Secure Application Development on Facebook
March 2010

This document provides a basic outline/best practice for developing secure applications on the Facebook platform. Facebook applications are web, desktop, or mobile applications that make use of the Facebook API to integrate tightly with the social network experience.

This paper will cover the following topics

- Overview of the Facebook Platform
- Security Goals
- Important Rules to Follow
- Common Vulnerabilities and Protections

This document is designed for the Facebook developer, but it can also be used as a reference for non-technical readers. Depending on the reader's level of technical understanding of security vulnerability classes and the Facebook platform, sections of the document may be skimmed or skipped.

Overview of the Facebook Platform

The Facebook application developer has many choices in how they integrate into the Facebook platform with their application. It is important to understand the basic elements of the platform before diving into the details of Facebook application security, as each method of integration has different security properties. If you are already familiar with the Facebook platform components, feel free to skip this section.

There are two main categories of Facebook applications: Platform applications and Facebook Connect applications. Both types of applications can use Facebook markup and scripting languages, as well as a REST client to access the Facebook API. Facebook Connect applications communicate with Facebook through crossdomain communication channels. Platform applications communicate directly with the Facebook servers. The following terms will be referenced when discussing Facebook applications:

- **Application canvas** – The application canvas is the page on Facebook servers where an application lives. Application canvas pages are accessed through the *apps.facebook.com* domain. For example, the application canvas URL for a fictional game called "Goatworld" might look like this: http://apps.facebook.com/goatworldgame/ . The application canvas page will either be Facebook markup language or an external site hosted within an IFRAME.

- **Canvas callback URL** – The canvas callback URL is the file or directory on the developer's application servers where the application files are hosted. Facebook proxies the content from the canvas callback URL to the application canvas page.

- **Post-Authorize callback URL** – The callback URL is a page on the developer's servers which is pinged by Facebook each time a user authorizes the application.

Platform applications

Platform applications run in a sandbox and are accessed through the application canvas page. There are two types of platform applications that use different methods for sandboxing, including FBML and IFRAME.

- **FBML –** Applications written using the Facebook markup and scripting languages instead of the traditional HTML and JavaScript. When a user accesses the application canvas page, the Facebook proxy pulls down the FBML from the application servers and translates it into HTML before rendering in the user's browser. It follows that the application code runs in the *apps.facebook.com* domain. These applications can access Facebook user data directly using FBML, but may also make calls to the Facebook REST API servers.

- **IFRAME –** Applications that are written using traditional web development languages such as HTML, JavaScript, CSS, and run on the developer's application servers in an IFRAME hosted in the Facebook application canvas page. These applications cannot use FBML directly, so they tend to rely on components from Facebook Connect, such as XFBML and the JavaScript client library, as well as the Facebook REST API.

Facebook Connect Applications

Facebook Connect applications do not run directly on the Facebook platform, but can access a set of powerful APIs to integrate closely with Facebook. Facebook Connect applications can be web, mobile or desktop applications. Connect applications use XFBML tags (which are similar to FBML) as well as the JavaScript client library, the Facebook PHP client or a Facebook REST client in any language.

Security Goals

The first step in securing any application is to outline the required security guarantees your application needs to make, and then to select mechanisms that support those guarantees. For a Facebook application, the security challenges are slightly different than a regular application. The following is a non-exhaustive list of important items that a Facebook application developer has to keep in mind when thinking about security:

- **Protect access to the Facebook user account.** When a user authorizes your Facebook application, they have permitted your application to obtain access to their Facebook account. The application is granted read access to profile data, friend information, and photo albums. The application can also be granted write access to the user profile, status updates, notifications, and the ability to send emails to the user, when the user grants the application the privilege to access restricted Facebook APIs. The application must be as resilient as possible against attacks which could compromise the application's access to a user's Facebook account through the trust the user has granted the application.

- **Respect the Facebook privacy policy.** In order to respect the user's trust in your application, as well as to keep your application from being removed by Facebook, it is very important to adhere to Facebook's privacy guidelines. Facebook denotes what user data is allowed to be stored by your application and for how long. When you do store Facebook account data, use sufficient access controls to guard this data from anyone but the legitimate, authenticated user. More information about privacy and storable user data can be found here: http://wiki.developers.facebook.com/index.php/Understanding_User_Data_and_Privacy http://wiki.developers.facebook.com/index.php/Storable_Data

- **Protect application-specific user account data and state.** Make sure you understand the differences between how Facebook applications and traditional applications approach authentication, authorization, and access controls. The application relies on

Facebook to tell whether a user has been authenticated, as well as other information about the user's session. It is your responsibility to ensure that this information has not been forged or tampered with, before calling Facebook APIs or returning Facebook data for the user.

- **Protect against common web application vulnerabilities that put application specific data and servers at risk**. The Facebook sandboxing of platform applications can sometimes be confused as a protection mechanism for the application. However, Facebook sandboxing only provides protection to Facebook servers. Application servers can be attacked directly, and the minor protection mechanisms the platform sandboxing provides can sometimes be bypassed. It is therefore just as crucial to guard your Facebook application against the same common vulnerabilities you would in a traditional application.

- **Maintain confidentiality and integrity guarantees when integrating Facebook with HTTPS sites**. When integrating an HTTPS site with the Facebook platform it is important to maintain the confidentiality and integrity guarantees offered by the protocol, in spite of the fact that Facebook does not serve its own content over SSL. The application canvas page is not served over SSL, which implies that it is impossible to provide support for SSL communication in an FBML application. Facebook Connect does provide support for SSL. This will be important to take into account when deciding on your method of integration with the Facebook platform.

Important Rules to Follow

Before delving into the specifics, consider the following list of important rules for secure development on the Facebook platform. The remainder of this document will provide further detail on the vulnerabilities and protections referred to in each of these rules.

- **Your Facebook Application secret must be kept secret**! The Facebook Application secret is given to you upon registering your application, and it is used to authenticate requests to and from the application. If the secret is revealed to an attacker then they can forge requests to and from the application. Store the secret in a secure place (i.e. not hardcoded in the source or in source control repositories, such as SVN) and make it available to as few people as possible, such as net ops and dev leads.

- **Do not hardcode FB application secret or any other secrets in SWFs, JavaScript, or any other client side code such as iPhone and Desktop applications**. Attackers can easily decompile SWFs to discover their secrets. The *strings(1)* program can be used to identify secrets in other client binaries. Instead, use the session secret to call many sensitive APIs, or a server side relay proxy to call APIs which require the application secret.

- **Check the Facebook signatures.** The Facebook signatures that are sent with requests should always be checked in order to verify that the request has not been tampered with and was sent legitimately by Facebook. The signature is created by an MD5 hash of the request parameters concatenated with the application secret, which should only be known to the application developer and Facebook. This signature can in some cases also be checked to provide protection from Cross-Site Request Forgery (CSRF) attacks when it is sent in the GET or POST parameters.

- **Provide additional CSRF protection mechanisms, especially with Facebook Connect sites**. It is recommended to implement a separate CSRF protection mechanism that is designed specifically to mitigate this vulnerability class. In Facebook Connect sites, the signature and parameters are sent as cookie values rather than as URL parameters. Therefore, these sites need additional CSRF protection as the signature will be automatically sent with the request, even if the request was forged by an attacker.

This applies to platform applications which make use of features of Facebook Connect. In this case, the signatures will sometimes be sent as request parameters, and sometimes as cookie values.

- **Validate and perform output encoding of user controlled data (as usual) and beware of XSS vulnerabilities in FBML applications**. In FBML applications, Facebook will translate any JavaScript to FBJS, which makes traditional XSS attacks that use JavaScript difficult (Facebook would hope this is impossible). However, the attacker can still inject FBML. If the attacker is able to inject into the application canvas, they can use the FB:SWF tag to inject a SWF from arbitrary domains. Facebook automatically passes any SWFs loaded in the application canvas all user specific secrets (such as the session key and session secret which must be passed with more sensitive API calls). This has an equivalent effect to XSS attacks on traditional applications that steal the user session cookies. Other types of FBML tags can also be injected, which allow an attacker to abuse the user's trust in allowing the application to have access to their Facebook account, or maliciously redirect them to other sites.

- **Configure restrictive crossdomain.xml policy file**. The crossdomain.xml policy file must be restrictive in order to keep attackers from changing the *flashvars* passed to the SWF, potentially subverting verification of the Facebook parameters passed. The crossdomain.xml policy file should be as restrictive as possible to allow access only from the trusted domains that are necessary.

- **Lock down administrative functionality**. Keep administrative functionality on an entirely separate domain, preferably available only internally. A common flaw in Facebook applications is that they allow access to the administrative functionality based on a white list of hardcoded Facebook IDs, and is accessible from the same domain as the application. This allows for XSS attacks to have an even more devastating effect and could make it easier to discover the structure of requests for CSRF attacks. Also, because Facebook does not use HTTPS, sessions of administrators may be hijacked and used to access the administrative functionality. Require additional authentication mechanisms for administrative functionality.

Common Vulnerabilities and Protections

The following section will cover a few common web application vulnerabilities where specific care needs to be taken in order to protect Facebook applications. The overview will include the following items:

- Authentication and Authorization
- Signature Verification
- Signature Generation
- Cross Site Request Forgery
- Cross Site Scripting
- Considerations for Flash Applications
- Secure Transport using SSL
- Administrative Functionality

Authentication and Authorization

In order for any Facebook application to make use of the API, the user must first authorize the application and establish a session. After a user authorizes the application, a session is created by Facebook. The session information is then passed by Facebook to the post-authorize URL as request parameters, or it is set in cookies scoped to the application domain using JavaScript. Facebook Connect applications rely only on cross-domain communication between Facebook and the application to process API calls. Both Facebook servers and the application servers must authenticate these communications. Otherwise, attackers could spoof or tamper with requests from either party.

Important elements of the Authentication and Authorization

- The **API key** – The API key is given to you when you setup your Facebook application. Facebook uses this unique key to identify your application. The key is public, and will be sent in the clear with requests sent by Facebook to your application, and can be included in client side code such as JavaScript and Flash.

- The **application secret** – The application secret is also distributed during application setup. This secret is used to authenticate the communication between Facebook and your application. The secret is private to your application and must never be revealed.

- The **session key** – The session key is the key to a temporary session which expires after an hour or when the user logs out of Facebook. The session key is set by Facebook when a user authorizes the application. The session key is used to call API methods that require an active session. For example, and calls to *friends.get* require a session key to determine which user to retrieve the friends for.

- The **session secret -** The session secret is a user-specific secret that can be used in place of the application secret when generating a signature for making sensitive API calls. The session secret is unique per user and short lived, and is thus much less sensitive than the application secret. The session secret should **always** be used instead of the application secret in any client side code like JavaScript, Flash, mobile or desktop applications. This keeps the application secret from being revealed either directly or through reverse engineering of client side binaries. The list of APIs which can be called using the session secret can be found here: http://wiki.developers.facebook.com/index.php/Session_Secret_and_API_Methods

- The **Facebook parameters** – Facebook passes parameters to the application with any requests that include the API key and the session key, as well as other values which provide information about the user and session. Facebook sends the parameters as part of GET requests for IFRAME applications and as part of the POST body for FBML applications. Different parameters will be set depending on the type of Facebook application and what the user is doing with the application.

- The **Facebook signature** – One value included in the Facebook parameters is the signature, which is an MD5 hash of all the parameters concatenated with the application secret. The application uses this value to verify that the parameters were not tampered with and were really sent by Facebook. Conversely, the signature allows for Facebook to verify that API requests have not been tampered with and were legitimately sent by the application identified by the given API key.

Authenticating Communications from Facebook

Authenticate any information received from Facebook. Authenticating information will ensure it was legitimately sent by Facebook and not spoofed or tampered with. When Facebook passes information about the user and session to your application in the Facebook request parameters or cookies, it may be tempting to use that information directly from the request. However, attackers can control this data to forge or tamper with all of the Facebook parameters sent in the request, excluding the signature. The signature cannot be forged, as it is generated with the application secret key which the attacker should not know. Before using the values of any of the parameters received from Facebook, re-calculate the expected signature and verify that it matches the signature sent with the request. In the event of a mismatch, reject the request entirely. Some of the functions in the PHP client do this automatically. For example, the function *get_valid_fb_params* from the Facebook PHP library will return the validated parameters.

Verifying and Generating Facebook Signatures

The chosen client library should have built in functions to verify and generate the signature, but it can also be preformed manually.

Detailed information on the signature verification process can be found here:
http://wiki.developers.facebook.com/index.php/Verifying_The_Signature.

Signature Verification

Signature verification using the application secret must **never** be done on the client side, as this would expose the secret. Always perform signature verification using server code.

The signature verification process is different depending on the type of Facebook application:

- **Platform applications** – In platform applications, Facebook will send the signature and other parameters as part of the GET or POST request. The application should grab the Facebook parameters from the request parameters, and recalculate the expected signature value using the signature generation algorithm, or the built in function from the Facebook client in use.

- **Facebook Connect Applications** – For Facebook Connect applications the Facebook parameters are sent as cookies. To verify these parameters, the application must extract the values from the cookies and perform the verification using the signature algorithm. The Facebook PHP client supports cookie based signature verification and is well-tested. The verification method must be checked in any unofficial libraries you may decide to use. Facebook Connect applications do not need to verify the signature very often. The most common scenario which requires verification of the Facebook cookies is when the session is being transferred from the client side to use server side libraries. When this occurs, the application must verify the cookies sent with the session transfer request. Otherwise, a malicious user could tamper with their cookies in order to trick the server into believing they were another user. More information on server side integration with Facebook Connect applications can be found here: http://wiki.developers.facebook.com/index.php/Using_Facebook_Connect_with_Server-Side_Libraries .

Signature Generation

Signatures must be generated in order to call sensitive APIs and in order to re-calculate the signature as part of the verification process. The Facebook PHP library includes functions for performing signature generation. For server side code, the application should generate the signature using these API functions or by re-implementing the signature generation algorithm.

For client side code such as Flash, JavaScript, mobile and desktop applications, which cannot use the application secret directly, there is an alternate secret used to generate the signature: the session secret. The session secret can be used to call many APIs, but there are some API functions that can only be called with the application secret. In this case, the client side application should create a simple, server-hosted proxy which makes the necessary API calls and relays the result back to the client side application. Depending on the type of application, there are several different options for retrieving the session secret.

- **Desktop and iPhone Applications –** Desktop and iPhone applications should either use a *Session proxy* (http://wiki.developers.facebook.com/index.php/Session_Proxy) to create a session and retrieve the session secret, or embed a Web browser in the application in order to use Facebook Connect to start the session. In order to obtain the user session in the traditional manner, an iPhone or Desktop application would have to call *auth.getSession*. However, calls to *auth.getSession* must be signed with the application secret. The application secret must **never** be hardcoded in the source of an iPhone application. Attackers can easily decompile Objective C and thus the secret can be discovered by anyone who downloads the application. Therefore, this API method is not safe for direct use in iPhone applications. The same is true for Desktop applications using Adobe Air or .NET.

The session proxy is simply a callback to an application server which then makes the *auth.getSession* call on the server and returns the resulting session variables back to the iPhone application. Ensure that the call to *auth.getSession* is made at an HTTPS endpoint, as this is required to protect the session secret from being revealed to network attackers when it is sent back to the application.

The other option is use the Facebook Connect API via a Web browser embedded in the application. In this case, the application developer must direct the embedded browser to Facebook login pages with specific URL parameters which direct Facebook to return the session information. One of these parameters contains the URL to which the user will be redirected after they successfully login. The session secret can then be retrieved from this URL and passed back to the application. The session secret can then be safely stored on the user's system until they close the application. More detailed information can be found at http://wiki.developers.facebook.com/index.php/Authorization_and_Authentication_for_Desktop_Applications

- **Facebook Connect Websites –** When making API calls using the JavaScript client library, Facebook Connect web applications should generate the signature for API calls using the session secret, in much the same way that desktop applications do. The session secret will be passed to the application callback URL after the user has authorized the application. More detailed information on this process can be found here: http://wiki.developers.facebook.com/index.php/Authorization_and_Authentication_for_Desktop_Applications

- **Flash** – Flash applications will be passed the Facebook parameters as flashvars from their hosting page. The hosting page must validate the signature on the server side before passing it to the SWF object, as the application secret cannot be safely stored in the SWF. This is due to the fact that SWFs can be downloaded by an attacker and easily decompiled in order to retrieve any secrets used in the ActionScript. The ActionScript API can then use the session secret passed in these variables to generate signatures for further API calls.

Cross Site Request Forgery (CSRF)

CSRF vulnerabilities exist when an application sends predictably structured requests which update server side state. When such a request contains no unpredictable parameters, an attacker can create a forged version of the request and send it on behalf of arbitrary users, without their knowledge. . CSRF attacks and defense are described in detail in the following CSRF whitepaper: https://www.isecpartners.com/files/CSRF_Paper.pdf.

This vulnerability class can be difficult to understand. One common example of a CSRF vulnerability and protection mechanism that we are all used to is the password update feature of many web applications. It is a common requirement that the user enter their current password as well as their new password in a password update form. This requirement is in fact a CSRF protection mechanism (as well as adherence to other design principles). Consider the case where the current password is not required when performing the update. In this case, an attacker could forge the request which updates the user password. If the attacker can get the victim to load the forged request in a browser with an active session on the site, the browser will send along the victim's session cookies with the request. This can be done by placing the forged request as the source of an IMG tag or an invisible IFRAME in a place where users of the application are likely to browse to, or as link disguised with a tinyURL service in a Facebook status message or comment box. When the server receives the request, it will have no way of distinguishing the forged request from a request sent purposefully by the user. The server will thus perform the password update action and update the user account with the attacker supplied password. The victim will then be locked out of their account. Fortunately, most applications require the current password to be sent with password update requests. This way, if an attacker attempts to send a forged request on behalf of their victim, the request will fail because the attacker will not have the victim's current password.

While the user password can act as a CSRF protection mechanism, it would be unreasonable to require the user to enter their password each time they make a request which updates server side state. Fortunately, there are other ways to provide protection from this attack.

General protection from CSRF attacks (recommended solution)

To provide protection against request forgery, the application should include and require a CSRF prevention token in the request that is both unpredictable and uniquely tied to the user session and action. This token will be sent with every request which updates server side state. When the server receives a request, it can check for the appropriate token value. If the token is incorrect or does not exist, the request will fail. One way such a token can be generated is through the following algorithm: *HMAC_sha1(action_name + secret, session_id)*. More CSRF token generation techniques can be found in the whitepaper referenced at the beginning of this section. This token can be sent in a hidden form field with each request. The server can then perform the calculation upon receiving the request and verify that the token sent with the request matches. Because the secret exists only on the server side, the attacker will have no way of formulating the token for use in a forged request. This is the recommended protection mechanism for CSRF attacks. However, in some cases the Facebook signatures can also provide protection from this attack.

Protecting against CSRF attacks using the Facebook Signatures

The Facebook signatures can provide protection from CSRF attacks, but they cannot be relied upon in all circumstances. The Facebook signatures are computed in a similar way to the recommended CSRF protection token algorithm; the signature is a cryptographic hash of the request parameters and a server side secret. Note that the use of the *HMAC_sha1* hashing algorithm is significantly stronger than the MD5 hashing algorithm used by Facebook, which has been proven to be vulnerable to some attacks[1]. While the signature is an adequate CSRF protection token, there are a couple very important caveats to keep in mind if the Facebook signature is used as a CSRF protection mechanism:

- **Facebook signatures cannot protect against CSRF attacks when sent as cookie values.** CSRF protection tokens provide no protection when sent as cookie values.

[1] http://www.win.tue.nl/hashclash/rogue-ca/

The browser will automatically send cookie values with the request, thus the attacker does not need to guess the value, and the structure of the request remains predictable. Facebook Connect applications send the signatures as cookie values, rather than request parameters. It is important to remember that these applications will have to provide a separate CSRF protection mechanism. This caveat highlights one of the reasons to implement a separate CSRF protection system that is designed specifically for this class of vulnerabilities. This way, code can be ported to different types of Facebook (and non-Facebook) applications and will still be protected from this class of attack.

- **The Facebook signature is not sent with all requests.** The Facebook signature is not always available for use as a CSRF protection token. If this is your chosen method to provide CSRF protection, you must analyze your application you must analyze your application for areas where the server side state is being updated, but the Facebook signature is not sent with the request parameters. These requests will have to be protected through a separate CSRF protection mechanism.

Cross Site Scripting (XSS)

XSS attacks are one of the most common and dangerous web application vulnerabilities. An XSS vulnerability exists when user controlled data is written directly to the page source without sufficient input validation or output encoding applied. A common example of this vulnerability is a web page which takes values from request parameters and includes them directly in HTML or JavaScript without validation or sanitization. Consider an example PHP page from an application which displays a username passed in via GET request parameters. The source of this page contains the following code:

```
<?php
echo(“Your username is:”.$_GET[“username”]);
?>
```

Now imagine that a malicious user tampers with the request parameters and enters the following string in the `username` field:

```
<script>alert(document.cookie);</script>
```

The source of the resulting page will then contain:

```
Your username is:
<script>alert(document.cookie);</script>
```

When the browser renders this page, the script tag that was entered as the username will be written to the source of the page. The browser will parse it as HTML and the script will be executed. The malicious user can then exploit this vulnerability against other users by creating a request such as the following:

http://example.com/displayUserName.php?username=<script>alert(document.cookie);</script>

The attacker must then convince their victims to load the malicious request in a browser with an active session on the vulnerable site. The attacker may do this by disguising the link with a tinyURL service and putting it in a place logged in users are likely to visit, such as a forum, or by posting it on a Facebook page. When victims load the malicious request in their browser, the attacker injected script will execute and the user's session cookie will be revealed to the attacker (in a real world scenario the script would contain something much more malicious than an alert box). For example, the SAMY MySpace worm was propagated by exploiting an XSS vulnerability[2]. The attacker could also use an XSS vulnerability to rewrite the source of the page so that it becomes a convincing phishing page, or a page which prompts users to install malware. Even cautious users may be tricked, as the page originates from a domain which they trust. This attack can be used in many different ways to compromise the user's browser and session, and is also usually very easy to exploit.

So how should the developer have prevented this vulnerability? The developer should have validated and sanitized the value before writing it to the source of the page. The corrected code would look as follows:

```
<?php
if(ctype_alnum($username)) ($_GET[“username”])){
        $username = htmlentities($username);
        echo(“Your username is:”.$username);
} else {
        echo(“The username you entered contains invalid
characters”);
}

?>
```

[2] http://en.wikipedia.org/wiki/Samy_%28XSS%29

The conditional statement use the built in PHP function *ctype_alnum* to verify that the string entered contains only alphanumeric characters. The *htmlentities* function is a built in PHP function which output encodes un-trusted data so that it is safe to use in the HTML context, so "<" becomes "&lt;" and ">" becomes "&gt;". The user controlled data from the GET request is now safe to write to the source of the page.

Prevent XSS attacks by using a combination of input validation and output encoding. Most class libraries include functions for performing output encoding. While output encoding provides strong protection against XSS, it is best to perform data validation before encoding. The validation and encoding should always be performed on the server side, and should be done using a whitelist of known good data. Instead of searching the data for bad characters, check that the string matches the expected format based on the type of input. For example, if the user controlled data is a postal zip code, validate that the data is numeric, rather than searching the data for "<" characters. It is always harder to enumerate the possible bad data, than to enumerate the possible good data. Validating user controlled data before use is simply the correct way to write code, and along with helping to prevent XSS vulnerabilities, will make the application run more smoothly as a whole. Output encoding can then be used for further protection, catching any data that could not be cleaned during input validation.

Cross Site Flashing (XSF)

Flash applications can also be vulnerable to a similar issue to XSS. When your Flash application uses functions which invoke JavaScript or retrieve URLs, you must check the data you pass to these functions. If the data is user controlled, than an attacker can enter malicious JavaScript or pass a malicious URL. Sometimes, the attacker may be able to load their own malicious SWF inside of the application due to such dangerous use of the *loadMovie* function. A common mistake is to pass data from *flashvars* to these dangerous functions. For more information see the following resource:
http://www.adobe.com/devnet/flashplayer/articles/secure_swf_apps.html

XSS in Facebook Applications

XSS attacks in Facebook applications are unusual due to the use of FBML, FBJS, XFBML, and the Facebook proxy.

- **XSS in Facebook iFrame applications.** In an XSS attack, the injection occurs in the domain of your application. If the injection is possible due to a lack of input validation and output encoding in the application code, then the attacker can execute arbitrary HTML and JavaScript; and in some cases XFBML. The injected code can be used to steal any Facebook cookies which are set when using Facebook Connect code. These cookies reveal the user session key.

- **FBML/FBJS applications.** In an XSS attack on FBML applications, the injected data will be written directly to the source of the page in the apps.facebook.com domain. However, as part of the Facebook sandboxing method for FBML applications, any injected script will be translated to FBJS. This means that an attacker cannot send the traditional XSS payload which, for example, uses JavaScript to access the user session. However, the attacker can still easily inject arbitrary FBML tags. When an attacker can inject FBML, they can do things like inject an FB:SWF tag (<fb:swf />), which embeds a SWF in the application canvas page. Whenever a SWF is loaded in the application canvas, Facebook will automatically pass it all of the Facebook parameters, including the session secret, as *flashvars*. The attacker can then send the *flashvars* back to their server and then use this information to access the application and perform actions on behalf of the user.

Considerations for Flash Applications

There are a few things to keep in mind when creating Facebook applications with Flash. Some of the information in this section may overlap with other sections, but is presented from the perspective of Flash applications.

To better understand the issues facing Flash applications, let us consider a fictional Flash based iFrame platform application called “Goatworld”, which is a game where players build teams of goat buddies with their Facebook friends. Users can send goat buddy requests to their Facebook friends, and the more goat buddies the user has, the higher level they obtain in the game.

The application is built using Flash, and runs from a SWF hosted in an iFrame on the application canvas page. The Facebook parameters are automatically passed to the application in the GET request which retrieves the hosting iFrame. The hosting page takes these Facebook parameters and passes them to the Flash application as *flashvars*. The Flash application then uses these parameters to make calls to Facebook using the ActionScript API. When the game loads, the SWF uses the *fb_sig_user* parameter to discover the current user's goat buddies, and then sets the player's level accordingly. Unfortunately, the hosting page does not validate the Facebook parameters before passing them to the Flash application. This allows an attacker to intercept the request to the hosting page, and change the *fb_sig_user* parameter to that of a different Facebook user, who has many goat buddies. Then, when the Flash application makes the call to retrieve the user's goat buddies, it will retrieve the list of the other user's friends. This allows the attacker to discover the Facebook friends of other users, in addition to providing them with a way to cheat in the game. This exemplifies that the Facebook signature must always be verified when trusting the Facebook parameters. However, there are also other ways that an attacker can manipulate the *flashvars* passed to a SWF. This has to do with the crossdomain communication policy configuration.

- **Crossdomain Policy –** By default, the Flash runtime enforces the same origin policy and SWFs are only allowed to connect back to their domain of origin. If the SWF needs to communicate to servers other than its domain of origin, the crossdomain.xml policy file must be in place to grant access. The crossdomain.xml policy will grant communication to that server from SWFs hosted on the domains specified. When a SWF attempts to make the connection, it will check for the crossdomain.xml file in the web root of the domain it is attempting to connect to. It is very important to properly configure this policy file so that it restricts the access it allows.

  A dangerously configured crossdomain.xml is as follows:

  ```
  <cross-domain-policy>
  <allow-access-from domain="*"/>
  </cross-domain-policy>
  ```

  This dangerous configuration allows access from any domain, effectively enabling Flash content on any site to attack your application. This would allow an attacker to host your SWF on their site, and although the SWF's domain of origin is now the attacker's, the SWF can still make calls back to its original domain because of the badly configured crossdomain.xml file in place. This gives the attacker control of the *flashvars* the SWF uses, and would allow the attacker to, for example, provide a different value for the *fb_sig_user* parameter and other Facebook parameters passed to the SWF. This also allows any domain to make crossdomain AJAX calls to your server, read data, and send custom headers with requests. This can lead to a whole host of serious issues.

Only configure a crossdomain.xml file that grants access only the specific, trusted, domains the SWF needs to connect from.

For example, consider a SWF whose domain of origin is sometimes *foo.example.com*, and sometimes *bar.example.com*, but it always needs to call back to code on *example.com*. The policy file hosted at example.com should then look like this:

```
<cross-domain-policy>
<allow-access-from domain="foo.example.com"/>
<allow-access-from domain="bar.example.com"/>
</cross-domain-policy>
```

This will allow the SWF to be hosted at either the *foo* or *bar* sub domains, but will not allow it to be hosted at the attacker's site. Furthermore, it will not allow arbitrary domains access to the server through crossdomain requests.

More detail on crossdomain.xml policy files can be found here: http://www.adobe.com/devnet/articles/crossdomain_policy_file_spec.html .

More information on development using Flash can be found at http://www.adobe.com/devnet/facebook/ .

Secure transport using SSL

The HTTPS protocol provides the ability to transport information securely over the network using public key cryptography. The protocol provides confidentiality and integrity guarantees. When communicating over HTTPS, one can have some confidence on who they are talking to and that the data sent cannot be successfully tampered with. Without using HTTPS there can be no such guarantee. Facebook does not serve its own content over HTTPS. However, it is still possible to leverage the security benefits of an HTTPS site using Facebook Connect. The following guide discusses how to use Facebook Connect with an SSL site: http://wiki.developers.facebook.com/index.php/Facebook_Connect_Via_SSL .

Areas of an application which deal with financial transactions, such as online payments, must use HTTPS to protect their communications. If you are writing a Facebook application that has a payments component, consider the following recommendations for integrating securely with Facebook.

- **Ensure that all sensitive API calls use HTTPS endpoints for client side applications.** When making sensitive API calls, ensure that they are sent over HTTPS by calling them from an HTTPS endpoint. For example, the function *auth.getSession* returns the user's session secret, which can be used to make sensitive API calls on behalf of the user. For client side applications, the session secret is used in place of the application secret, and therefore must be guarded carefully. Calls to *auth.getSession* should be made to an HTTPS endpoint, so that it will be returned to the client application over a secure connection. The same applies for other sensitive server side APIs.

- **Ensure that sessions with the secure site are initiated over HTTPS.** When a Facebook platform application which is served over HTTP needs to direct the user to a secure site (for example to start a financial transaction), it is crucial that the session be initiated over HTTPS. A common mistake is to direct the user to the secure site using HTTP, and then internally redirect to the HTTPS connection after, without changing the session identifier. This is dangerous as it reveals the session identifier in the clear, before redirecting to the HTTPS site. In the best case, the secure site should not be accessible over HTTP at all.

- **Set the *Secure* flag on all sensitive cookies.** Setting the *Secure* flag on cookies tells the browser to only send the cookie over an HTTPS connection. This will keep sensitive cookies from being revealed to network attackers. Not setting the *Secure* flag severely undermines the protections provided by HTTPS, as the secure session can still be compromised.

Administrative Functionality

Because the Facebook platform is different than traditional application platform, securing administrative functionality can be confusing, and some compromises have to be made. Developers of a Facebook application can set certain users to be administrators and moderators of the application. There are certain API functions that can only be called by administrators. These functions provide the ability to perform administrative actions for the Facebook application, such as banning and unbanning users, setting application properties, and retrieving information about the applications usage statistics. It is tempting to include these functions in the same application canvas as the normal user functionality. Unfortunately, this design goes against the principle of least privilege. It is a best practice to keep the administrative functionality as separate as possible from the normal user application. For Facebook applications, the following recommendations should be considered in order to secure administrative functionality.

- **Require HTTPS.** Any administrative pages must be served only over HTTPS in order to protect from local network attackers. This should be followed even if the administrative portion of the application is served only internally, to protect from rogue insider attacks. The application can still use the Facebook API by using Facebook Connect with SSL to authenticate and then transferring the session to the server side in order to make admin API calls which require the application secret. The session transfer process is described in detail here: http://wiki.developers.facebook.com/index.php/Using_Facebook_Connect_with_Server-Side_Libraries.

- **Host administrative functionality on a different domain than the normal application.** It can be tempting to host the administrative portion of the application on the same domain as the rest of the application. One common scenario is to check if the Facebook ID sent with the request is an administrator's ID and then to display the corresponding administrative functionality in the same Facebook canvas page as the rest of the application. This is not recommended as it can allow for XSS attacks to have a more devastating effect, and can make it easier for an attacker to discover the administrative pages in order to perform CSRF attacks. Instead, the administrative functionality should be hosted on an entirely separate domain. The application can still use the Facebook API by using Facebook Connect and transferring the session to the server side API for calls which require the application secret.

- **Make endpoints available internally only.** The application should only be available on the internal network. As a rule, administrative interfaces should never be available from the internet. Internal only hosting will greatly increase the difficulty of attacking the application, as the attacker will first have to gain access to the internal network.

- **Require additional authentication** – The administrative portion of the application can use Facebook Connect to log into the application via Facebook credentials. However, Facebook sessions are not secured over HTTP, so there is a potential for hijacking an administrators session. If an administrator's session is hijacked by an attacker, they may be able to use that session to access the administrative functionality. In order to provide further protection, require administrators to log into the application using separate credentials.

Author: Justine Osborne

Justine Osborne is a Security Consultant/Researcher at iSEC Partners, an information security organization. At iSEC, Justine specializes in application security, focusing on web application penetration testing, code review, and secure coding guidelines. She also performs independent security research, and has presented at security conferences such as Blackhat, Defcon and DeepSec. Her research interests include emerging web application technologies, dynamic vulnerability assessment tools, Rich Internet Applications (RIA), and mobile device security.

iSEC Partners, Inc.

iSEC Partners is a proven full-service security consulting firm, dedicated to making Software Secure. Our focus areas include:

- Mobile Application Security
- Web Application Security
- Client/Server Security
- Continuous Web Application Scanning (Automated/Manual)

Published Books

Notable Presentations

RSACONFERENCE

Black Hat

web2.0 EXPO

InfoWorld

COMPUTERWORLD SNIA STORAGE NETWORKING WORLD

Whitepaper, Tools, Advisories, & SDL Products

- 12 Published Whitepapers
  - Including the first whitepaper on CSRF
- 38 Free Security Tools
  - Application, Infrastructure, Mobile, VoIP, & Storage
- 8 Advisories
  - Including products from Apple, Adobe, and Lenovo
- Free SDL Products
  - SecurityQA Toolbar (Automated Web Application Testing)
  - Code Coach (Secure Code Enforcement and Scanning)
  - Computer Based Training (Java & WebApp Security)