How I Learned to Stop Worrying and Love Plugins

Chris Grier Samuel T. King
University of Illinois

Dan S. Wallach
Rice University

Abstract

This position paper argues that browsers should be responsible for specifying and enforcing security policies for browser plugins. By enabling the browser to make security decisions on behalf of the plugin, browsers can significantly reduce the impact of plugin vulnerabilities and eliminate much of the risk posed by today's plugin exploits. We propose policies for document access, persistent state, network connections and other devices that browser-based security policy can implement.

1 Introduction

Web browser plugins have become a ubiquitous tool on the Internet for videos, music, and documents. The introduction of new, feature-rich plugins has revolutionized web applications; for example, Flash Player is behind what makes YouTube work – without the streaming video support added in Adobe Flash 7, YouTube would not have taken off [7].

Unfortunately, plugins are riddled with security vulnerabilities and expose users to significant risk. Plugins are the single largest source of vulnerabilities in browsers today, accounting for 476 reported vulnerabilities in 2007 compared to 163 for browsers, including IE, Firefox, and Safari combined [13]. Each plugin is responsible for implementing its own security policy and enforcement mechanisms that fail when an attacker can exploit a plugin. To interact with the browser, plugins use a plugin API, such as the NPAPI [9], supported by the browser. Plugin APIs differ across browsers but in general, the browser provides plugins with document (i.e., DOM) access, network connectivity and interaction with other browser components. Except for some document accesses, each plugin is responsible for restricting the use of this API by the plugin content as well as controlling access to the underlying system.

Allowing plugins to enforce and define security policies has completely failed to prevent plugin attacks from damaging systems. Many plugin vulnerabilities enable code execution attacks, resulting in modification of the underlying system and damage to the browser. One recent example is a malicious PDF advertisement that exploited the PDF browser plugin to perform drive-by-downloads and install malware automatically [10].

Although mechanisms for plugin security are well understood, current policies are either too restrictive and break commonly used sites, or are too permissive and provide limited security benefits. To provide isolation from exploited plugins, browsers such as OP [3], Chrome [1], and Gazelle [4] place plugins in a separate process, isolating the plugin from the rest of the browser while exposing an interface for plugins to communicate. OP and Gazelle both implement an interface that imposes security policies on plugin instances. OP does so by implementing policies specifically designed for plugin use while Gazelle uses a standard browser policy across all resources. Both policies are too restrictive and impose unacceptable limitations on plugins, such as restricting network access to the domain of the content provider, breaking popular sites such as YouTube. By default Chrome only isolates plugins. Further sandboxing can be applied by Chrome but even this can cause compatibility issues [1].

Our position is that using browsers to impose control on plugins offers the best combination of security and flexibility. Using a browser architecture that places plugins into separate processes provides a barrier that the browser can leverage to control and interact with the plugin. The browser should be responsible for controlling access to page resources (document tree), persistent state (cookies and other local storage), network connections, and to other devices. The browser is already responsible for instantiating the plugin in modern browsers and understands page semantics, allowing it to make meaningful security decisions.

We propose several policies that strike a balance between security and flexibility and our goal is to foster an open discussion about plugin policies. In Section 4 we provide a summary of each plugin policy for document, network, persistent state and device access.

2 Benefits

By using the browser rather than the operating system to control plugins, the browser can use additional semantic information from web pages for plugin-related security decisions as well as provides a single location for all security decisions. Figure 1(a) shows current plugin architectures while Figure 1(b) presents our proposed architecture and restricted interfaces. Our goal is to provide browser resilience against an exploited plugin. Although this seems ambitious, most of the mechanisms to properly restrict plugins have been developed by previous research, though none define appropriate policies for plugins. Our design aims to provide fault tolerance and protect both the browser and system from plugin vulnerabilities.

Isolating and containing each plugin in a separate process allows the browser to tolerate plugin bugs that would normally crash the entire browser and contain attacks inside the plugin process. Data from IE crashes indicate that over 70% of browser crashes were due to plugins [15], demonstrating the need for increased reliability.

In addition to fault isolation, external security mechanisms are necessary to limit the ability of the plugin to interact with the operating system directly. Sandboxing systems such as systrace, NativeClient and XaX enable application level safety properties that we can use to restrict plugins on different platforms [8, 14, 5]. Without sandboxing, a plugin vulnerability exposes not only the browser to attack but places the whole system at risk. By restricting the plugin from accessing the local system and forcing it to use browser provided APIs we can prevent damage to the underlying system.

To prevent the browser from being attacked or used to launch an attack, policies must also limit a plugin's ability to corrupt and directly interact with the browser. Controlling browser access also prevents internal browser state from being exposed to plugin attacks.

3 Plugin capabilities

The challenge of securing plugins is to maintain a large range of functionality and enable feature-rich applications while at the same time providing safety and resilience to plugin exploits. The diverse functionality of plugins makes them useful a wide range of web applications, but also poses a major security threat if left unchecked. In this section we discuss the different types of plugin permissions on document access, persistent state, network connections and other device access.

Document access. The browser plugin API was originally designed to allow embedding new types of content and provided direct screen access for plugins to draw in. As plugin and browser support evolved, plugins were given additional access to internal browser structures without any limitations. As the document tree interfaces were standardized, plugins retained access to core document and window objects.

Today's plugins can access the window contents, including the document tree, JavaScript global environment, and cookies. The plugin API includes "scriptable" extensions that allow two-way interaction between page and plugin content using JavaScript. Plugin content can expose functions for JavaScript in the page to call, and the plugin can execute JavaScript in the context of the page. In the current model, the browser is completely unaware of the content of the interaction, and through JavaScript, plugins are given full access to page-level resources.

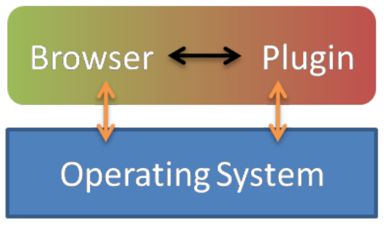
Persistent state. Plugins maintain their own files on the local file system for storing persistent state across browsing sessions without any interaction with the browser. Plugin storage is separate from browser cookies, where accesses are performed according to same-origin policy. Rather than allowing plugin content to access the file system directly, current plugin implementations in Flash and Silverlight each provide their own notion of secure storage on top of the local file system layer, and allow plugin content to access the per-plugin storage.

Plugins like Flash and Silverlight provide a domain-based storage mechanism with different access policies than browser cookies, presenting a interesting discrepancy between the two storage mechanisms. "Flash cookies" are the Flash Player terminology for domain-based storage, and are similar to browser cookies except they allow different types of data and are stored using the file system, *not* browser cookies. These flash cookies have been the subject of recent privacy concerns because Flash requires a specialized mechanism to view and remove stored data [2].

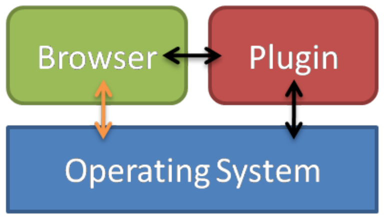
Network access. Plugins make network requests by accessing host networking APIs directly and through the browser provided plugin API. NativeClient recognizes local socket access as a security risk and cuts off plugins from local network access. Other plugins, such as the Java Runtime, Flash and Silverlight each have their own policy for restricting network access that differs from the browser's same-origin limitations on JavaScript.

Differences in networking policies between the browser and plugins expose interesting attack vectors and creative work arounds. A current "solution" to same-origin network limitations in JavaScript is to use a Flash as a proxy for XMLHttpRequests, an XML-based asynchronous network request, since Flash has no such same-origin restrictions. Using Flash to issue network requests on behalf of JavaScript subverts the same-origin policy limitations intended by browsers. DNS rebinding attacks [6] also become a problem since each plugin has a different policy for resolving and pinning DNS entries internally.

Device access. As with any general purpose application, plugins can access hardware attached to a system. These devices include the screen, microphones, speak-



(a) The current architecture for including plugins. Both the browser and plugins are able to make system calls and interact with the OS directly (orange lines).



(b) Suggested plugin architecture. The plugin is given sandboxed access to the OS and interacts with system resources through the browser.

Figure 1: Plugin architectures. Black lines are plugin specific, restricted APIs. Orange lines are unrestricted access.

ers, and video cameras just to name a few. These resources can be used by attackers, as demonstrated by a recent proof of concept exploit that tricks users into enabling the microphone using the Adobe Flash Player, turning a malicious flash applet into a mechanisms for spying on users *without exploiting the plugin* [11].

Plugins also are given access to the screen in order to render content. The screen is a new area to consider when dealing with plugin security. Plugins such as Flash and Silverlight include the ability to resize and full-screen without interacting with the page embedding the plugin content. Plugins can render any content, possibly even mimicking browser and window manager chrome.

4 Proposed plugin policies

In this section we outline potential policies for document, network and persistent state access that can be applied to plugins. We assume that the plugin is running in a separate process from the browser and all interaction with the underlying system is performed through the browser. Using this architecture, policy can then be enforced on the calls into the browser to provide secure execution of plugin content. Although each policy restricts plugin functionality, we hope to achieve a balance of functionality and security with each policy.

4.1 Document access

By limiting access to the document, plugins can be prevented from stealing valuable browser state as well as hijacking control of the page. Each document access policy restricts the information available to plugins in order to prevent information theft by malicious plugin content, while still maintaining most of the functionality the plugin expects and relies on.

Clean document. Rather than direct document access, the clean document policy provides the plugin content with a different view of the document that has been sanitized by the browser. Sanitization involves removing any non-standardized content and text from the document tree retaining the original document structure and element types but without any page-specific data.

Copy-on-write document. With a copy-on-write document, the plugin is provided with access to the original document (or a clean document). Any modifications to the document by the plugin cause a copy of the document to be made and changes to the page written to the copy. The modified document can then be committed or reverted, similar to the cookie policy from Doppelganger [12].

Proxy objects. Instead of interacting with the page directly, the plugin is given access to a set of objects that act on behalf of the plugin. Proxy objects are constrained by capabilities set for them by the page, and any accesses are subject to the proxy object policy.

Rooted subtree. In the rooted subtree policy, a plugin is given access to a limited portion of the document tree. From a subtree (or clean subtree) of the document, the plugin can only access nodes inside the subtree. The plugin has full access to the subtree and can add, update, and delete nodes.

4.2 Network access

Network policies restrict network requests based on IP, domain or other identifiers determined by the browser. For most plugins, JavaScript's same-origin policy will cause unacceptable restrictions. If we try and force same-origin policy onto plugin network connections we are removing significant functionality that makes plugins useful.

Server-specified allow. The server-specified allow is a variation on Flash crossdomain.xml policy where the plugin developer includes network-level access control specification on the destination server. Developers can specify restrictions in the page itself as attributes on the plugin element, or by a server-side file hosted in a fixed location. In addition, the destination of the outgoing network request can further deny access to resources.

All or one. The plugin content makes an implicit choice of document access or network access in the all or one policy depending on how the plugin behaves. If the plugin content accesses the document, the browser restricts network access to prevent cross-domain communication, including the ability to create cross-domain images or scripts in the document. If the plugin content accesses cross-domain network resources, the browser denies access to the document tree and all other browser states.

Same-company policy. Similar to same-origin policy, the browser restricts access based on the company that owns a set of domains and origins, redefining the trust boundary from the origin to the company. This alleviates the burdens of same-origin restrictions and enables content delivery networks and cooperation between services owned and trusted by a single company. To determine the corporate boundary, the browser can use DNS registration information.

4.3 Persistent state access

Persistent state policies restrict file system access to prevent an exploited plugin from accessing arbitrary files. By allowing plugins to access a select set of files on the system and carefully controlling file system access, browsers can enable safe access.

Automatic local and global files. To differentiate between global plugin state and site-specific plugin state we identify both types of data automatically. In our technique we install and initialize the plugin and any files created during installation are considered to be global states – all plugin instances are given read/write access to these global state files. After the plugin has been installed, we consider all subsequent files created by plugin instances site-specific states, which are protected according to our browser-level security policy.

For site-specific states, we redirect all file system calls to the browser, which maintains a private storage mechanism that is controlled by the browser security policy. Our persistent state policy separates site-specific states based on the domain of the content to avoid cross-domain interference.

Jailed accesses. This browser provides a virtual folder (e.g., Unix `chroot`) where all file system accesses from a plugin are performed. The virtual folder provides containment from the rest of the file system as well persistent storage across browsing sessions. The browser provides each plugin instance with a different virtual folder depending on the plugin content domain.

4.4 Device access

Device policies restrict access to devices based on the device functionality. For example, a policy could prohibit a plugin from listening on the microphone or allow access to a video camera. Each policy limits the interaction with the underlying hardware to provide safety during a plugin attack while allowing access to the core functionality that a plugin requires.

Page permitted, user granted. Using the page permitted, user granted policy the page including the plugin specifically allows access to a device in addition to the user granting access to the device requested.

Capability enforced. In this policy the plugin requests a set of capabilities for each device. For example, a plugin can request only camera access, the browser then prohibits other access to the plugin. Unsafe capability requests, such as camera and network access, would require authorization by the user.

5 Conclusions

Using a browser architecture to support and secure plugins can be an effective means to mitigate the threats posed by plugin vulnerabilities. We proposed browser-enforced policies for plugins that may potentially improve security without affecting plugin functionality for today's Web. We are currently in the process of determining the effects of our policy on common plugin use and will present our findings at the conference.

References

[1] A. Barth, C. Jackson, C. Reis, and The Google Chrome Team. The security architecture of the chromium browser, 2008. `http://crypto.stanford.edu/websec/chromium/chromium-security-architecture.pdf`.

[2] Electronic Privacy Information Center. EPIC Flash Cookie Page, 2005. `http://epic.org/privacy/cookies/flash.html`.

[3] C. Grier, S. Tang, and S. T. King. Secure web browsing with the op web browser. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, 2008.

[4] C. Grier, H. J. Wang, A. Moshchuk, S. T. King, P. Choudhury, and H. Venter. The multi-principal os construction of the gazelle web browser. In *To appear in the Proceedings of the 18th USENIX Security Symposium*, August 2009.

[5] J. Howell, J. R. Douceur, J. Elson, and J. R. Lorch. Leveraging legacy code to deploy desktop applications on the web. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, December 2008.

[6] C. Jackson, A. Barth, A. Bortz, W. Shao, and D. Boneh. Protecting browsers from DNS rebinding attacks. In *Proceedings of the 14th ACM conference on computer and communications security (CCS)*, 2007.

[7] J. Karim. YouTube: from concept to hypergrowth, October 2006. Video clip.

[8] A. Kurchuk and A. D. Keromytis. Recursive sandboxes: Extending systrace to empower applications. In *Proceedings of the 19th IFIP International Information Security Conference (SEC)*, August 2004.

[9] Mozilla. Netscape Plugin API. `http://www.mozilla.org/projects/plugins/`.

[10] B. Prince. Attackers Infect Ads with Old Adobe Vulnerability Exploit, February 2009. `http://www.insidetech.com/news/articles/4146-eweek-ads-infect-users-thanks-to-adobe-flaw`.

[11] RSnake. Clickjacking details, October 2008. `http://ha.ckers.org/blog/20081007/clickjacking-details/`.

[12] U. Shankar and C. Karlof. Doppelganger: Better browser privacy without the bother. In *Proceedings of the 13th ACM conference on computer and communications security (CCS)*, 2006.

[13] Symantec. Symantec Global Internet Security Threat Report Trends for July-December 07, April 2008. `http://www.symantec.com/business/theme.jsp?themeid=threatreport`.

[14] B. Yee, D. Sehr, G. Dardyk, J. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, May 2009.

[15] A. Zeigler. IEBlog: IE8 and Loosely-Coupled IE (LCIE), March 2008. `http://blogs.msdn.com/ie/archive/2008/03/11/ie8-and-loosely-coupled-ie-lcie.aspx`.