
- **OAuth 2**: No replay or integrity protection.

Security-Relevant Features

Attribute	Password	Digest	NTLM	OAuth 1	OAuth 2
Login sends password	yes	no	no	n/a	optional
Client needs to store password	yes	yes	yes	no	no
Password sent during session	yes	no	no	no	no
Unlimited session tokens	yes	no	no	no	yes
One-time session tokens	no	yes	yes	yes	no
Request integrity protections	no	optional	no	yes	no

My Recommendation?

- Logins: **None are ideal**
  - Passwords may expose credentials on wire
  - Digest requires weak server-side storage
  - NTLM and Digest allow use of un-cracked hash

- Sessions: **OAuth 1**
  - Very strong
  - Resistance to replays and request tampering
  - Credentials can be revoked without resetting password

Other Considerations

Other Considerations

- Server performance
  - May be impacted by too many logins w/too strong hash
  - But how big a percentage of load is that, really?
- Future sever flexibility
  - Migrating to different hashes, different systems
- Client technical capabilities
  - Security of device token storage
    - 66% of iOS apps I surveyed stored tokens insecurely
  - OAuth isn't embedded in browsers
  - Do NOT try to implement in JavaScript

Other Considerations

- Password Reset
  - Still the weakest link
  - Compromise one email account, get everything?
  - SMS not necessarily better
  - Attackers have chained one recovery to another…
- Onboarding
  - Very narrow window for attacks
  - But in some circumstances may be a viable target

Response Builder

Testing Authentication

- Need to understand authentication when testing apps
- Common systems are usually easy to recognize
- How to test custom (or customized) systems?
  - Need to understand system to assess risk
  - Replicating responses proves (some) understanding
- Much custom coding required each time

- Building a simple tool to help

Shortcuts

- Simplify testing of known and custom systems
- Will still require some coding
  - Not turn-key, GUI, fill-in-the-blanks and shoot
- Helps to demonstrate results using real data
- Extend and modify in response to need

- Not ready today (SORRY!)
- Can show you what I’m working towards

Common Systems

```
% build_response --oauth1
Enter the parameters:
  Callback? oob
  Consumer Key? QkNDREItQjNDRC00NkNGL
  Consumer Secret? OEUyM0MtQUQzNi00Q0M4L
  Nonce? MTZBNDAtMTNBNy00MEQwL
  Signature Method? HMAC-SHA1
  Timestamp? 1453030020
  URL? https://rabbit.com/login
  Version? 1.0

Signature: xTAwVGGUGjT1ViJH5EQtPKeRn50%3D
```

Custom Systems

```
prompt userid "Userid"
prompt password "Password"
prompt realm "Realm"
compute msg1 join(":", userid, password, realm)
compute ha1 md5(msg1)
prompt sn "Server Nonce"
prompt cn "Client Nonce"
prompt nc "Nonce Counter"
prompt qop "QOP"
prompt method "Method (GET)"
prompt path "Path"
compute msg2 join(":", method, path)
compute ha2 md5(msg2)
compute msg3 join(":",ha1,sn,nc,cn,qop,ha2)
compute result md5(msg2)
```

Possible Enhancements

- Full-out client app
  - Can connect to a remote server
  - Prompts user for input (userid, password)
  - Makes computation, displaying intermediate results
  - Sends result to server
  - Possible fuzzing of parameters
- Server app
  - Similar functionality, but pretending to be a server
  - see also httpbin.org

Conclusion

Conclusions

- Many different methods for authenticating applications
- Only about five in very common use
  - Basic, Digest, NTLM, OAuth 1, and OAuth 2
- All have strengths, all have weaknesses
- The strongest (OAuth 1) largely derided as “too hard”
  - The recommended alternative (OAuth 2) is very weak

Hope For The Future

- Improvements to browsers would be helpful
  - Improve initial login security
  - Integrate OAuth 1 or similar system
  - Direct JavaScript support and interfaces

- New developments are exciting
  - FIDO U2F
  - Emphasis on two-factor and two-step systems
  - Alternative authentication systems (Digits)

References

- Slides will be available online
- White paper
  - Additional detail and better explanations
  - Extensive references
- Test and demonstration tool
  - Available once it’s finished
- Available at NCC and on my blog
  - https://www.nccgroup.trust/us/our-research/
  - https://darthnull.org/publications

Questions

- David Schuetz
  - Senior Security Consultant at NCC Group
  - david.schuetz@nccgroup.trust
  - @DarthNull

Locations

North America

Atlanta
Austin
Chicago
New York
San Francisco
Seattle
Sunnyvale

Europe

Manchester - Head Office

Amsterdam
Basingstoke
Cambridge
Copenhagen
Cheltenham
Edinburgh
Glasgow
Leathered
Leeds
London
Luxembourg
Malmo
Milton Keynes
Munich
Vilnius
Wetherby
Zurich

Australia

Sydney