OAuth WRAP 2.0 Security Considerations

Background

This is a draft, originally written on April 2, 2010, updated on April 22, 2010.

It doesn't cover the JS profile we've discussed.

I haven't read Eran's latest comments.

I'm also way behind on the mailing list threads.

This is a draft.

It probably has some mistakes, and it may have lots of mistakes.

I've included vague references to other forms of authentication besides the bearer token approach in WRAP.

Did I mention this is a draft?

References

draft-hard-oauth-01
draft-recordon-oauth2
draft-barnes-oauth-model-01.
WRAP 0.9.7.2
http://tools.ietf.org/html/rfc4949
http://oauth.net/core/1.0a/

OAuth WRAP Security Overview

Note: the definitions section here overlaps a lot with both draft-hard-oauth-01, WRAP 0.9.7.2, and draft-recordon-oauth2.

OAuth WRAP consists of a number of profiles and patterns for securely distributing and using authorization tokens. Different environments have different security characteristics, and OAuth WRAP tries to leverage strong authentication mechanisms where they exist, and mitigate authentication weaknesses where necessary. Minimizing key distribution problems is in scope for OAuth WRAP. Recovery from key compromise is also in scope. The specification looks for practical trade-offs in security, performance, and deployability.

Security Principals

There are several relevant security principals in OAuth WRAP:

Users: these are people who must authorize Client access to Protected Resource Servers.

Authorization Server: a third-party trusted by various Protected Resources, by Users, and by Clients.

Protected Resources: API servers that hold data.

Clients: these are devices or applications that wish to access Protected Resources. Clients may run in a variety of environments, including web servers, web pages without server-side support, installed applications on a user's computer, or applications running on servers. Different clients have different security characteristics and capabilities, and OAuth WRAP typically recommends different profiles for different environments.

Authentication Techniques

WRAP has facilities to leverage strong authentication systems where they exist, and recommends alternatives where strong authentication does not exist today. The following authentication techniques are used frequently within the specification.

HTTPS server authentication: HTTPS is used to provide confidentiality and integrity of data sent between users, clients, and servers. Servers are authenticated using HTTPS server certificates. Users and clients authenticate through other means.

Callback URLs: Web servers may direct clients to send near-arbitrary data to a particular URL, relying on the client to properly authenticate the destination server before sending the data. Data sent via callback URL is prone to leak in some environments. For example, browsers may leak callback URLs in referer headers, or in the browser history. Callback URLs may also leak in web server log files. The primary security advantage of callback URLs is that they leverage existing authentication and key distribution infrastructure, i.e. the PKI used for HTTPS.

Client identifier and client secret: Clients may authenticate using other means, but common practice today is for clients to authenticate using passwords sent over secure channels. Client secrets are high-entropy passwords, so they are not generally vulnerable to brute-force attacks. They are also not generally vulnerable to phishing attacks because clients are automated programs, not human beings. The security of the client secret generally depends on the security of the systems with access to the secret.

User names and user passwords: Users may authenticate using other means, but common practice today is for users to authenticate using passwords sent over secure channels. User passwords are not typically high-entropy, and so may be vulnerable to brute-force attacks. User passwords may also be phished, or stolen by malware, or may leak through a variety of other systems. A primary goal of the OAuth protocol is eliminating dependencies on passwords for user authentication.

Refresh tokens: refresh tokens represent long-lived capabilities to access data. They may also be bound to client identifiers and client secrets, in the sense that clients may be required to authenticate before using the refresh token. Refresh tokens are minted and verified by Authorization Servers.

Access tokens: access tokens represent short-lived capabilities to access data. They are not bound to client identifiers. Access tokens are minted by Authorization servers, and verified by Protected Resource servers.

Automatic Processing of Authorizations: some Authorization Servers may wish to automatically process authorization requests from clients which have been previously authorized by the user. When the User is redirected to the Authorization Server to grant access, the Authorization Server detects that the User has already granted access to that particular Client. Instead of prompting the User for approval, the Authorization Server automatically redirects the User back to the Provider.

Automatic processing creates additional security risks by removing an element of social engineering from attacks. If no automatic approval is implemented, the attacker must convince the User to approve access.

Risks from automatic approvals can be mitigated either via white listing of callback URLs or by authentication of clients, or both. If neither technique is possible, Authorization Servers MUST NOT implement automatic approvals. Authorization Servers MAY mitigate the risks associated with automatic processing by limiting the scope of Access Tokens obtained through automated approvals.

Note: flesh this out to cover cryptographic signatures once we have something better than OAuth 1.0 signatures.

Client Account and Password Profile - Security Considerations

Note: I am basing this on the client account and password profile (section 5.1) from WRAP 0.9.7.2. I think the version in draft-hard-oauth-01 mixed up whether a refresh token was necessary in this profile. I'm assuming that this profile returns an access token, and not a refresh token.

This profile is intended for use by automated tasks that authenticate with passwords. Provisioning steps are assumed to involve human beings, but users are not involved with other steps.

The security of this profile depends entirely on the secrecy of the client secret and the access token.

Clients MUST NOT send client secrets to untrusted servers. Clients MUST use HTTPS to authenticate the Authorization Server and protect the confidentiality of the client secret in transit.

Authorization Servers SHOULD use cryptographic hash functions to store one-way hashes of client secrets instead of plaintext.

Client secrets SHOULD contain sufficient entropy that they can withstand off-line brute-force attacks on the hashed passwords.

In the event of compromise of the client secret, the client secret must be changed. When client secrets are changed Authorization Servers MAY revoke access tokens that were issued based on the old client secret. Alternatively, Authorization Servers MAY allow such access tokens to expire naturally.

Note: access tokens probably merit separate security considerations of their own.

Assertion Profile - Security Considerations

Note: this is based on the assertion profile from draft-hard-oauth-01: http://tools.ietf.org/html/draft-hardt-oauth-01#section-5.2

The security of this profile depends entirely on how Clients obtain assertions and how Authorization Servers verify assertions. These choices are largely deployment considerations. This profile exists to allow Clients and Authorization Servers to leverage preexisting trust relationships where they exist.

Clients obtain assertions using whatever security systems are available to them in their deployment environment. For example, they might read a username and password out of a file they store securely, and then present that password to a local security service. They might authenticate to a local security service using Kerberos or IPSec. They might read a private key from disk and create an assertion themselves.

Specifying exact details of how the local authentication happens is not required for interoperability across Clients and Authorization Servers, and so is out of scope of this specification.

After the local authentication steps are complete, the Client has an assertion that is trusted by the Authorization Server.

Clients MUST NOT send assertions to untrusted servers. Clients MUST use HTTPS to authenticate the Authorization Server and protect the confidentiality of the assertion in transit.

If the trust relationship used to create and verify assertions changes (e.g. the key used to sign assertions is rotated), Authorization Servers MAY revoke access tokens that were issued based on the previous trust relatonship. Alternatively, Authorization Servers MAY allow such access tokens to expire naturally.

Note: I think it is very likely that some deployments will have a a local security service that completes the entire Assertion Profile flow on behalf of the Client, and then returns just the access token to the client. Is this worth describing in more detail?

Username and Password Delegation Profile - Security Considerations

Note: several variants of the client + username + password profile have been suggested. I'm basing this section on the variant proposed on the IETF mailing list in early March 2010. There is also interest in supporting a third variant for installed applications, but I am too lazy to write that up right now.

Actually, this whole profile seems to be one of the more controversial ones. Lots of people need something like this and have differing opinions on the details. I'm going to make a bunch of assumptions just so I can write this section:

Assumption #1: the client is a server that is capable of keeping secrets.

Assumption #2: those secrets can be changed.

Assumption #3: the client needs long-lived access to user data.

Assumption #4: no client is going to bother implementing full captcha handling.

Assumption #5: authorization servers are still going to need to rate-limit.

Assumption #6: clients are going to need to tell their users *something* when the authorization servers rate limit password authentication attempts.

Assumption #7: authorization servers will return a URL to the client. The client should tell the user to vist that URL for more information about why their account is locked and authentication is failing.

The username and password delegation profile is intended to be used by Client servers that the Authorization Server trusts to request usernames and passwords from Users. The desired end-goal is for the Client to obtain long-term access to the User's data without needing to save the User's password.

Provisioning

Clients are assumed to know which Authorization Server they need to contact to exchange the username and password for a refresh token.

Client secrets SHOULD contain sufficient entropy that they can withstand off-line brute-force attacks on the hashed passwords.

Authorization Servers SHOULD use cryptographic hash functions to store one-way hashes of client secrets instead of plaintext.

Salted hashes or other password storage best practices SHOULD be used for storage of User passwords.

Processing Usernames and Passwords

The first step in the protocol flow is to send the client id, client secret, username, and password from the Client server to the Authorization Server.

Clients MUST NOT send client secrets or user passwords to untrusted servers. Clients MUST use HTTPS to authenticate the Authorization Server and protect the confidentiality of the secrets in transit.

The Authorization Server MUST verify the Client id, Client secret, username, and password.

Authorization Servers MAY rate-limit authentication attempts to mitigate the risk of brute-force attacks on usernames and passwords.

Generating Refresh Tokens and Access Tokens

After the Authorization Server has verified the Client and User identities, the Authorization Server returns a Refresh Token and Access Token to the client.

Authorization Servers SHOULD provide Users with the ability to revoke access granted to the Client.

Authorization Servers SHOULD use cryptographic hash functions to store one-way hashes of refresh tokens instead of plaintext.

Refresh tokens SHOULD contain sufficient entropy that they can withstand off-line brute-force attacks on hashed tokens.

Refresh tokens MUST be bound to particular Client ids.

Note: refresh tokens probably merit separate security considerations of their own.

Refreshing Access Tokens

Access tokens are assumed to expire. Clients need to present their Client Secret and Refresh

Token in order to obtain new access tokens.

Authorization Servers MUST verify that the Client id and Client secret are valid before issuing new Access Tokens. Authorization Servers MUST also verify that the Client id to which the refresh token was issued matches the Client id that is requesting a fresh access token.

Recovering from Compromise

An attacker who gains access to an Access Token gains temporary access to user data, until the access token expires or is revoked.

An attacker who gains access to a Refresh Token and Client Secret gains permanent access to user data.

If the Client secret can be changed, changing the client secret is sufficient for recovery. Revoking Refresh Tokens is not necessary. When client secrets are changed Authorization Servers MAY revoke access tokens that were issued based on the old client secret. Alternatively, Authorization Servers MAY allow such access tokens to expire naturally.

If the Client secret cannot be changed, compromised refresh tokens MUST be revoked.

Web App Delegation Profile - Security Considerations

Note: looks draft-recordon-oauth2 dropped some of the security checks included in draft-hard-oauth-01? Why?

Note: I disagree with some of the strategies below, for various reasons, but all of these approaches are widely deployed in various environments. I'm trying to offer pros and cons of existing practice, and I swear I'm not making any of this up. =)

Note: how should we talk about third-party web servers vs Clients here? The web site is one component of the client, but may not be the only component.

This profile is intended for use by a web server calling the protected resource on behalf of a resource owner. This involves the authorization server passing a token, via the browser, to the third-party web site. Tokens passed via browsers are prone to leaking via referer headers, open redirectors, request logs, and in the browser history. Multiple techniques are available for verifying the identity of the third-party web site and making this exchange more secure. These techniques can be combined for better security.

The three primary techniques described here are:

- verification codes in callback URLs
- callback URL whitelisting

- client secrets issued by the authorization server to the client

Other means of authenticating clients (e.g. client certificates) may be used, but are out of scope of this specification.

Verification Code

The protocol requires that the Authorization Server create an unpredictable verification code and pass the verification code to the Client web server via the browser. Passing these codes via the browser is critical for the security of the protocol (failure to include a verification code led to the session fixation attack against OAuth 1.0). However, browsers may unintentionally leak these codes to untrusted web sites. Verification codes may leak through a variety of mechanisms.

Referer headers: browsers frequently pass a “referer” header when a web page embeds content, or when a user travels from one web page to another web page. These referer headers may be sent even when the origin site does not trust the destination site. The referer header is commonly logged for traffic analysis purposes.

Request logs: web server request logs commonly include query parameters on requests.

Open redirectors: web sites sometimes need to send users to another destination via a redirector. Open redirectors pose a particular risk to web-based delegation protocols because the redirector can leak verification codes to untrusted destination sites.

Browser history: web browsers commonly record visited URLs in the browser history. Another user of the same web browser may be able to view URLs that were visited by previous users.

Because of the sheer variety of ways a verification code may leak, multiple layers of defense are necessary.

Verification codes MUST be time-limited. Authorization Servers SHOULD reject verification codes that are older than a few minutes.

Verification codes MAY be single-use tokens.

If an Authorization Server observes multiple attempts to redeem a verification code, the Authorization Server MAY revoke all tokens granted based on the verification code.

Aside from risk of leaking, verification codes MUST be unpredictable. If an attacker can guess a verification code, they may be able to redeem the verification code to gain access to user data.

Authorization Servers SHOULD implement other security mechanisms such as callback URL white listing or Client authentication to protect against leaked verification codes.

Callback URL White Listing

Authorization servers may white list the callback URLs of trusted partners. White lists can be more or less strict depending on deployment considerations.

URL white lists should always include the scheme of the URL. Use of https is RECOMMENDED to mitigate the risk of man-in-the-middle attacks on browsers being redirected to callback URLs.

One possible strategy is to white list entire host names or domains, e.g. a white list of “http://*.example.com” or “http://www.example.com”. This strategy provides security only if every web page in the entire host name or domain is equally secure and is equally trusted.

Another strategy is to white list scheme, host, and path or subpath, but allow arbitrary query parameters.

Fixing the entire URL, including query parameters, is also possible. However, Authorization Servers MUST preserve the oauth_state parameter to allow Clients to preserve state through the redirect to the Authorization Server and back.

Note: not sure how to discuss use of the URL fragment. It protects against some attacks, but not against others. It makes implementation more complex. It is faster for some use cases, and slower for others.

Client Authentication

Authorization Servers MAY authenticate callback URLs by binding verification codes and callback URLs to client authentication. This works as follows:

The Client passes the client id to the Authorization Server when requesting approval.

When the Authorization Server creates the verification code to the specified client id and callback URL.

When the Client requests an access token, it submits the verification code, the callback URL, the client id, and the client secret to the Authorization Server. The Authorization Server determines which client and callback URL are associated with the callback URL, and then makes the following security checks:

- the client secret MUST match the client identifier
 If the client secret does not match, the wrong client is attempting to redeem the verification code.

- the client identifier on the access token request MUST match the client identifier from the authorization request

If the client identifier does not match, the wrong client is attempting to redeem the verification code.

- the callback URL on the access token request MUST match the callback URL from the authorization request
If the callback URL does not match, the verification code was sent to an illegitimate callback URL and may have leaked.

Note: draft-hardt-oauth-01 has the AS checking the callback URL here. This seems wrong - is there any reason for the AS not to check the callback URL before minting the verification code?

Note that other types of client authentication (besides the client secret) work equally well for this.

Generating Refresh Tokens and Access Tokens

... same as for Username and Password Delegation Profile.

Refreshing Access Tokens

... same as for Username and Password Delegation Profile.

Recovering from Compromise

... same as for Username and Password Delegation Profile.

Note: maybe add notes about “remember me” settings, as per OAuth 1.0a security considerations?

Automatic Processing of Authorizations

Authorization Servers MAY implement automatic processing of authorizations based on either the client secret, the callback URL, or both.

Rich App Profile - Security Considerations

Note: it looks like draft-recordon-oauth2 combined the Web Client Flow with the Rich App Profile? I’m going to write the considerations for the draft-hardt-oauth-01 rich client profile. I’m concerned about combining the two flows because Authorization Servers can use callback URL whitelisting for Web Client flows, but not for the Rich App Profile. Also concerned that we lost all of the logic that rich clients can use to pick up verification codes and callback URLs. Doing this from installed applications is really different from doing it from javascript.

Note: it looks like all three of WRAP/draft-recordon/draft-hardt omit the callback URL from the access token request. Not sure that’s a good idea.

The Rich App Profile is intended for use by client applications that can open a web browser to request user approval. Note that these client applications cannot authenticate themselves to Authorization Servers. Callback URL white listing is ineffective because the client application is not actually part of a web site and the browser same-origin policy cannot be used. Client secrets are ineffective because the client applications are subject to reverse engineering to extract the secrets.

All security here comes from the user consent step and the distribution of the verification code. If the verification code leaks to an unauthorized client, the user's data will leak as well.

Generating Refresh Tokens and Access Tokens

After the Authorization Server has verified the Client and User identities, the Authorization Server returns a Refresh Token and Access Token to the client.

Authorization Servers SHOULD provide Users with the ability to revoke access granted to the Client.

Authorization Servers SHOULD use cryptographic hash functions to store one-way hashes of refresh tokens instead of plaintext.

Refresh tokens SHOULD contain sufficient entropy that they can withstand off-line brute-force attacks on hashed tokens.

Note: refresh tokens probably merit separate security considerations of their own.

Refreshing Access Tokens

Access tokens are assumed to expire. Clients need to present their Refresh Token in order to obtain new access tokens.

Authorization Servers do not need to verify any client information before issuing new Access Tokens. The refresh token is sufficient.

Recovering from Compromise

An attacker who gains access to an Access Token gains temporary access to user data, until the access token expires or is revoked.

An attacker who gains access to a Refresh Token gains permanent access to user data. The client secret cannot be changed, so the compromised refresh token MUST be revoked.

Automatic Processing of Authorizations

Rich client applications cannot be reliably authenticated, so Authorization Servers MUST NOT implement automatic processing of authorizations.

Device Profile - Security Considerations

Damian Menscher assisted with this analysis.

The Device Flow is suitable when the client is a device which does not have an easy data-entry method (e.g. game consoles or entertainment centers), but where the end-user has access to a separate computer with simple data-entry methods (e.g. their home computer, a laptop or a smartphone).

Note that because this flow does include a code passed by the Authorization Server to the device, the protocol is intrinsically vulnerable to the same session fixation attack as OAuth 1.0. This session fixation attack would work as follows:
- attacker retrieves a user code and device code from the Authorization Server
- attacker asks user to approve the user code
- user approves the user code
- attacker uses the user and device code to retrieve the user's data

An alternate attack is for the attacker to attempt to link their own account to another user's device code. The account linking attack would work as follows:
- attacker uses their own account to login at the service provider
- attacker repeatedly guesses user codes
- if the attacker guesses a valid user code, another user's device is now linked to the attacker's account.

The impact of this attack varies depending on the use case. This attack requires less social engineering than the session fixation attack.

This flow is intended for use cases where better security mechanisms are not possible.

Generating Device Codes

User codes must be short enough that users can copy them from the device to another computer without making typographical errors.

Users codes must last long enough that users can successfully enter the device code into the Authorization Server.

Unused user codes must expire quickly enough that attackers cannot appreciably reduce the number of available user codes by requesting too many.

User codes must contain sufficient entropy that an attacker cannot brute-force a user code before it expires.

User code entropy can be increased either by making the code longer, or by increasing the variety of characters used in the code. For example, a code consisting of the digits 0-9 contains roughly 3.2 bits of entropy per character. A code consisting of lower-case letters and digits but omitting characters that are easily confused (i, j, l, m, n, o, v, w, 0, 1) contains roughly 4.7 bits of entropy per character. Mixing upper- and lower-case can also increase entropy.

The security of user codes depends on the following factors:

devices_per_second: the number of legitimate devices requesting approval each second.

average_approval_seconds: the average amount of time it takes for a legitimate user to approve a device.

max_approval_seconds: that maximum amount of time a device code can be outstanding before it expires and users can no longer approve the code.

all_possible_codes: the total number of codes an Authorization Server could possibly generate.

guesses_per_second: the number of times per second an attacker can attempt to guess a code.

attacker_seconds: the length of time an attacker will persist in the attack.

The probability that an attacker has successfully guessed a user code is given by the following formula:

let attacker_guesses = guesses_per_second * attacker_seconds

let attacker_reserved_codes = guesses_per_second * max_approval_seconds
    This is the number of codes that the attacker has guessed previously and knows are no longer valid.

let available_codes = all_possible_codes - attacker_reserved_codes

let num_valid_codes = devices_per_second * average_approval_seconds

let P(attacker fail) = (1 - num_valid_codes / available_codes) ^ attacker_guesses

let P(attacker succeeds) = 1 - P(attacker fails)

As an example, consider an Authorization Server that uses 8 character user codes consisting only of decimal digits. Assume that there 100 users approving devices every second, and that they take on average five minutes per device. Assume that users are allowed to take a maximum of ten minutes to approve a device. Assume an attacker using a botnet that can send 100 requests per second for one minute.

devices_per_second = 100 users * 300 seconds per user = 30,000 guesses_per_second = 100

attacker_reserved_codes = 100 * 600 seconds = 60,000

available_codes = 8 character codes with ten digits per character - 60,000= 10^8 - 60,000

attacker_seconds = 60 seconds

attacker_guesses = 6000

After one minute, the attacker has an 83% chance of having guessed a valid device code.

If the user code is changed to 12 decimal characters, the probability of attacker success after only 6000 attempts drops to .02%. The attacker needs to send tens of millions of requests instead.

If the user code is changed to consist of 12 characters from 0-9a-z, but omitting easily confused characters like i, j, l, m, n, o, v, w, 0, 1, the probability of attacker success becomes negligible.

Service providers SHOULD rate limit attempts to redeem user codes.

Other Security Considerations

... security considerations are otherwise almost identical to the rich client profile ...

Device codes should be high-entropy.

Under normal circumstances, a user will only enter a user code at the Authorization Server once. If an Authorization Server observes multiple users entering the same user code, it is possible an attacker has guessed the user code of a legitimate user. The Authorization Server SHOULD revoke all access tokens and refresh tokens minted based on the verification code.