# Knock Knock?

## A Survey of iOS Authentication Methods

David Schuetz
Senior Consultant, Intrepidus Group

@DarthNull

ShmooCon XI, January 17, 2015

# Welcome!

- David Schuetz
  - Senior Security Consultant at Intrepidus Group
  - iOS app security, web app testing, etc.
  - Crypto puzzle geek (frequent winner and author)
  - Spent years as a government contractor
    - Sometimes a little more risk-averse that most

# Agenda

- Introduction
- Selection of Apps
- Workflow
- Findings
- Suggestions
- Conclusion

# Introduction

# Applications!

- Millions of apps available in iOS App Store
- Many (most?) of them are completely standalone
  - Photo editors, games, calculators
- Many access sensitive information over the net
  - Banks
  - Health care
  - Work related
- Most of these will have web-based interfaces as well
  - Often with features beyond mobile apps
  - Credentials stolen from mobile may be useful there

# Backdoor via weak authentication

- Lost phone, left unlocked:
  - May be able to directly extract credentials
  - Configure a proxy and intercept credentials
  - If jailbroken....jackpot!
- Even if not lost or stolen:
  - May be able to MITM using public Wi-Fi

- Question: How exposed are most apps?

- Related: How are apps authenticating today, generally?

# Authentication Survey

- General assessment of a single security vector

- Select a cross-section of iOS applications
- How do these applications authenticate?
- Do they do anything insecurely?
- Could they do anything better?
- Anything exceptional (good or bad)?

# Selecting Applications

# My iPhone

- On my fourth iPhone in 6 years
- Accumulated many, many applications
  - Over 230 actually installed on phone today
  - (not even thinking about those I've deleted)
- Should be a pretty representative sampling of apps

- However: this means it's sort-of self-selected
  - So not really "pure" from a research standpoint

# Dropping from the list apps which:

- Have no obvious network services (100+ apps)
- Use only OS-managed services (like iCloud calendar)
- Use only 3rd party services (game tweeting scores)
- Only local network services (like SSH or VNC)
- Anonymous services (no userid)
- Those I don’t actually have logins for
- Anything no longer in the app store

# Applications remaining

- Nearly 50 applications
- Of these, seven couldn't be intercepted
  - Ignored proxy settings
  - Limited time meant I couldn't dig deeper
- Actually looked at 40 apps, including:
  - Banking
  - Healthcare
  - Travel
  - Cloud storage
  - Social networking

# Which applications?

- Not gonna tell ya. (Ha, ha!)
- Won’t endorse (or condemn) any individual app
  - Very narrow assessment focus
  - Didn’t look for bugs or weaknesses
- High level survey
  - Only spent an hour or less per app
  - Deep dives would take much longer

# Workflow

# Made four passes

- Using MITM proxy with untrusted root certificate
    - See whether apps even notice
    - Most actually did
- Using MITM proxy with trusted cert
    - See whether a forged cert works
    - Most of the time, it did
    - Bulk of work happened here
- Re-launched all apps days later
    - Final verification and loose ends
- Also re-launched with missing CA cert

# Looking for...

- Initial authentication
  - How are credentials passed?
- Continued (session) authentication
  - Credentials passed each time, or are tokens used?
- Renewed authentication
  - Can app automatically login after being quit?
- Credential storage
  - Is password stored, or just a token?
  - Is anything stored insecurely?

# High-level review targets

- Put another way, we want to verify:
  - Secure Network
  - Secure Login
  - Secure Session
  - Secure Storage

# Not looking for

- Specific security flaws
  - Though I found a few weaknesses
  - Including PII enumeration using email addresses
- Server-side issues
- Security of the application itself
- Didn’t reverse-engineer, disassemble, etc.
  - Not nearly enough time
  - Only two apps dropped because of complexity
- Didn’t look at additional “followup” authentication
  - Like asking your password before a purchase

# Ideas for future talks

- Third-party (4th party?) services
  - Analytics providers: Crashlytics, Flurry, Hockey, etc.
  - Many supported one or more of these
  - What data is being sent? How detailed? Is it secure?

- General application security
  - Unsafe data storage
  - Use (or lack) of data protection / file encryption
  - Sensitive data leakage in cache, logs, crash data, etc.

# General process

- Jailbroken iPhone 5C running iOS 8.1.2
- Installed all apps from app store
  - Only one app already on phone, so all rest are “fresh”
- Monitor /var/log/syslog
- Intercept traffic using Burp Suite
  - Launch application, monitor login process
  - Wander through app, monitor continuing session auth
  - Kill app, launch again, observe re-login process
- Review keychain for new entries
- Copy sandbox off device and review data storage

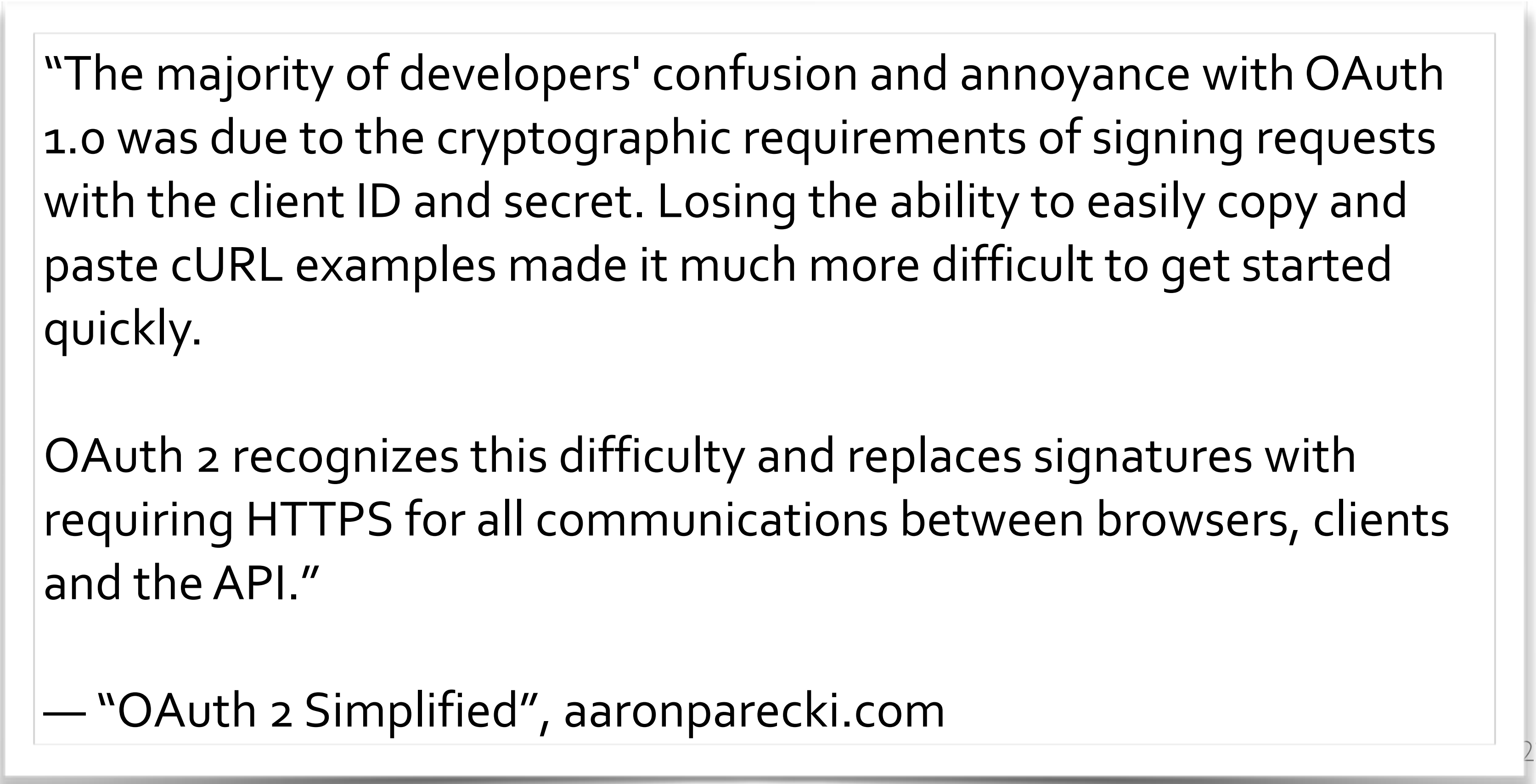
# Findings

# Review Target: Secure Network

- Credential-stealing attacks:
  - Public wifi, attacker MITMs connection
  - Unlocked device, attacker installs proxy + cert
- Good design:
  - Use TLS (No, seriously, use it)
  - If cert is untrusted, refuse connection
    - Blocks naive MITM
  - If possible, refuse if cert is unknown: PIN CERT
    - Blocks MITM with trusted but malicious cert

# Why is TLS so important?

- Most authentication methods absolutely rely upon it
- TLS compromise == You Win (something)



"The majority of developers' confusion and annoyance with OAuth 1.0 was due to the cryptographic requirements of signing requests with the client ID and secret. Losing the ability to easily copy and paste cURL examples made it much more difficult to get started quickly.

OAuth 2 recognizes this difficulty and replaces signatures with requiring HTTPS for all communications between browsers, clients and the API."

— "OAuth 2 Simplified", aaronparecki.com

# This bears repeating

- Dynamic tokens “confuse developers”
  - So we’ll rely on TLS
- TLS security far from guaranteed
  - CA compromise
  - OpenSSL bugs
  - Forced proxy use (work, school, airplane)
- I think the opposite stance is required:
  - Use TLS, but don’t rely on it
  - ***Assume your communications can be intercepted***

# First pass: No CA certificate

- Of forty applications tested:
  - 38 failed with some kind of error
  - One didn't care
  - One didn't notice because it didn't use TLS
- Terrible user experience

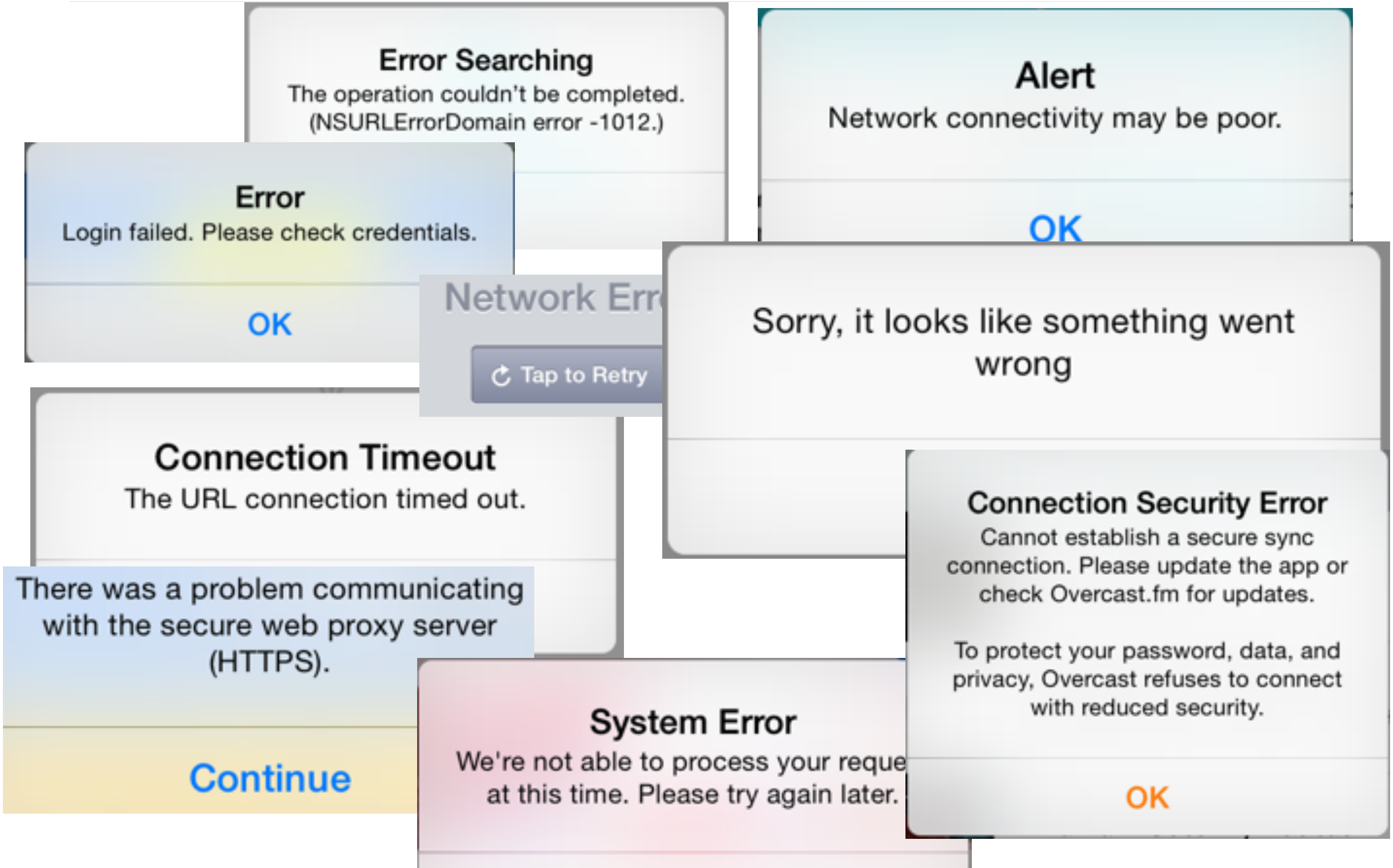
| Error Message Displayed | |
|---|---|
| General "network error" message | 14 |
| General "proxy error" message | 7 |
| Generic error (like "try again later") | 6 |
| "Check Credentials" | 6 |
| NSURLErrorDomain -1012 | 4 |
| Helpful and accurate message | 1 |

# Great error messages

- “Network connectivity may be poor”
- “Oops! Thanks for your patience while we get it fixed. Looks like something was left unplugged.”
- “Oops! We’re currently experiencing a number of technical difficulties within our app. We are actively working to resolve these issues and return the app to full functionality as soon as possible. Thank you for your patience.”

# One of these is not like the others

intrepidusgroup
part of nccgroup



**Error Searching**
The operation couldn't be completed.
(NSURLErrorDomain error -1012.)

**Alert**
Network connectivity may be poor.

OK

**Error**
Login failed. Please check credentials.

OK

Network Err

Sorry, it looks like something went wrong

**Connection Timeout**
The URL connection timed out.

There was a problem communicating with the secure web proxy server (HTTPS).

Continue

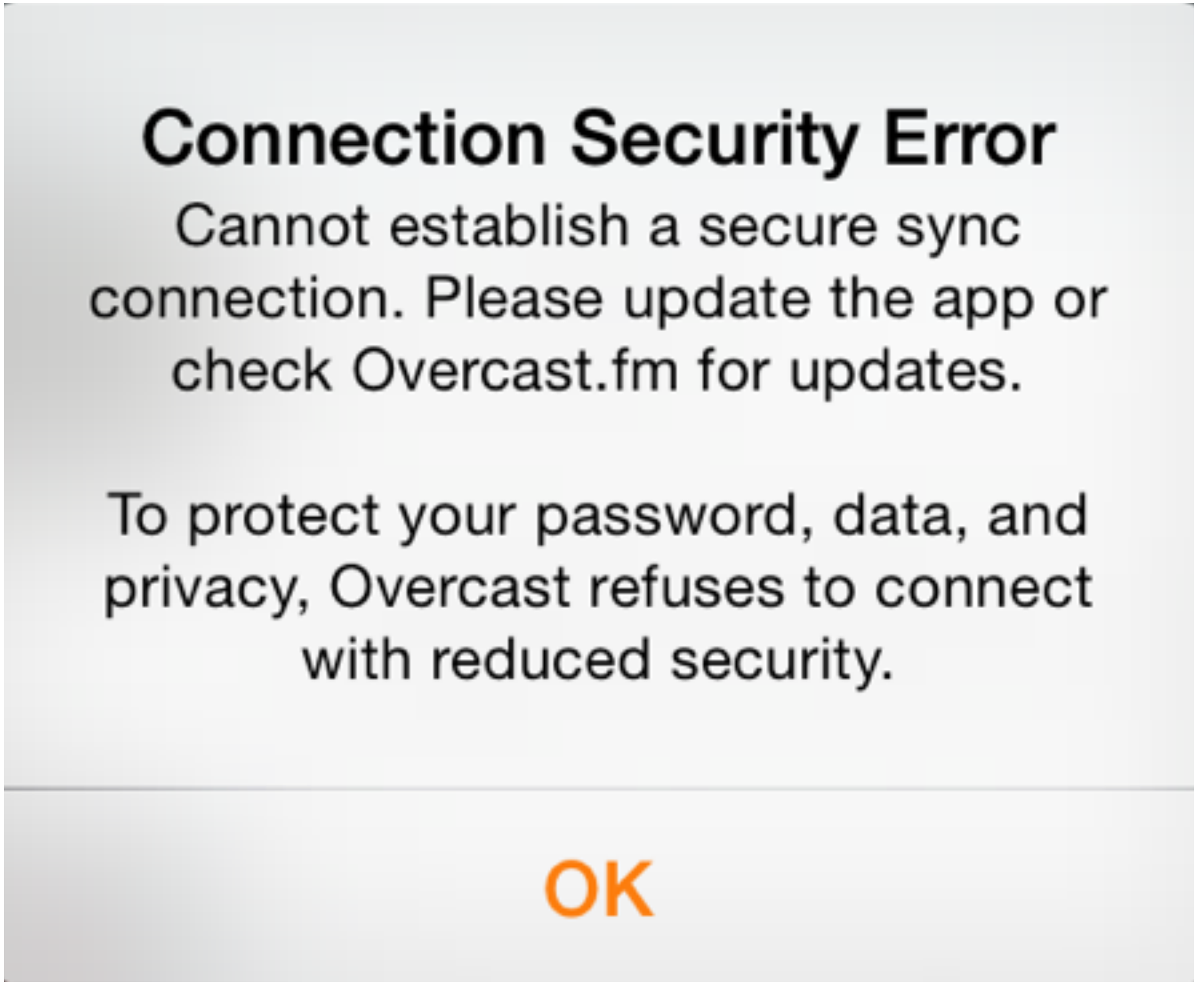
**Connection Security Error**
Cannot establish a secure sync connection. Please update the app or check Overcast.fm for updates.

To protect your password, data, and privacy, Overcast refuses to connect with reduced security.

OK

**System Error**
We're not able to process your reque
at this time. Please try again later.

# Best error message

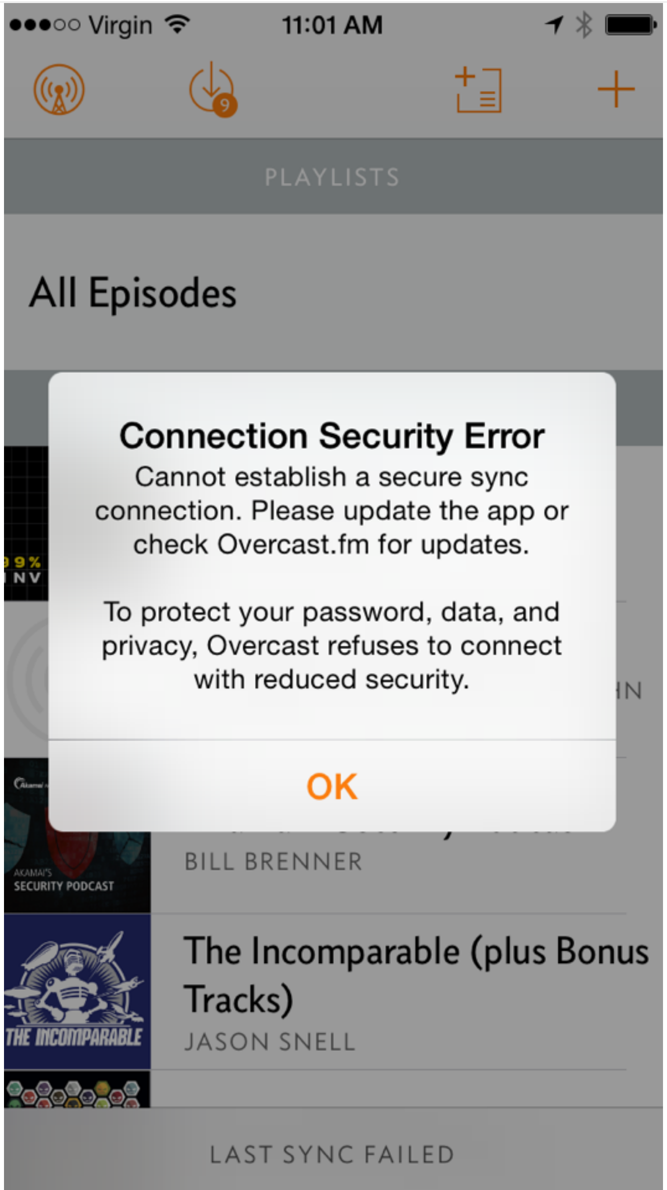


**Connection Security Error**

Cannot establish a secure sync connection. Please update the app or check Overcast.fm for updates.

To protect your password, data, and privacy, Overcast refuses to connect with reduced security.

# Best LOL: It’s a podcast app!



**Connection Security Error**

Cannot establish a secure sync connection. Please update the app or check Overcast.fm for updates.

To protect your password, data, and privacy, Overcast refuses to connect with reduced security.

OK

# Second pass: Installed Burp cert

- Of 38 apps remaining:
  - 34 applications worked fine
  - Two applications appeared to be pinned
    - But I bypassed using Snoop-It
  - One appeared pinned (and couldn't be bypassed)
  - One appeared pinned and may even be cert based:
    - "Missing authentication credentials [cert]"

- Able to continue analysis with 36 apps
  - (plus two from before, so 38 total)

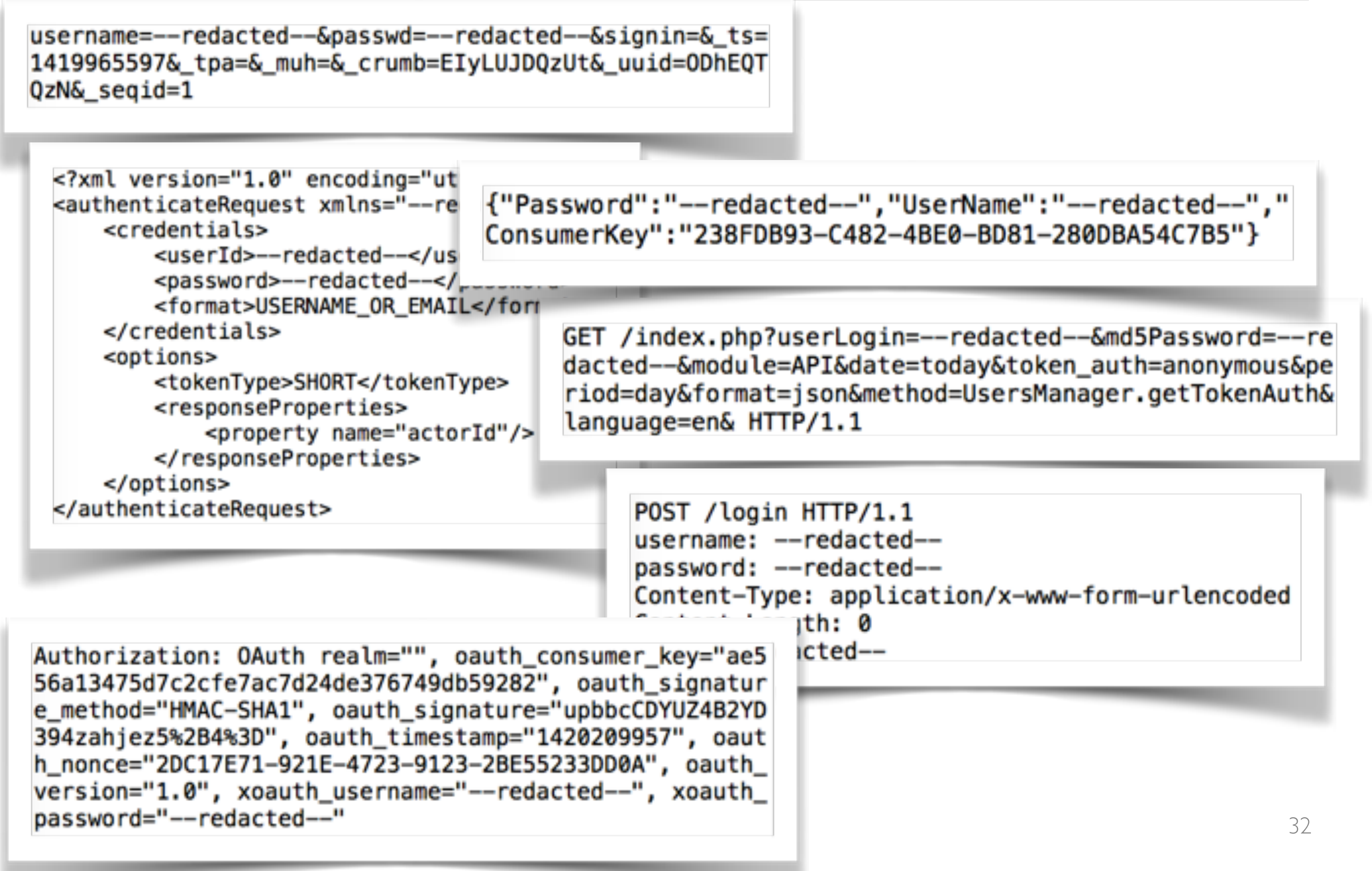
# Review Target: Secure Login

- Attacker:
  - Circumvents one or more controls
  - Steals password from network
  - Reuses password elsewhere (web interface)
- Good design:
  - Never store or send password
  - Store some kind of hash or refresh token

# Initial authentication

- Most passed “plaintext” userid and password
  - Vast majority via data in the POST body
  - A handful via HTTP headers
    - Two were not even base-64 encoded
  - Two sent credentials as part of the URL
    - One had userid, MD5(password)
    - One had human-readable plaintext
- Some passed secondary credentials over POST
  - Answer to security question
  - 2-step verification PIN

# Many data formats



```
username=--redacted--&passwd=--redacted--&signin=&_ts=
1419965597&_tpa=&_muh=&_crumb=EIyLUJDQzUt&_uuid=ODhEQT
QzN&_seqid=1
```

```
<?xml version="1.0" encoding="ut
<authenticateRequest xmlns="--re
    <credentials>
        <userId>--redacted--</us
        <password>--redacted--</
        <format>USERNAME_OR_EMAIL</form
    </credentials>
    <options>
        <tokenType>SHORT</tokenType>
        <responseProperties>
            <property name="actorId"/>
        </responseProperties>
    </options>
</authenticateRequest>
```

```
{"Password":"--redacted--","UserName":"--redacted--","
ConsumerKey":"238FDB93-C482-4BE0-BD81-280DBA54C7B5"}
```

```
GET /index.php?userLogin=--redacted--&md5Password=--re
dacted--&module=API&date=today&token_auth=anonymous&pe
riod=day&format=json&method=UsersManager.getTokenAuth&
language=en& HTTP/1.1
```

```
POST /login HTTP/1.1
username: --redacted--
password: --redacted--
Content-Type: application/x-www-form-urlencoded
                  gth: 0
                 acted--
```

```
Authorization: OAuth realm="", oauth_consumer_key="ae5
56a13475d7c2cfe7ac7d24de376749db59282", oauth_signatur
e_method="HMAC-SHA1", oauth_signature="upbbcCDYUZ4B2YD
394zahjez5%2B4%3D", oauth_timestamp="1420209957", oaut
h_nonce="2DC17E71-921E-4723-9123-2BE55233DD0A", oauth_
version="1.0", xoauth_username="--redacted--", xoauth_
password="--redacted--"
```

# Obfuscated credentials (beyond B64)

- Encryption
- Binary data structure
- GZip compression
- Only about five apps total

# Cool methods

- Encrypted password
  - App sends userid, server responds with key
  - App encrypts password with this key and submits
  - An attacker with just the one packet is out of luck
    - Of course, if they saw the key…
- Full encryption
  - Two appeared encrypted (not positive)
  - Another sent a private key upon login
    - All subsequent traffic included signature

# Dumb behavior

- Autocorrect in the username field
- Type in your account correctly:
  - It’s close to an English word
  - You tap to the password field
  - The phone “helps” you by correcting your userid
- Why?!?
- One app took me like 5 tries to notice

# Review Target: Secure Sessions

- Attacker (avoiding previous controls)
  - Can steal tokens and use elsewhere
- Good design:
  - Revocable token — user can de-authorize device
  - Use a dynamic token
    - Prevents replay attacks
    - However, attacker could return error, then use token
  - Use signatures
    - Request can't be altered
    - Also prevents transfer of unused tokens (as above)

# Continued authentication

- How are requests authenticated as the app is used?
- Two sent password with every request
- Most sent some kind of token
  - Many some kind of OAuth variant
- Tokens passed in a variety of locations:
  - URL parameters
  - Cookies
  - Authentication: or custom HTTP headers
  - POST request body

# Interesting tokens

- Most tokens decoded to meaningless binary
- A couple decoded to ASCII text
- One:
  - Base64(<client id>:<uid>:<?>:1:<timestamp>:0--<?>)
  - The last block was 16 bytes, possibly an MD5 hash?
  - Might be able to brute force a server secret. Maybe.

# Renewed authentication

- Force-quit each app then re-launched it
  - A few sent the userid and password again
  - A few asked the user to re-enter some credentials
  - Most sent a stored token
- Didn’t test token expiration times
- Almost all stored some credentials
  - What were they?
  - And how were they stored?

# Cool method

- Stores userid, encrypted with a server-side key
  - Client never sees the key
  - Client keeps partially readable name (“MyAcc***”)
- When logging in, app sends encrypted userid
- Then asks for password

# Review Target: Secure Storage

- Attacks
  - Unlocked device
  - Attach via USB
  - Directly access files in application sandboxes
  - Extract credentials that way
- Use the keychain — it's designed for this
  - Absolutely password and tokens
  - If possible, store the userid here as well

# Keychain storage

- Most secure location for storing sensitive data
- Hard to extract:
  - Need a jailbroken device
  - Or the iTunes backup password
- Found 8 usernames, 17 tokens, and 4 passwords
- Also found lots of other information:
  - Some preference settings
  - Push tokens
  - Credentials for 3rd party services (twitter, etc.)

# Less safe storage

- Preferences file (just a property list, not encrypted)
  - Almost half stored the username here
  - One included the user's password
- HTTP cache (URLs, some other header & POST data)
  - Eleven had a password or token
- Cookies file (Cookies)
  - Only a handful of userids and tokens
- Other files and caches in /Library and /Documents
  - Including tokens and even one password

# Third and fourth passes

- Third pass — renewed authentication (days later)
  - Launched all 38 apps again
  - Intercepted and reviewed login
  - No significant changes from first testing
  - Mostly verifying the previous findings
- Fourth pass — MITM again, with no CA
  - All behaved as before: TLS errors
    - Some seemed to work, using local data
    - But failed when accessing the network
  - Need to check the certificate at all times

# Summary of Findings

# TLS Certificates

| Handling of TLS Certificates | |
|---|---|
| Refused to work with bad MITM cert | 34 |
| Didn’t care about bad MITM cert | 1 |
| Didn’t notice bad cert because it used HTTP | 1 |
| Refused to use trusted MITM cert (pinning) | 4 |

# Initial Authentication

| Initial login credentials sent via: | |
|---|---|
| Plaintext POST parameters | 25 |
| Obfuscated (but decipherable) POST parameters | 1 |
| Obfuscated (possibly encrypted) POST parameters | 2 |
| URL Parameters | 2 |
| HTTP Headers | 7 |

# Continuing authentication

| Continuous session authentication via: | |
|---|---|
| Re-sending userid / password credentials | 2 |
| Dynamic tokens (OAuth 1.0, PKI, etc.) | 7 |
| Fixed tokens (long- or short-lived) | 28 |

| Session credentials sent via: | |
|---|---|
| URL parameters | 9 |
| POST body | 5 |
| HTTP headers | 24 |

# Renewed authentication

| After quitting and restarting the application, the app: | |
|---|---|
| Re-sends userid and password | 4 |
| Sends stored token | 28 |
| Asks user for password | 5 |
| Asks user for both userid and password | 1 |

# Local credential storage

| Location | Userid | Password | Session Tokens |
|---|---|---|---|
| Keychain | 11 | 5 | 14 |
| Preferences | 14 | 1 | 5 |
| Cookie File | 3 | 1 | 5 |
| HTTP Cache | 9 | 2 | 9 |
| /Documents | 11 | 0 | 4 |
| Other /Library | 3 | 0 | 4 |

- Six apps stored no passwords or tokens at all
- Passwords split between filesystem and keychain
- More tokens found on filesystem than in keychain

# Cool things

| Applications which included: | Userid |
|---|---|
| Great (and mostly accurate) TLS warning | 1 |
| Encrypted userid storage | 1 |
| Encrypted password delivery | 1+ |
| Totally encrypted data | 1 or 2 |
| Certificate based session | 1 or 2 |

# Suggestions

# Good Ideas

- Better storage and leak management
  - Store everything in the keychain
  - Work to avoid leakage to cache files
  - Unique tokens for session and renewed authentication
- Advantages:
  - Can't easily extract credentials from unlocked device
  - User can revoke device tokens from website
- Disadvantages:
  - Initial authentication can still be intercepted
  - Session tokens can be intercepted and reused

# Better

- Store only a hash of the password
  - HASH( HASH(password) + nonce + timestamp) )
  - Server refuses old tokens and re-used nonces
  - Refresh token regularly (weekly, or even daily)
- Advantages:
  - Actual password never sent over network
  - If token is compromised, automatically expires quickly
- Disadvantages:
  - Attacker can intercept request & return error to app
    - Use valid token with new request (password reset?)

# Best

- All packets get “one time” token
  - HASH( HASH(pw) + nonce + timestamp + HASH(req) )
  - Send userid, overall hash, nonce, timestamp
- Strengths:
  - Authentication tokens can’t be reused
  - Can’t modify or forge requests using unused tokens
- Weaknesses:
  - More complicated
  - Server needs to remember recent nonces

# Above and Beyond

- Use one-time tokens (as above)
- Encrypt all requests with public key for server
- Optionally, sign everything with client key

- Should be damned near impossible to intercept

- Extraction of hash from keychain remains a risk
  - For highest security, don’t even store hash
  - User enters password with each new application launch

# Weak links remain

- If you make the mobile app unbreakable…
- ….attackers will focus on browser-based interface
- Much harder to add these features to web apps

# Existing specifications

- OASIS / SAML
  - Timestamp, nonce, password hash, request hash
  - Only ever seen this once (on a customer’s app)
  - Concept is simple, spec is complex and probably scary
- OAuth 1.0
  - Also supports signed, authenticated requests
  - Developers thought it was too complicated
- OAuth 2.0
  - Greatly simplified - easier to implement
  - Doesn’t have as many features - not as strong

# Conclusion

# General conclusion

- Generally, the apps I tested were:

× Excellent  
× Pretty good  
✓ Not bad, but could be better  
× Scare the hell out of me

- Of 38 apps which completed review
  - 12 had only minor issues
  - 6 had at least one major issue
  - ***ZERO had no issues at all***
- A few simple fixes could kick most up a level

# Overall Scores?

- Again, not a complete assessment so not entirely fair
- If I had to categorize biggest issues seen:

| Severity | Description | |
|---|---|---|
| Low | Userid stored insecurely | 27 |
| | Application does not use certificate pinning | 36 |
| Medium | Credentials or tokens passed as URL parameters | 9 |
| | Tokens Stored Insecurely | 21 |
| | Application sends credentials for every request | 2 |
| High | Application does not use TLS or ignores cert errors | 2 |
| | Password stored insecurely | 4 |
| | *— NO MEDIUM OR HIGH —* | 12 |
| | *— NONE OF THE ABOVE —* | 0 |

# Top 5 Suggestions

- Use TLS certificate pinning
  - We rely on it, take extra steps to ensure it's reliable
- Store credential components only in keychain
  - Especially password or session tokens
  - If possible, avoid storing the password
- Always use strong hash generation (PBKDF2, HMAC)
- Take steps to avoid cache / cookie leakage
- If possible, use one-time (nonce/timestamp) tokens
  - Even better, tie to hash of request

# Future work

- Wider survey
  - More formal study of broader cross section
  - Or possibly focus on one or two app types
  - May be difficult if you need $$ accounts (banks, etc.)
- Other surveys
  - General security issues
  - Third-party data collection (analytics, crash reporting)
  - Third party support services (cloud storage, etc.)
- More detailed "best practices" recommendations

# Thanks!

## Europe

Manchester - Head Office
Cheltenham
Edinburgh
Leatherhead
London
Munich
Amsterdam
Zurich

## North America

Austin
Atlanta
Chicago
New York
San Francisco
Seattle
Sunnyvale

## Australia

Sydney