Neat, New, and Ridiculous Flash Hacks

Mike Bailey

Skeptikal.org
1-22-2010

Adobe's Flash Player has recently come under heavy fire for a variety of security vulnerabilities, from buffer overflows and memory corruption issues to null pointer and information disclosure vulnerabilities. While these are all serious issues, they also have one thing in common: they are easy to patch. As long as a user keeps Flash Player up-to-date, he only needs to worry about 0-day attacks which, while serious, are rare in such a widely deployed technology. We won't be looking at those kinds of issues.

In fact, the issues described in this paper are not Flash vulnerabilities at all. Instead, we will address design issues and common errors in Actionscript programming and server configuration which may be leveraged in an attack on a user, web browser, or website. There is no single patch for any of these issues, but that only serves to make them more serious. This research can be compared to a much better understood web security issue: cross-site scripting. While looking for buffer overflows or low-level code execution flaws in a Javascript parser may yield interesting results, if a researcher (or an attacker) disregards XSS as an attack vector, he is severely limiting his options. Security research is all about finding new options in unexpected places.

Some of the attacks described herein were previously discovered and discussed by other researchers—some as far back as 2006. By formalizing the issues and bringing them all together in one place, I hope to provide a broad overview as well as a specific understanding of the possibilities Flash creates for an attacker and the risks that a security administrator must consider when implementing and working with Flash.

The Same Origin Policy

When web applications were first beginning to include interactive and potentially sensitive content, browser and web developers had a problem: how to keep information from leaking between two different websites. The web was designed to be an open forum for exchanging information, but some types of information—user credentials, personal data, and sensitive communications—needed to be kept private. The concept of a Same Origin Policy was designed to prevent websites from accessing such information. No data (such as browser cookies) that belonged to one website should be accessible to another website.

As client-side scripting languages such as Javascript began to push the boundaries of what a web browser could do, it became more important for the browser to strictly enforce the boundaries of websites and web applications. The Same Origin Policy eventually became a core tenet of modern web browser security. At the same time, attackers began to find new and different ways to violate that policy—Cross-Site Scripting and Cross-Site Request Forgery. These have many applications, but the primary purpose of them is in the name: to enable cross-site attacks.

The modern Same Origin Policy dictates that scripts from Site A may not access scripts and cookies, or read content from Site B. This is useful because Site B can have sensitive cookies, anti-CSRF tokens, and user-specific data embedded in pages. Flash's scripting language, Actionscript, is based on ECMAScript, as is Javascript, so it is no surprise that Flash's Same Origin policy is very similar in structure to that of Javascript. There are a few key differences which will be addressed below, but most of this paper will involve ways to circumvent or abuse this policy.

Faulty Crossdomain.xml Policies

The first major difference between Flash and Javascript's origin policies is that Flash allows a web

server administrator to explicitly allow Flash objects from a secondary domain to communicate with resources on his server. By placing an XML file in the root directory of the web server, he can include specific directives that say which servers' Flash objects are allowed to interact with it. Actions that Flash objects may perform include performing HTTP requests, parsing the data returned by those requests, and submitting forms. In short, a Flash object has full access to that server.

In theory, the crossdomain.xml file is sound—it prevents malicious Flash objects from performing scripted attacks on the server while allowing "trusted" systems to communicate with it. In practice, most system administrators implement excessively permissive crossdomain directives. A server with an unrestricted policy (allow *) is essentially open to the world—any Flash object on the internet may communicate with it, and it is no more safe than a web application riddled with XSS vulnerabilities. A server with a wildcard subdomain in its policy (*.example.com) is considerably safer, but as I will demonstrate, there are many attack vectors that may be leveraged against these as well.

In 2006, Jeremiah Grossman found that 6% of the top 100 websites have unrestricted crossdomain policies. He predicted that this risk was likely to grow. In 2008, Jeremiah used a slightly different set of websites, but found that 7% are unrestricted, and 11% have *.domain.com. Again using a different sample, in late 2009, I performed a similar analysis of Alexa's top 1000 websites and found that 13.4% have unrestricted policies, and 37.6% have overly permissive policies such as *.example.com. While a simple problem to fix, this issue is clearly not going away. What's more, it is already being actively exploited by attackers.

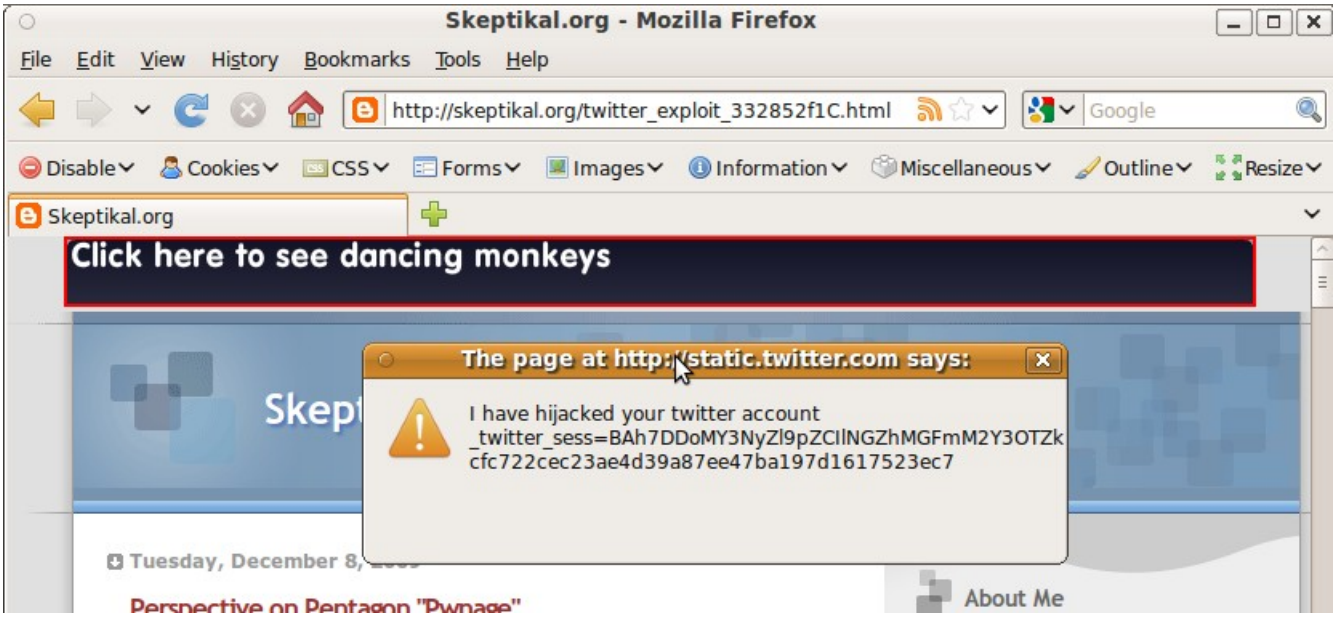
In October 2009, the popular blogging site LiveJournal was affected by a Flash-based worm. All the details of the attack are an interesting case study, but the exploit phase depended on an overly permissive crossdomain policy, which allowed scripts on one domain to forge requests to livejournal.com, extract personal information from a user's profile, send it to one of three central servers, and propogate the Flash object to that user's blog. Any of that blog's readers would, in turn, be infected. This is classic web worm behavior, but instead of leveraging XSS or browser exploits, it leveraged Flash and a flawed crossdomain.xml file.

It should be noted that as long as a server does not have exploitable or interactive content, a permissive crossdomain.xml file may not be a problem, but the cases where this is true are very rare. When securing a website, it is best to limit those policies as much as possible, and very few administrators do.

Cross-site Scripting attacks via Flash (AKA Cross-Site Flashing)

In 2007, Stefano di Paula published research at OWASP Europe detailing methods of injecting Javascript and Actionscript into unvalidated inputs within a Flash object. Dubbed "Cross-Site Flashing," this attack is a close relative of Cross-Site Scripting. Using a poorly coded SWF object as the attack vector provides several advantages to the attacker.

First among them is the ease of exploitation. While XSS is still only beginning to be understood by many web developers, Flash developers tend to be designers first and programmers second. Consequently, Flash objects are not coded with security in mind, and the number of vulnerable objects across the web is likely in the tens of millions, if not higher. To a savvy attacker, injecting Javascript into a Flash object is no more difficult than injecting Javascript into a web page.



Another reason that Cross-Site Flashing can be advantageous is that the server may not ever show any sign that it is under attack. While XSS attacks must be developed and tested on live sites, vulnerable SWF objects can be downloaded from the server, deconstructed, and exploits developed offline. When the attacker intends to perform the attack on a user, he can still prevent information from being sent to the server, even as it is being attacked.

HTTP Parameter Pollution is essentially a way of attacking a website by sending the same parameter multiple times. A simple example would be a query string in a URL:

http://example.com/?page=foo&page=bar

Depending on web technologies in use, various unexpected behaviors may arise. A Flash object can receive parameters in a similar fashion, but it will only use the final iteration of a name-value pair. Additionally, a hash mark in the query string will prevent part of a URL from being sent to a server, but will not prevent the Flash object from parsing the full string. Thus, the following URL may look like a perfectly normal request to the server while passing a malicious string to the Flash object.

http://example.com/file.swf?url=foo#&url=bar

Same Origin Policy Revisited

At this point in the discussion, one critical fact still needs to be addressed. A SWF object contains several methods to execute Javascript, and when loaded directly in the web browser (rather than embedded in a web page), it will execute that Javascript in the context of the domain it was loaded from. Thus an object served from foo.com will execute all Actionscript and Javascript in the foo.com domain.

However, when embedded in a web page on a separate domain (bar.com), that same Flash object may execute Actionscript in the foo.com sandbox and Javascript in the bar.com sandbox.

In theory, this behavior should cause no problem—a trusted Flash object *should* have access to the

server it is loaded from, and it should not matter what web page it is included in. However, this relies heavily on the abilities and attention to detail of the developer, the server administrator, and many other parties. With this in mind, several exploitation scenarios arise.

The first scenario involves embedding a malicious object in a web page on the targeted server. This is the simplest to prevent, as the ability to do so also implies an HTML injection vulnerability, which would enable Cross-Site Scripting. A similar attack would involve the attacker placing a useful SWF file on his own server and encouraging others to embed it in their web pages. Such widgets are common in the interactive web, but once in heavy use, it would be trivial for an attacker to replace that file with a malicious one, attacking any websites that used it.

In the second scenario, an attacker could corrupt an innocent but poorly designed Flash object on the target server. One method of doing so is Cross-Site Flashing, as discussed earlier. Another approach may involve a Flash object that performs calls to Javascript on the embedding page and takes action based on the results. As Javascript methods can be modified on the fly, it would be possible for an attacker to embed that vulnerable object in his own web page, modify the methods called by the Flash object's Javascript interface, and poison the results that are returned to it.

The final way to abuse Flash's "Two-Origin" policy is to place a malicious object on the target server, then embed that object in the attacker's web page. While it initially seems unlikely that an executable object could be uploaded onto the server, as most web developers have been warned about the risks of handling file uploads, there are several more Flash quirks that enable this attack, making it remarkably difficult for a developer to prevent.

Simply put, the Flash Player does very little validation of a file before executing it, short of ensuring the integrity and structure of the content. Whether this is a vulnerability in Flash player may be arguable, but the result is that many webservers will both accept and serve back SWF files thinly disguised as other filetypes simply by changing the file extension of the object to "jpg," "gif," or something similarly innocent. Flash Player, in turn, will readily execute those files.

Some web applications, however, do perform analysis of uploaded files before allowing them to be served back to the user. An image file that is clearly not an image file will be rejected as invalid. This validation may still be bypassed by creating a file that is **both** a valid Flash object and another, innocent format.

The SWF file format used for Flash objects requires a specific set of bytes at the beginning of the file, but it may have an arbitrary amount of "junk" data at the end of the file and still execute. The ZIP format, on the other hand, allows junk data at the beginning of the file, and the actual data can be placed at the end (technically, the ZIP format does not explicitly allow such junk data, but it does not prohibit it, and conventions, implementation, and variations on the standard regularly use this feature for metadata and multi-format files such as the one described here).

With this information in mind, it is trivial to create a file that is simultaneously a valid Flash object and a valid Zip file. Considering that many web applications allow uploads of Zip files (indeed, some webmail applications explicitly require it for attachments), this provides a wide variety of sites that malicious files may still be uploaded to. Considering that the Zip format is also used for files such as Microsoft Office Open XML documents, XPI files, and JAR archives, the attack surface begins to appear a bit wider.

This vulnerability is extremely common in certain types of websites, specifically those that are designed to handle files. Webmail applications such as Gmail and Squirrelmail, document repositories, online forums, and syndication sites have all been found vulnerable.

At the time of writing, even Adobe's bug tracking application, which was created by a third party, will allow Zip files with embedded (and executable) SWFs to be uploaded. This is important because it demonstrates how these attacks can be combined: www.adobe.com's crossdomain.xml file explicitly allows access from *.adobe.com. If a malicious object were uploaded to bugs.adobe.com and executed, it would not only affect that application, but it could affect Adobe's main website as well.

This file upload attack can only be prevented by a server's administrator being extremely careful about accepting content from users. He must specifically account for it, or take extreme defense in depth measures that simply are not available to most websites. This attack is extremely difficult to prevent, and quite a few open source, commercial, and custom built applications are vulnerable.

It should be noted that Flash Player 10,0,02 provided a partial fix, or at least a way for administrators to better protect themselves. Files served with a “Content-Disposition: attachment;” HTTP header will not be executed by Flash Player. If possible, all user-supplied content should be served with this header.

Conclusion

While this is only scratching the surface of the ways that web browsers and applications can be abused by interactive web technologies, it does provide a broad overview that can help a security administrator evaluate his own risk. From server-side configurations such as crossdomain.xml to application logic issues like input validation, there are many matters that should be considered. To be sure, Flash Player is useful technology—virtually a requirement in order to use the web—but it does present certain risks which are rarely taken into account when designing a website, maintaining a server, or simply using a web browser.