OAuthSessionFixationAttack

Background

OAuth is vulnerable to a session fixation attack. This document describes how to fix the OAuth consumer code in Shindig to be compatible with fixes to the OAuth protocol. (Note that this began as a description of the design, then was updated to reflect the implementation.)

Two-legged OAuth is not impacted, so there are no changes there.

The OAuth service provider code in Shindig needs changes as well, but they are relatively minor compared to the consumer-side changes and will not be discussed in this document.

Assumptions

Assumption: http://oauth.googlecode.com/svn/spec/core/1.0a/oauth-core-1_0a.xml is an accurate description of the new OAuth spec.

Assumption: Many service providers will require preregistration of callback URLs for consumer keys, sometimes with proof of domain ownership. In order to be compatible with this requirement, it is important that the callback URLs used by opensocial applications/Shindig be predictable. Varying query parameters is definitely acceptable. Varying path components might be OK. Varying hostname is definitely not acceptable.

Assumption: OAuth request tokens for service providers will continue to be long lived (tens of minutes or longer). Many SPs have reduced their request token TTL to a few minutes because of the session fixation attack, so this may not be a safe assumption. If request token TTLs are too short, we’re going to run into problems where a gadget embeds the approval URL in a page, and then the token expires before the user approves access. If that happens, we’ll need to revisit how we kick off the approval process so that we fetch a request token when a user opens the popup window, instead of before hand.

Assumption: we need to maintain backwards compatibility with existing gadgets and OAuth service providers.

Detailed Design

Gadget XML Changes

Today, gadgets may or may not include an oauth_callback parameter on the approval URL they include in their gadget spec. Here’s an example of a declarative definition:

```
```

Even if the author doesn’t include the callback URL in their spec, they may still add an oauth_callback parameter in javascript at runtime:

```
approvalUrl = approvalUrl + "&oauth_callback=http://mycustomcallback.com";
```

That flexibility is no longer possible with the new OAuth spec, because the callback URL is now a critical part of the security of the protocol.

If an OAuth service provider supports the new version of the protocol, gadgets that specify an oauth_callback URL in their spec will find that the parameter is ignored. Instead, the oauth proxy will insert a callback URL. Service providers may associate callback URLs with particular gadget consumer keys. For example, if a service provider has a special consumer key for gadget X, they might allow the owner of gadget X to specify a custom callback URL for use with that key.

Gadgets that specify an oauth_callback URL at runtime may break when used with new service providers. The service providers will notice the oauth_callback URL appear on both the request token request and the approval request. At best the service provider will silently drop the version from the approval request. Some service providers may reject the request entirely instead. (So far no SPs have done this, they all use the oauth_callback parameter from the request token step. Good!)

Initialization

Gadgets will trigger three-legged OAuth exactly as they do today, by requesting data from a service provider with authz=oauth or oauth_use_token=always.

Request token request
Shindig will send a request token request to the service provider and include the additional parameter: oauth_callback=<callback URL>. For example

```
```

The callback URL will be generated as follows:

- **Protocol:** same as protocol used for makeRequest.
- **Host:** hardcoded to match value from configuration. Example: oauth.gmodules.com
- **Path:** hardcoded to match value from configuration. Path to special oauth callback servlet. Example: /gadgets/oauthcallback
- **Query parameters:**
  - `cs`: encrypted and signed callback state parameter.

The `cs` parameter will be wrapped up using the standard Shindig BlobCrypter interface. It doesn't actually need to be encrypted (signing would be sufficient), but the encryption doesn't hurt anything and may be useful in the future. The blob will contain the following name/value pairs:

- **u:** full URL to the oauthcallback servlet hosted on the gadget domain. This will be verified both via Host header and by a locked-domain check based on the gadget security token.
- **ts:** timestamp. Blob will expire after one hour. (A few minutes would probably be sufficient).

The locked-domain check on the hostname is necessary to prevent the callback token from being leaked to a malicious gadget. Otherwise the following attack is possible:

1. Attacker installs good gadget.
2. Attacker steals security token from good gadget, sends to makeRequest on a locked-domain that belongs to their own malicious gadget.
3. Attacker gets back approval URL and request token associated with attacker's id.
4. Attacker gets victim to click on approval URL.
5. Victim is asked "Do you want gadget <good-gadget> to access your data?"
6. Victim says yes.
7. Victim is redirected to callback servlet, which then forwards callback token to malicious gadget.

**Request token response**

If the service provider supports the new OAuth protocol, they will return an additional parameter with their request token response: oauth_callback_confirmed=true. The proxy could use that parameter to detect whether the SP supports the new protocol or not. (So far it hasn't been necessary.)

The approval URL specified in the gadget will be updated to include the oauth token, and then returned to the gadget.

**Authorization Request**

The standard oauthApprovalUrl parameter will be returned to the gadget. The gadget will embed this URL in a link in the page using the oauthpopup library. When the user clicks the link the approval window will be opened.

The service provider will prompt the user to confirm access, and when done will redirect back to the oauth_callback.

**Callback Request**

The callback request will arrive at the callback servlet, looking something like this:

```
```

The callback servlet will decrypt and verify the "cs" parameter to identify which gadget domain originated the approval request. The callback servlet will then redirect to <gadget-domain>/gadgets/oauthcallback, minus the "cs" parameter.

Today the OAuth callback servlet is very simple, it just calls window.close(). That will be updated a bit to something like this:

```
try {
    window.opener.oauthReceivedCallbackUrl_ = document.location.href;
} catch (e) {
}
window.close();
```

The oauthpopup library monitors for when the window closes. When the window closes, the gadgets.oauth.Popup.handleApproval_ method is invoked.

**Gadget repeats request for user data.**
The oauthpopup library notices when the window closes and invokes gadgets.oauth.Popup.handleApproval_. That in turn invokes the closeCallback_, which will then call makeRequest again.

makeRequest will check for window.oauthReceivedCallbackUrl_. If that value is present, it will be copied into the parameters sent to the gadget server as OAUTH_RECEIVED_CALLBACK. Callback tokens are single use, so the cached callback URL will be cleared as soon as it is used. (window.oauthReceivedCallbackUrl_ = undefined.)

Example:

```js
  closeCallback_();
  fetchData();
  gadgets.io.makeRequest(...);
  XMLHttpRequest(... with OAUTH_RECEIVED_CALLBACK=window.oauthReceivedCallbackUrl_ ...)
```

**Server-side handling of repeat request for user data**

Server-side code in OAuthRequest will check for the OAUTH_CALLBACK_RESPONSE parameter on the request. It will parse oauth_verifier out of the URL, and pass the oauth_verifier on the access token request.

The service provider will verify the request token and callback token, and then return an access token and token secret.

**Further data requests.**

Further data requests work as they normally do.