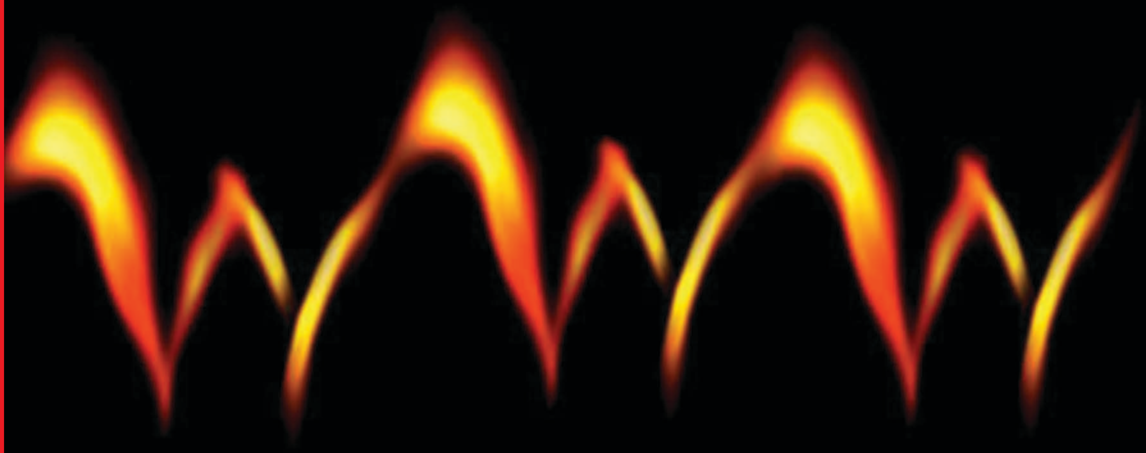
Four Attacks on OAuth - How to Secure Your OAuth Implementation

A technical study of an emerging open-protocol technology and its security implications.

SANS

Working
Papers
in
Application
Security



Four Attacks on OAuth – How to Secure Your OAuth Implementation

A technical study of an emerging open-protocol technology and its security implications

Volume 3.1

Written by Khash Kiani, khashsec@gmail.com

SANS

Working Papers in Application Security

Four Attacks on OAuth – How to Secure Your OAuth Implementation

Written by
Khash Kiani

This article is part of a series dedicated to application security and secure coding practices. In coming months, the SANS Institute will release additional articles like this that cover other aspects of application security and secure coding. However, please don't wait until the end of the series to start following these tips. Make a commitment today to ensuring that your applications and code are more secure.

www.sans.org/info/39149

This article briefly introduces OAuth with an example, and presents scenarios of how insecure implementations of OAuth can be abused maliciously. We examine the characteristics of some of these attack vectors, and discuss ideas on countermeasures against possible attacks on users or applications that have implemented this protocol.

An Introduction to the Protocol

OAuth is an emerging authorization standard that is being adopted by a growing number of sites such as Twitter, Facebook, Google, Yahoo!, Netflix, Flickr, and several other Resource Providers and social networking sites. It is an open-web specification for organizations to access protected resources on each other's web sites. This is achieved by allowing users to grant a third-party application access to their protected content without having to provide that application with their credentials.

Unlike Open ID, which is a federated authentication protocol, OAuth, which stands for Open Authorization, is intended for delegated authorization only and it does not attempt to address user authentication concerns.

There are several excellent online resources, referenced at the end of this article, that provide great material about the protocol and its use. However, we need to define a few key OAuth concepts that will be referenced throughout this article

Key Concepts

- **Server** or the **Resource Provider** controls all OAuth restrictions and is a website or web services API where User keeps her protected data
- **User** or the **Resource Owner** is a member of the Resource Provider, wanting to share certain resources with a third-party web site
- **Client** or **Consumer Application** is typically a web-based or mobile application that wants to access User's **Protected Resources**
- **Client Credentials** are the consumer key and consumer secret used to authenticate the Client
- **Token Credentials** are the access token and token secret used in place of User's username and password

Example

The actual business functionality in this example is real. However, the names of the organizations and users are fictional.

MyBillManager.com provides Avon Barksdale a bill consolidation service based on the trust Avon has established with various Resource Providers, including BaltimoreCellular.com.

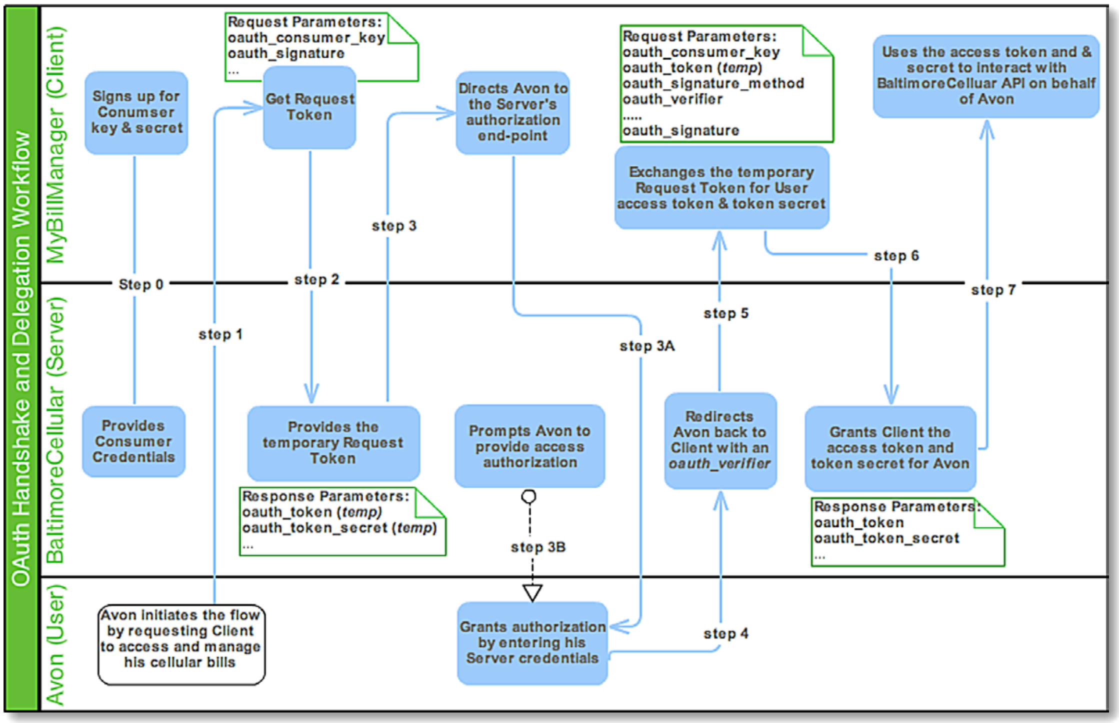
Working Papers in Application Security

Four Attacks on OAuth – How to Secure Your OAuth Implementation

A technical study of an emerging open-protocol technology and its security implications

Written by
Khash Kiani

In the diagram below, we have demonstrated a typical OAuth handshake (aka OAuth dance) and delegation workflow, which includes the perspectives of the User, Client, and the Server in our example:



OAuth Handshake and Delegation Workflow
MyBillManager (Client)
BaltimoreCellular (Server)
Avon (User)
Signs up for Conumser key & secret
Request Parameters: oauth_consumer_key oauth_signature ...
Get Request Token
Directs Avon to the Server's authorization end-point
Request Parameters: oauth_consumer_key oauth_token (temp) oauth_signature_method oauth_verifier ..... oauth_signature
Exchanges the temporary Request Token for User access token & token secret
Uses the access token and & secret to interact with BaltimoreCelluar API on behalf of Avon
Step 0
step 1
step 2
step 3
step 3A
step 3B
step 4
step 5
step 6
step 7
Provides Consumer Credentials
Provides the temporary Request Token
Response Parameters: oauth_token (temp) oauth_token_secret (temp) ...
Prompts Avon to provide access authorization
Redirects Avon back to Client with an oauth_verifier
Grants Client the access token and token secret for Avon
Response Parameters: oauth_token oauth_token_secret ...
Avon initiates the flow by requesting Client to access and manage his cellular bills
Grants authorization by entering his Server credentials

Figure 1: The entire OAuth1.0 Process Workflow

Insecure Implementations and Potential Solutions

In this section, we will examine some of the security challenges and insecure implementations of the protocol, and provide solution ideas. It is important to note that like most other protocols, OAuth does not provide native security nor does it guarantee the privacy of protected data. It relies on the implementers of OAuth, and other protocols such as SSL, to protect the exchange of data amongst parties. Consequently, most security risks described below do not reside within the protocol itself, but rather its use.

1. Lack Of Data Confidentiality and Server Trust

While analyzing the specification, and the way OAuth leverages the keys, tokens, and secrets to establish the digital signatures and secure the requests, one realizes that much of the effort put into this protocol was to avoid the need to use HTTPS requests all together. The specification attempts to provide fairly robust schemes to ensure the integrity of a request using its signatures; however it cannot guarantee a request's confidentiality, and that could result in several threats, some of which are listed below.

1.1 Brute Force Attacks Against the Server

An attacker with access to the network will be able to eavesdrop on the traffic and gain access to the specific request parameters and attributes such as oauth_signature, consumer_key, oauth_token, signature_method (HMAC-SHA1), timestamp, or custom parameter data. These values could assist the attacker in gaining enough understanding of the request to craft a packet and launch a brute force attack against the Server. Creating tokens and shared-secrets that are long, random and resistant to these types of attacks can reduce this threat.

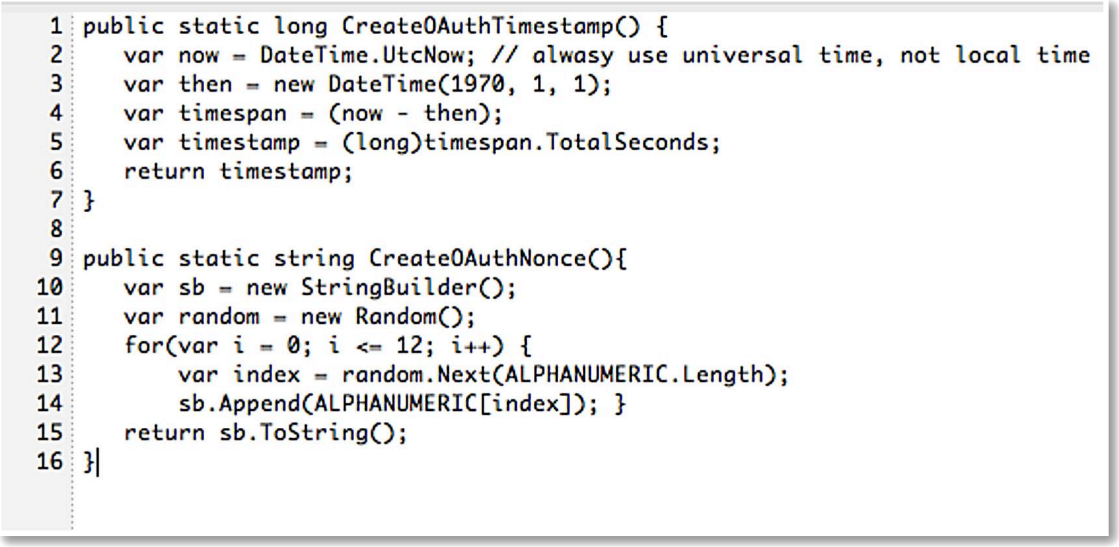
1.2 Lack of Server Trust

This protocol is all about authenticating the Client (consumer key and secret) and the User to the Server, but not the other way around. There is no protocol support to check the authenticity of the Server during the handshakes. So essentially, through phishing or other exploits, user requests can be directed to a malicious Server where the User can receive malicious or misleading payloads. This could adversely impact the Users, but also the Client and Server in terms of their credibility and bottom-line.

1.3 Solutions

As we have seen, the OAuth signature methods were primarily designed for insecure communications, mainly non-HTTPS. Therefore, TLS/SSL is the recommended approach to prevent any eavesdropping during the data exchange.

Furthermore, Resource Providers can limit the likelihood of a replay attack from a tampered request by implementing protocol's Nonce and Timestamp attributes. The value of oauth_nonce attribute is a randomly generated number to sign the Client request, and the oauth_timestamp defines the retention timeframe of the Nonce. The following example from Twitter .NET Development for OAuth Integration demonstrates the creation of these attributes by the application:



```
public static long CreateOAuthTimestamp() {
   var now = DateTime.UtcNow; // alwasy use universal time, not local time
   var then = new DateTime(1970, 1, 1);
   var timespan = (now - then);
   var timestamp = (long)timespan.TotalSeconds;
   return timestamp;
}

public static string CreateOAuthNonce(){
   var sb = new StringBuilder();
   var random = new Random();
   for(var i = 0; i <= 12; i++) {
       var index = random.Next(ALPHANUMERIC.Length);
       sb.Append(ALPHANUMERIC[index]); }
   return sb.ToString();
}
```

Figure 2: Sample oauth_nonce and oauth_timestamp methods

2. Insecure Storage of Secrets

The two areas of concern are to protect:

- Shared secrets on the Server
- Consumer secrets on cross-platform clients

2.1 Servers

On the Server, in order to compute the oauth_signature, the Server must be able to access the shared-secrets (a signed combination of consumer secret and token secret) in plaintext format as opposed to a hashed value. Naturally, if the Server and all its shared-secrets were to be compromised via physical access or social engineering exploits, the attacker could own all the credentials and act on behalf of any Resource Owner of the compromised Server.

2.2 Clients

OAuth Clients use the consumer key and consumer secret combination to provide their authenticity to the Server. This allows:

- Clients to uniquely identify themselves to the Server, giving the Resource Provider the ability to keep track of the source of all requests
- The Server to let the User know which Client application is attempting to gain access to their account and protected resource

Securing the consumer secret on browser-based web application clients introduces the same exact challenges as securing shared-secrets on the Server. However, the installed mobile and desktop applications become much more problematic:

OAuth's dependency on browser-based authorization creates an inherit implementation problem for mobile or desktop applications that by default do not run in the User's browser. Moreover, from a pure security perspective, the main concern is when implementers store and obfuscate the key/secret combination in the Client application itself. This makes the key-rotation nearly impossible and enables unauthorized access to the decompiled source code or binary where the consumer secret is stored. For instance, to compromise the Client Credentials for Twitter's Client on Android, an attacker can simply disassemble the classes.dex with Android dissembler tool, dexdump:

```
“  dexdump –d classes.dex
```

2.3 The Threats

It is important to understand that the core function of the consumer secret is to let the Server know which Client is making the request. So essentially, a compromised consumer secret does NOT directly grant access to User's protected data. However, compromised consumer credentials could lead to the following security threats:

- In use cases where the Server MUST keep track of all Clients and their authenticity to fulfill a business requirement, (charge the Client for each User request, or for client application provisioning tasks) safeguarding consumer credential becomes critical
- The attacker can use the compromised Client Credentials to imitate a valid Client and launch a phishing attack, where he can submit a request to the Server on behalf of the victim and gain access to sensitive data

Regardless of the use case, whenever the consumer secret of a popular desktop or mobile Client application is compromised, the Server must revoke access for ALL users of the compromised Client application. The Client must then register for a new key (a lengthy process), embed it into the application as part of a new release and deploy it to all its Users. A nightmarish process that could take weeks to restore service; and will surely impact the Client's credibility and business objectives.

2.4 Solutions

Protecting the integrity of the Client Credentials and Token Credentials works fairly well when it comes to storing them on servers. The secrets can be isolated and stored in a database or file-system with proper access control, file permission, physical security, and even database or disk encryption.

For securing Client Credentials on mobile application clients, follow security best practices for storing sensitive, non-stale data such as application passwords and secrets.

The majority of current OAuth mobile and desktop Client applications embed the Client Credentials directly into the application. This solution leaves a lot to be desired on the security front.

Few alternative implementations have attempted to reduce the security risks by obfuscation or simply shifting the security threat elsewhere:

- Obfuscate the consumer secret by splitting it into segments or shifting characters by an offset, then embed it in the application
- Store the Client Credentials on the Client's backend server. The credentials can then be negotiated with the front-end application prior to the Client/Server handshake. However, nothing is going to stop a rogue client to retrieve the Client Credentials from application's back-end server, making this design fundamentally unsound

Let's consider a better architectural concept that might require some deviation from the typical OAuth flow:

- The Service Provider could require certain Clients to bind their Consumer Credentials with a device-specific identifier (similar to a session id). Prior to the initial OAuth handshake, the mobile or desktop application can authenticate the User to the Client application via username and password. The mobile Client can then call home to retrieve the Device ID from the Client's back-end server and store it securely on the device itself (e.g. iOS Keychain). Once the initial request is submitted to the Serve with both the Client Credentials and Device ID, the Service Provider can validate the authenticity of the Device ID against the Client's back-end server.

The example below illustrates this solution:

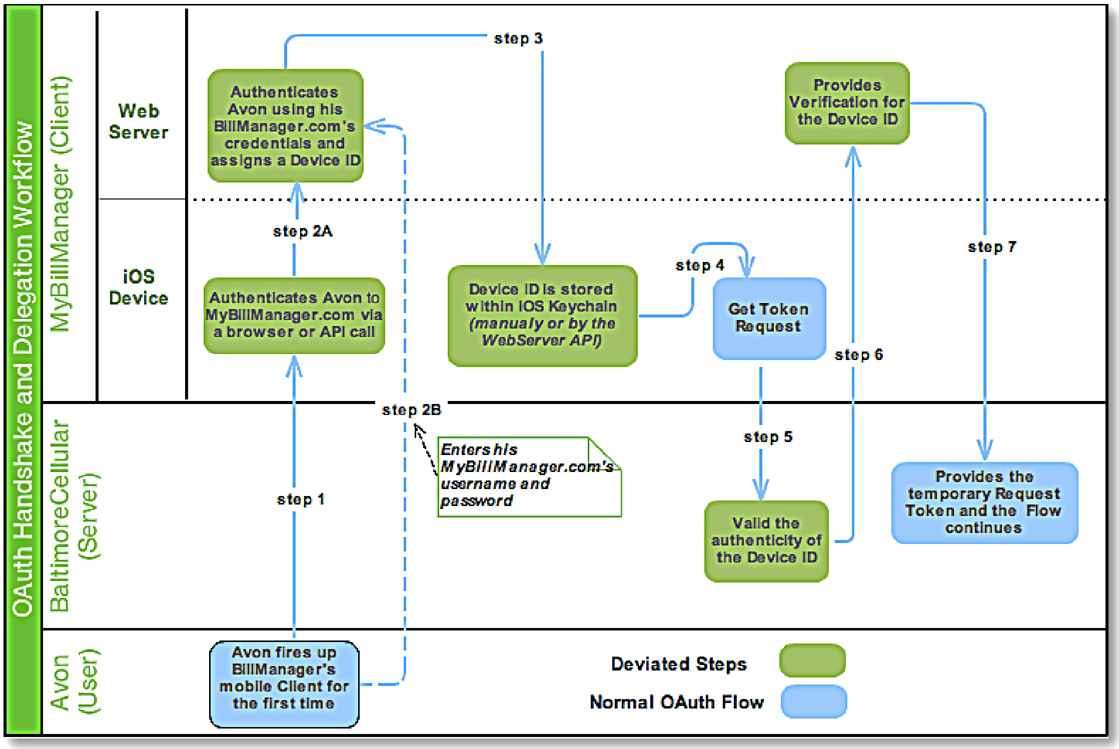

OAuth Handshake and Delegation Workflow
MyBillManager (Client)
Web Server
iOS Device
BaltimoreCellular (Server)
Avon (User)
step 3
Authenticates Avon using his BillManager.com's credentials and assigns a Device ID
Provides Verification for the Device ID
step 2A
step 4
step 7
Authenticates Avon to MyBillManager.com via a browser or API call
Device ID is stored within iOS Keychain (manualy or by the WebServer API)
Get Token Request
step 6
step 2B
Enters his MyBillManager.com's username and password
step 5
step 1
Valid the authenticity of the Device ID
Provides the temporary Request Token and the Flow continues
Avon fires up BillManager's mobile Client for the first time
Deviated Steps
Normal OAuth Flow

Figure 3: alternative solution for authenticating OAuth Clients

OAuth's strength is that it never exposes a User's Server credentials to the Client application. Instead, it provides the Client application with temporary access authorization that User can revoke if necessary. So ultimately, regardless of the solution, when the Server cannot be sure of the authenticity of the Client's key/secret, it should not solely rely on these attributes to validate the Client.

3. OAuth Implementation with Flawed Session Management

As described in the first example *(see figure 1)*, during the authorization step, the User is prompted by the Server to enter her login credentials to grant permission. The user then is redirected back to the Client application to complete the flow. The main issue is with a specific OAuth Server implementation, such as Twitter's, where the user remains logged in on the Server even after leaving the Client application.

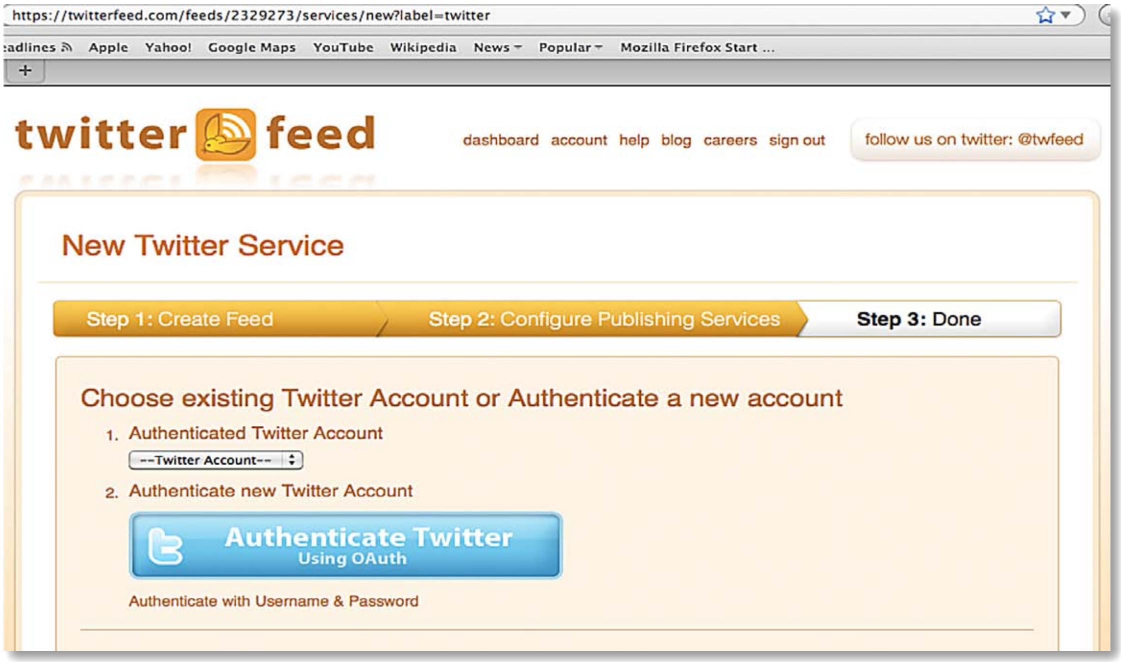
This session management issue, in-conjunction with Server's implementation of OAuth Auto Processing, to automatically process authorization requests from clients that have been previously authorized by the Server, present serious security concerns.

Let's take a closer look at Twitter's implementation:

There are numerous third-party Twitter applications to read or send Tweets; and as of August of 2010, all third-party Twitter applications had to exclusively use OAuth for their delegation-based integration with Twitter APIs. Here is an example of this flawed implementation:

twittered *is a popular application that allows blog feeds to user's Twitter accounts.*

Avon Barksdale registers and signs into his twitterfeed client. He then selects a publishing service such as Twitter to post his blog.



https://twitterfeed.com/feeds/2329273/services/new?label=twitter

New Twitter Service

Step 1: Create Feed — Step 2: Configure Publishing Services — Step 3: Done

Choose existing Twitter Account or Authenticate a new account

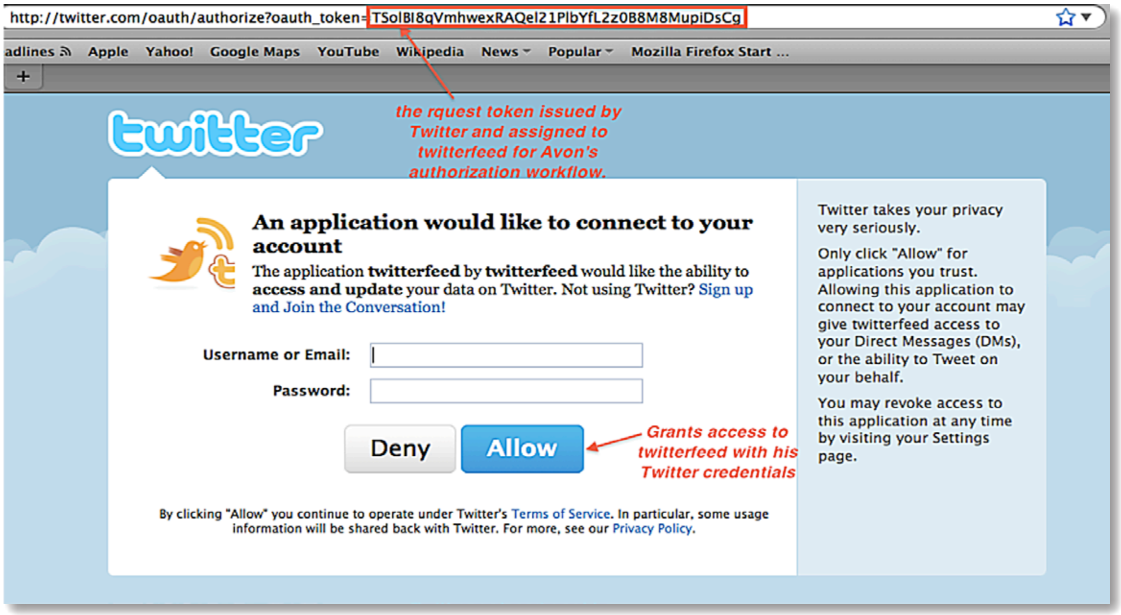
1. Authenticated Twitter Account
--Twitter Account--

2. Authenticate new Twitter Account

Authenticate with Username & Password

Figure 4: twitterfeed.com client authorization page

Avon is directed to Twitter's authorization endpoint where he signs into Twitter with his username and password and grants access to twitterfeed.



http://twitter.com/oauth/authorize?oauth_token=TSolBl8gVmhwexRAQel21PlbYfL2z0B8M8MupiDsCg

the rquest token issued by Twitter and assigned to twitterfeed for Avon's authorization workflow.

An application would like to connect to your account

The application **twitterfeed** by **twitterfeed** would like the ability to **access and update** your data on Twitter. Not using Twitter? Sign up and Join the Conversation!

Username or Email:

Password:

Grants access to twitterfeed with his Twitter credentials

By clicking "Allow" you continue to operate under Twitter's Terms of Service. In particular, some usage information will be shared back with Twitter. For more, see our Privacy Policy.

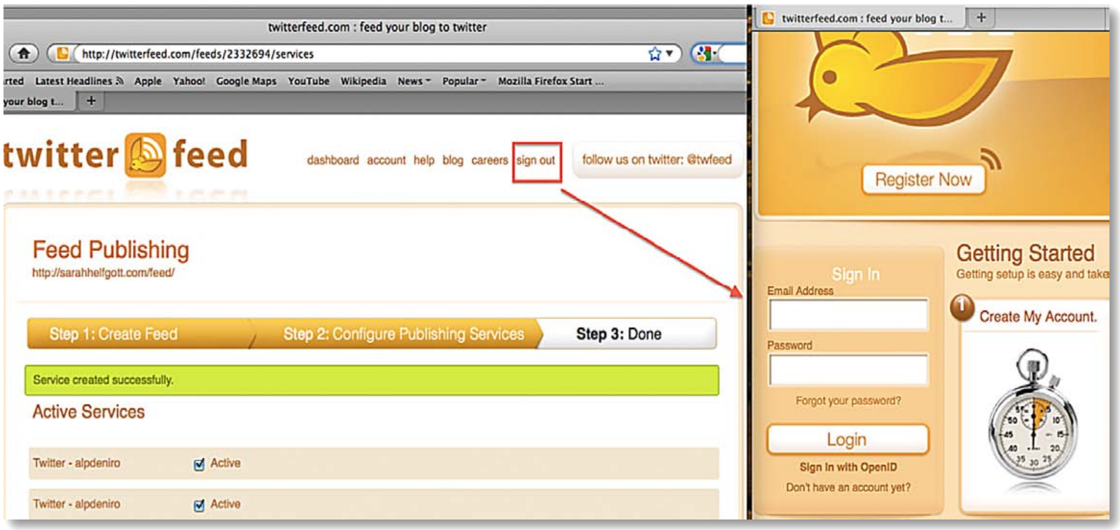
Twitter takes your privacy very seriously.

Only click "Allow" for applications you trust. Allowing this application to connect to your account may give twitterfeed access to your Direct Messages (DMs), or the ability to Tweet on your behalf.

You may revoke access to this application at any time by visiting your Settings page.

Figure 5: twitter's authorization page for third-party applications

Now, Avon is redirected back to twitterfeed where he completes the feed; and then signs out of twitterfeed and walks away.



twitterfeed.com : feed your blog to twitter

http://twitterfeed.com/feeds/2332694/services

Feed Publishing
http://sarahhelfgott.com/feed/

Step 1: Create Feed | Step 2: Configure Publishing Services | Step 3: Done

Service created successfully.

Active Services

Twitter - alpdeniro ☑ Active

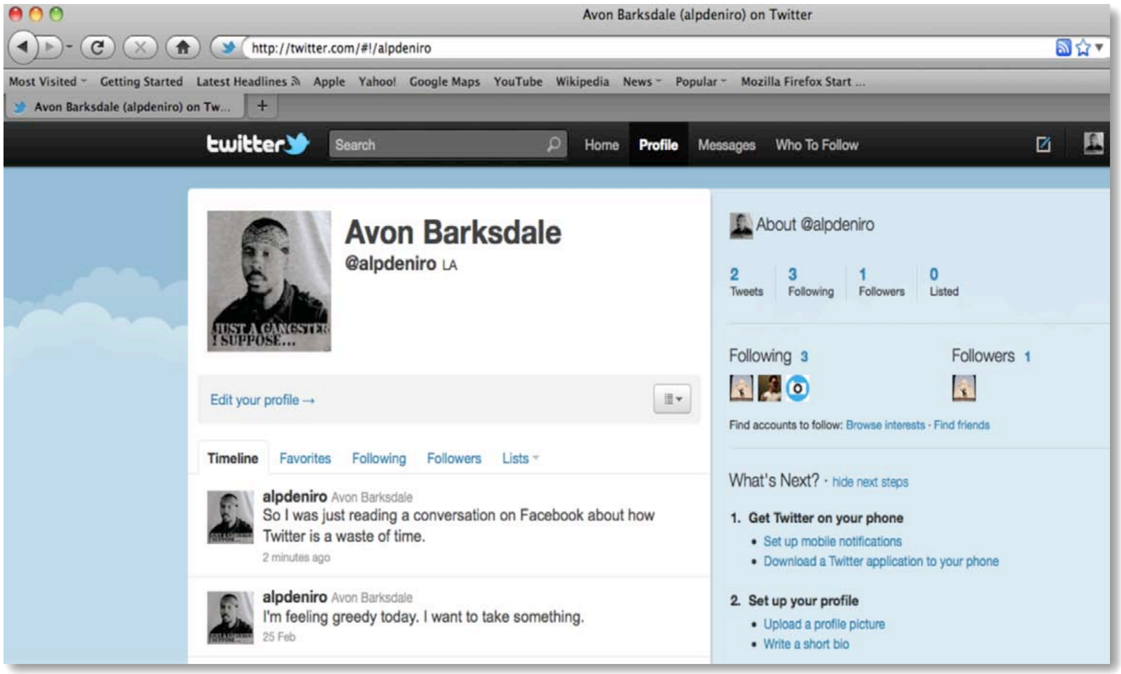
Twitter - alpdeniro ☑ Active

Sign In
Email Address
Password
Forgot your password?
Login
Sign In with OpenID
Don't have an account yet?

Getting Started
Getting setup is easy and take
1 Create My Account.

Figure 6: twitterfeed's workflow page post server authorization

A malicious user with access to the unattended browser can now fully compromise Avon's Twitter account; and deal with the consequences of his action!



Avon Barksdale (alpdeniro) on Twitter

http://twitter.com/#!/alpdeniro

Avon Barksdale
@alpdeniro LA

alpdeniro Avon Barksdale
So I was just reading a conversation on Facebook about how Twitter is a waste of time.
2 minutes ago

alpdeniro Avon Barksdale
I'm feeling greedy today. I want to take something.
25 Feb

About @alpdeniro

2 Tweets · 3 Following · 1 Followers · 0 Listed

Following 3 · Followers 1

Find accounts to follow: Browse interests · Find friends

What's Next? · hide next steps

1. Get Twitter on your phone
   - Set up mobile notifications
   - Download a Twitter application to your phone

2. Set up your profile
   - Upload a profile picture
   - Write a short bio

Figure 7: Twitter session remains valid in the background after user signs out of twitterfeed.com

3.1 The Threat

The User might not be aware that he has a Server session open with the Resource Provider in the background. He could simply just log out of his Client session and step away from his browser. The threat is elevated when leaving a session unattended becomes inevitable in use cases where public computers are used to access OAuth-enabled APIs. Imagine if Avon was a student who frequently accessed public computers for his daily tweet feeds. He could literally leave a Twitter session on every computer he uses.

3.2 Solutions

Resource Providers such as Twitter should always log the User out after handling the third-party OAuth authorization flow in situations where the User was not already logged into the Server before the OAuth initiation request.

Auto Processing should be turned off. That is, servers should not automatically process requests from clients that have been previously authorized by the resource owner. If the consumer secret is compromised, a rogue Client can gain ongoing unauthorized access to protected resources without the User's explicit approval.

4. Session Fixation Attack with OAuth

OAuth key contributor, Eran Hammer–Lahav, has published a detailed post, referenced at the end of this article, about this specific attack that caused a major disruption to many OAuth consumers and providers. For instance, Twitter had to turn-off its third-party integration APIs for an extended period of time, impacting its users as well as all the third-party applications that depended on Twitter APIs.

In summary, the Session Fixation flaw makes it possible for an attacker to use social-engineering tactics to lure users into exposing their data via few simple steps:

- Attacker uses a valid Client app to initiate a request to the Resource Provider to obtain a temporary Request Token. He then receives a redirect URI with this token:
  - http://<resource_provider.com>/oauth/authorize?oauth_token=XyZ
- At a later time, the attacker uses social-engineering and phishing tactics to lure a victim to follow the redirect link with the server-provided Request Token
- The victim follows the link, and grants client access to protected resources. This process authorizes the Request Token and associates it with the Resource Owner

The above steps demonstrate that the Resource Provider has no way of knowing whose Request Token is being authorized, and cannot distinguish between the two users.

- The Attacker constructs the callback URI with the "authorized" Access Token and returns to the Client:
  - http://<client_app.com>/feeds/…/oauth_token= XyZ&…
- If constructed properly, the Attacker's client account is now associated with victim's authorized Access Token

The vulnerability is that there is no way for the Server to know whose key it is authorizing during the handshake. The author describes the attack in detail and proposes risk-reducing solutions. Also, the new version of the protocol, OAuth2.0, attempts to remediate this issue via its Redirect URI being validated with the authorization key exchange.

Summary

As the web grows, more and more sites rely on distributed services and cloud computing. And in today's integrated web, users demand more functionality in terms of usability, cross-platform integration, cross-channel experiences and so on. It is up to the implementers and security professionals to safeguard user and organizational data and ensure business continuity. The implementers should not rely on the protocol to provide all security measures, but instead they should be careful to consider all avenues of attack exposed by the protocol, and design their applications accordingly.

References

Protocol Specification
http://tools.ietf.org/html/rfc5849

OAuth Extensions and Code Sample
http://oauth.net/code/

Twitter API References
http://dev.twitter.com/pages/oauth_faq

OAuth session fixation
http://hueniverse.com/2009/04/explaining-the-oauth-session-fixation-attack

Four Attacks on OAuth – How to Secure Your OAuth Implementation

A technical study of an emerging open-protocol technology and its security implications

Written by
Khash Kiani

OAuth2.0 Specification: http://tools.ietf.org/html/draft-ietf-oauth-v2-13

About the Author

Khash Kiani is a senior security consultant at a large health care organization. He has been designing, securing and implementing software applications for over 13 years. Khash specializes in security architecture, application penetration testing, PCI and social-engineering assessments. He currently holds the GIAC GWAPT, GCIH, and GSNA certifications. Khash can be reached at khashsec@gmail.com

Last Updated: April 17th, 2015

Upcoming SANS Training

Click Here for a full list of all Upcoming SANS Events by Location

Security Operations Center Summit & Training	**Washington, DCUS**	**Apr 24, 2015 - May 01, 2015**	**Live Event**
SANS ICS London 2015	**London, GB**	**Apr 27, 2015 - May 02, 2015**	**Live Event**
SANS SEC401 London	**London, GB**	**Apr 27, 2015 - May 02, 2015**	**Live Event**
SANS Bahrain 2015	**Manama, BH**	**May 02, 2015 - May 07, 2015**	**Live Event**
SANS Security West 2015	**San Diego, CAUS**	**May 03, 2015 - May 12, 2015**	**Live Event**
SANS Secure India 2015	**Bangalore, IN**	**May 04, 2015 - May 23, 2015**	**Live Event**
SANS Secure Europe 2015	**Amsterdam, NL**	**May 05, 2015 - May 25, 2015**	**Live Event**
SANS/NH-ISAC Healthcare Cybersecurity Summit	**Atlanta, GAUS**	**May 12, 2015 - May 19, 2015**	**Live Event**
SANS Melbourne 2015	**Melbourne, AU**	**May 18, 2015 - May 23, 2015**	**Live Event**
SANS Pen Test Austin 2015	**Austin, TXUS**	**May 18, 2015 - May 23, 2015**	**Live Event**
SANS Secure Thailand 2015	**Bangkok, TH**	**May 25, 2015 - May 30, 2015**	**Live Event**
SANS ICS Security Training - Houston	**Houston, TXUS**	**Jun 01, 2015 - Jun 05, 2015**	**Live Event**
SANS ICS410 Vienna in Association with IAEA	**Vienna, AT**	**Jun 06, 2015 - Jun 10, 2015**	**Live Event**
SANS Dublin 2015	**Dublin, IE**	**Jun 08, 2015 - Jun 13, 2015**	**Live Event**
SANSFIRE 2015	**Baltimore, MDUS**	**Jun 13, 2015 - Jun 20, 2015**	**Live Event**
SANS Rocky Mountain 2015	**Denver, COUS**	**Jun 22, 2015 - Jun 27, 2015**	**Live Event**
SANS Pen Test Berlin 2015	**Berlin, DE**	**Jun 22, 2015 - Jun 27, 2015**	**Live Event**
Cyber Defence Canberra 2015	**Canberra, AU**	**Jun 29, 2015 - Jul 11, 2015**	**Live Event**
SANS Capital City 2015	**Washington, DCUS**	**Jul 06, 2015 - Jul 11, 2015**	**Live Event**
Digital Forensics & Incident Response Summit	**Austin, TXUS**	**Jul 07, 2015 - Jul 14, 2015**	**Live Event**
European Security Awareness Summit	**London, GB**	**Jul 08, 2015 - Jul 10, 2015**	**Live Event**
SANS London in the Summer	**London, GB**	**Jul 13, 2015 - Jul 18, 2015**	**Live Event**
RSA Conference 2015	**OnlineCAUS**	**Apr 19, 2015 - Apr 22, 2015**	**Live Event**
SANS OnDemand	**Books & MP3s OnlyUS**	**Anytime**	**Self Paced**