sePlugin: Towards Transparently Secure Plugins in Your Internet Explorers

Lei Liu[1], Xinwen Zhang[2], Guanhua Yan[3], and Songqing Chen[1]

[1] Dept. of Computer Science, George Mason University
{lliu3, sqchen}@cs.gmu.edu
[2] Computer Science Lab, Samsung Information Systems America
xinwen.z@samsung.com
[3] Information Sciences, Los Alamos National Laboratory
ghyan@lanl.gov

Abstract. To support a variety of web applications with diverse contents such as video and audio, plugins have been widely adopted to extend the functionality of existing web browsers. Although instrumental for developing web applications efficiently, the open mechanism of plugins poses tremendous threats to browser security. While a plethora of approaches have been proposed to improve the overall security of web browsers, their treatment of plugins is either incomplete (e.g., missing many plugins for browser extensions) or demands code modification for legacy browsers and plugins or introduces significant performance degradation. Against this backdrop, we propose a novel in-process sandboxing mechanism called *sePlugin* to monitor and confine suspicious behaviors of plugins in popular commodity browsers such as Microsoft Internet Explorers. sePlugin enhances the security of a browser by controlling how plugins access the browser's internal objects and external system-level resources such as file systems and network interfaces. sePlugin deals with both native and .NET-based plugins and its unique design renders it possible to work with commodity web browsers without requiring any modifications to the legacy browser architecture or plugin code. We have implemented sePlugin in IE8 under Windows XP and IE8 and results from a number of experiments attest to both its strong capability of security policy enforcement and its low operational overhead (only 4.46% of browser loading time).

Key words: Plugins, Browser Security, In-process Sandbox, Web Security

1 Introduction

Since the inception of WWW in the early 90's, web contents have evolved from simple static plain text to dynamic and rich media contents that are common in the current Internet. One common approach to extending web browser capabilities is using plugins. From a functionality point of view, there are two types of plugins in contemporary web browsers: *browser extensions* and *content extensions* [13]. Browser extensions allow software developers to add customized functionalities to existing browsers so as to improve user interfaces and browsing experience. For example, Microsoft Internet Explorer (IE) defines various browser extensions such as toolbars and browser helper objects (BHOs). Other popular browsers like Firefox also provide browser add-ons for similar functionalities [12]. On the other side, content extensions enhance a browser's capabilities of rendering web objects, such as audio clips, streaming files, and flash videos.

While plugins greatly improve users' browsing experience, they do not come as free: plugins commonly run third-party code in the same process space as the browser, thus increasing the complexity of a browser's internal structure and potentially introducing new vulnerabilities. This actually has been evidenced by the ever-increasing attacks that target web browsers through plugins. A recent study [21] shows that nowadays a large portion of malware infections are in the form of plugins on web browsers such as IE. An overwhelming majority of malware have a browser extension such as a BHO or a browser toolbar. Another study [25] has shown that out of 120 distinct spyware programs, about 90 use BHOs as an entry point to monitor user activities, 46 use the IE toolbar mechanism, and some malware programs even use both mechanisms. BHO, one of the most commonly used plugins, is named by CERT [19] as one of the most frequently used techniques employed by spyware writers along with stand-alone applications. In addition, although attacks have been found on other platforms and browsers such as Linux and Firefox [20], as of today the occurrence of malware affecting other platforms and browsers is much lower than that affecting Microsoft Windows and IE.

The threat posed by the increasing usage of plugins is rooted from the fact that mainstream web browsers like IE have been lacking a comprehensive security model for plugins for many years. As concerns over security breaches through browsers have mounted in the past few years, there have been plenty of research dedicated to improving browser security [21, 7, 26, 23]. In particular, several new browser architectures have been designed recently to provide a complete solution by taking security as one of the fundamental requirements. The major objective of these approaches is to integrate a security model into the browser execution environment at different levels, and most of them enforce strong isolation using the same-origin policy (SOP) [14]. Some advanced browsers such as OP browser [18], Gazelle [24] and Chrome [1] run plugins in separate plugin processes rather than the browser process and enforce security policies on individual plugin processes.

Even with all these efforts that take security seriously in browser design, security threats posed by malicious plugins still have not been fully alleviated. *First*, most of these new browser designs only consider plugins used as content extensions [24], which can be secured by breaking them into separate OS-level processes and adopting the SOP policy. For plugins used as browser extensions, another major type of plugins introducing numerous security issues, these new architectures typically do not provide effective solutions. In particular, as most browser extensions are designed to handle information that is distributed over the entire browser, it is difficult to split them from a browser's kernel and apply the traditional SOP policy. *Second*, to enforce strong isolation, most existing solutions divide the browser system into components including plugins that run as OS-level processes, and OS-level sandboxing techniques (e.g., SELinux) are used to constrain the interactions among different components and between components and the OS. As a result, inter-process communications inevitably introduce high computational cost in these new architectures. For example, in the OP browser, the X-server is required to merge GUI display, which causes considerable performance problems as acknowledged by the authors [18]. *Third*, nearly all these solutions demand an overhaul of major browser components including the browser kernel and the plugins. The wide deployment of legacy web browsers such as IE and a large number of existing plugins suggest that a practical solution can only survive if it is compatible with these legacy applications without introducing too much performance overhead.

To this end, we propose in this paper *sePlugin*, a new mechanism that provides security protection to web browsers and the underlying OS by transparently sandboxing third-party plugins. Different from existing solutions, sePlugin does not isolate each plugin or each plugin instance into an individual OS-level process. Instead, sePlugin builds an intra-process sandbox for each plugin, which monitors accesses from the plugin to critical resources including internal browser objects and OS resources (e.g., file systems and network interfaces). Meanwhile, sePlugin enforces a set of high-level security policies to control accesses that are originated from plugins. In a nutshell, the distinguishing features of sePlugin can be summarized as follows: (1) it works with the commodity web browsers without requiring code modifications to legacy browser architecture or plugin code; (2) it is lightweight and introduces as little as 4.46% performance overhead for fine-grained access control; (3) it offers a comprehensive solution to enforcing various security policies on both types of browser plugins; and (4) it deals with both native and .NET-based plugins. We have implemented sePlugin in Windows XP and IE8 and the results from a variety of experiments show that sePlugin is a viable solution in practice due to its strong protection capabilities and low computational overhead.

The remainder of the paper is organized as follows. Section 2 introduces related work and also highlights some differences between sePlugin and existing browser architectures. Section 3 presents a high-level description on how we design sePlugin. We continue to elaborate on the implementation details of sePlugin in Section 4 and discuss the access control enforcement in sePlugin for IE8 in Section 5. In Section 6, we present results about the effectiveness and efficiency of sePlugin from a set of experiments with real-world plugins. Section 7 concludes this paper.

2 Related Work

Due to continuous threats from web browsers [15, 11, 19], a lot of efforts have been made to re-define the browser's software framework. The OP browser [18] breaks a browser into several isolated components, each of which runs as a separate OS-level process. The OS-level sandboxing technique is used to control the permissions of plugins, which thus reduces the likelihood of illegitimate accesses. Thus, OP can enforce different types of security policies including the SOP policy. Gazelle [24] is a secure web browser based on IE. Gazelle identifies web site principals defined by the SOP policy and puts principals into separate protection domains. The protection domain of a principal instance is a restricted/sandboxed OS process. The use of processes guarantees the isolation of principals even in the face of attacks that exploit memory vulnerabilities. Such a multi-principal OS construction brings significant security and reliability benefits to the overall browser system. Google Chrome [1] isolates the browser rendering engines (including HTML, CSS, image, and JavaScript) from the browser kernel by putting them into two different processes with the process-per-site-instance model by default. For plugins, Chrome does not provide strong isolation. Plugin contents from different principals or sites share a plugin process in Chrome, and the process runs in the browser user's full privilege by default. Therefore, it can access OS resources with the user's permission rights.

All the above designs restrict plugins' accesses to the browser objects with IPC control. In addition, they often require modifications to legacy browsers and plugins and thus cannot enhance the security of mainstream web browsers such as IE. Moreover, these newly designed browser architectures only consider plugins for content extensions, leaving browser extensions without protection. It is actually difficult to break plugins for browser extensions such as BHOs and toolbars into separate processes as they usually are designed to handle information distributed over the entire browser, and it is not clear how to define origins for them. As most of the above architectures leverage OS-level sandboxing mechanisms to constrain plugins' behavior and enforce security policies, they inevitably introduce significant performance overhead for web browsing.

As a spyware containment technique specifically targeting browser plugins, SpyShield [22] offers a general protection mechanism against surveillance by spy add-ons such as BHOs. SpyShield blocks the access of an untrusted add-on whenever sensitive data are being accessed by its host application (a browser). To achieve this, SpyShield inserts an access-control proxy between untrusted add-ons and their host applications to control their communications by enforcing a set of security policies on add-on interfaces. Optionally, SpyShield provides an one-process solution to run plugins within a browser process or a two-process solution to run plugins as independent processes. SpyShield, however, has access control only for browser objects, and it thus does not refrain malicious native plugins from accessing OS resources. Furthermore, SpyShield only deals with COM-based plugins, and its two-process approach, albeit optional, also brings high inter-process communication (IPC) overhead. Lastly, the access control polices in SpyShield have to be configured by the user, which is often error-prone. Compared with existing approaches, sePlugin aims to provide comprehensive confinement not only for content extensions, but also for browser extensions. The protection service is not only for browser objects, but also for OS resources.

3 sePlugin Design

3.1 Threat Model

sePlugin protects browser objects and system resources from malicious plugins. Therefore we assume that the IE browser kernel is trustable so that it works as originally designed and can enforce default security policies. We further assume that other components in IE such as the COM rendering engine and the JavaScript engine are not compromised. In our current design we consider neither collusion attacks between malicious plugins nor malicious active contents of a web page loaded by IE (e.g., malicious JavaScript code that actively sends sensitive data obtained from a web page to a plugin, leading to information leakage to outside attackers). Instead, sePlugin focuses on enforcing security policies on active access requests from plugins via pre-defined interfaces.

On the other side, sePlugin deploys a sandbox for each plugin, which intercepts and controls all possible access attempts from the plugin to browser and system resources. Thus, a plugin cannot bypass sePlugin. Furthermore, to compromise the integrity of sePlugin, a malicious plugin has to access some browser objects or system resources, say, by modifying the implementation of sePlugin or change its security policies. Certainly the plugin has to invoke some APIs to achieve these objectives, which are controlled by sePlugin itself. Thus we have the assurance that during runtime, sePlugin cannot be bypassed or compromised, due to its advantageous position in implementation. Provided this, we assume that all resource and physical memory access requests from a plugin are through pre-defined APIs (e.g., COM interfaces) or system calls via Windows API, which is the case for IE on Windows platforms. We do not consider attacks from plugins with direct kernel code calling for physical memory access and system calls. This is a feasible assumption in practice: *malware directly calling kernel code are rare because they are prone to cause browser crashing* [2].

3.2 Overview of sePlugin

sePlugin aims to create a secure and controllable environment for executing all kinds of plugins while minimize performance degradation due to such protection. For this purpose, sePlugin heavily relies on in-process sandboxes, which we will illustrate in more detail later. Besides in-process sandboxes, sePlugin also includes a sandbox manager and an entry point, called *proxy*, which is used when a plugin is to be loaded. Figure 1 shows a high-level of the sePlugin architecture.

As shown on Figure 1, in-process sandboxes are created and managed by the sandbox manager upon the execution of plugins, with assistance from the proxy. To deal with various types of plugins and system resource accesses, the in-process sandbox further consists of interfaces dealing with COM objects, system calls and .NET-based plugins respectively, and a policy enforcement part. In the following subsections, we will present more details about these components.

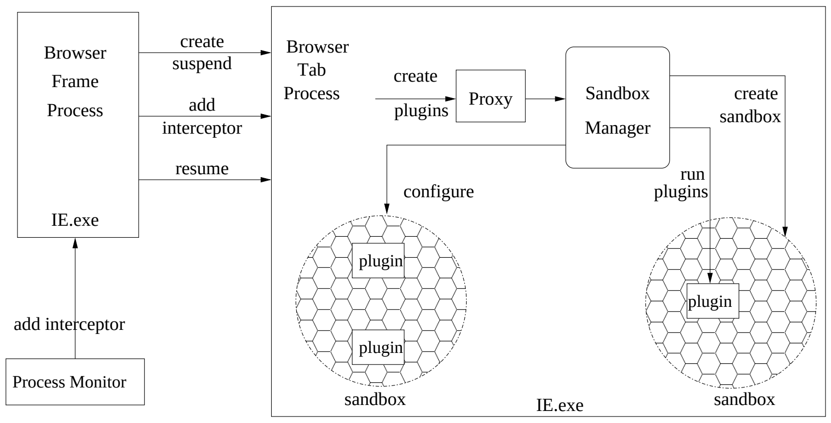

Fig. 1. sePlugin Overview

3.3 Sandbox Manager

The sandbox manager is the center of sePlugin. It generates, manages, and configures all in-process sandboxes in a browser process. After creating an in-process sandbox, the sandbox manager can alter the security configuration at run-time by changing its security policies. As the user navigation changes, the sandbox manager can also change the origins that plugins are allowed to access.

3.4 Proxy

As shown in Figure 1, whenever the browser process is about to create a plugin, the sandbox manager creates an in-process sandbox to contain that plugin. This is done with the assistance of a proxy. This proxy, an important component in sePlugin, is the entry point for a plugin to connect with sePlugin. In sePlugin, this proxy works as follows:

– When a web browser loads a plugin (called *target plugin*), a proxy is created first. The proxy implements the same COM interfaces as the target plugin to be loaded. This enables the web browser to easily load and run a proxy like a real plugin.
– The proxy also works as a delegate for communications. All COM communications thereafter to the target plugin are forwarded by the proxy.

3.5 In-process Sandbox

An in-process sandbox is the running environment created by sePlugin to execute plugins. To reduce performance impact while providing strong security enforcement, sePlugin relies on in-process sandboxes (instead of OS-level process sandboxes). An in-process sandbox is a runtime environment within an OS-level process. It provides a layer that monitors and controls the communications between the code running inside the sandbox and the rest of the process and the underlying OS. It also offers interfaces to change configurations for different security purposes. Thus it allows to enforce access control on plugins running inside. On the other side, since plugins are not implemented as separate processes, the overhead is low due to no IPC involved.

On Windows platforms, COM [9] is the base of a Windows application and all web browser plugins are also COM objects. Here, we present the design of in-process sandboxes for COM-based applications (e.g., IE). With sePlugin, COM-based plugins can be put into different in-process sandboxes according to their security requirements, and any plugin inside an in-process sandbox cannot directly access resources such as other IE components and the underlying OS outside the sandbox.

In sePlugin, an in-process sandbox performs two major tasks, *component isolation* and *security enforcement*. To accomplish component isolation, an in-process sandbox has a COM interceptor to intercept all COM communications, and a system call interceptor to intercept all system calls. To support .NET-based plugins, an in-process sandbox also has a .NET interceptor. Besides interceptors, in-process has a security checker for access control. Compared with other alternative solutions such as the OP browser and Gazelle, an in-process sandbox has the following advantages: (1) While an in-process sandbox enforces security isolation just like a process-level sandbox, computational cost is greatly reduced because all communications between components are local within the OS process. (2) For all COM-based applications such as IE, an in-process sandbox provides transparent access control so that it could be deployed on existing IE browsers without any source or binary code modification to the browser or plugins. (3) Flexible security policies can be defined and enforced by the in-process sandboxing mechanism. An in-process sandbox provides interfaces to set and change security policies. For various IE plugins, SOP and other high level policies can be enforced dynamically while the user is navigating multiple websites. (4) The computational cost of the in-process sandboxing mechanism is low. In-process sandboxes are only created for plugins rather than for the entire process.

COM interceptor All plugins in a web browser are COM objects. Although there is no mandatory restriction, plugins in IE are designed to access browser resources via COM interfaces. To control accesses to browser resources, an in-process sandbox uses a COM interceptor to intercept COM communications. A plugin can also access browser resources via direct memory operation, which will be discussed later.

The COM interceptor works closely with the proxy. When the proxy creates the target plugin, it also creates other proxy objects that implement web browser resource interfaces. For example, when a BHO plugin is created, the proxy creates another proxy object implementing interface *IID_IWebBrowser2* that presents a WebBrowser control. Because the proxy works as a delegate for the target plugin, those proxy objects are passed to the target plugin as interfaces to intended web browser resources. The combination of all these proxy objects provides a complete web browser environment for a plugin. The target plugin cannot communicate directly with the browser; instead, it has to use the proxy objects to communicate with the rest of the browser. All proxy objects are COM objects, and they act as a middleman for COM communications by the target plugin.

System Call interceptor In IE, plugins are designed to access browser resources via COM interface, but there is no mechanism that forbids plugins to access browser resources via other approaches. For instance, a plugin can directly read physical memory of other browser components via system calls or Windows APIs like *IESetProtectedModeCookie*. Besides browser resources, plugins can also access underlying OS resources through system calls, and can even compromise sePlugin through system calls. It is thus necessary to ensure that system calls issued by plugins are not malicious.

On the Windows platform, applications make system calls usually through Windows API. Although it is possible that malware evade the Windows API interceptor by calling kernel code directly, this is unusual in practice because as presented by threat model malware developers have to know the target operating system, its service pack level, and some other information in advance. Empirical results [2] show that most malware are designed to attack a large user base and thus use Windows APIs instead of calling the kernel code directly. Hence, sePlugin intercepts Windows APIs to monitor and control system calls made by plugins.

Interception of Windows APIs has been well studied for a long time [3]. In sePlugin, the challenge is how to achieve fine-grained interceptions. As sePlugin applies security policies to plugins, it needs to intercept system calls made only by plugins rather than by the entire process. According to work [3], such fine-grained interception of system calls can be achieved by modifying the Import Address Table.

Because malicious plugins run in the same memory context as the browser and sePlugin, malicious plugins have the potential to attack sePlugin. To protect the integrity of sePlugin, some system calls that could be used to attack sePlugin will be intercepted such as *WriteMemory*. In such a way, the system call interceptor ensures that sePlugin itself would not be compromised by the malware.

.NET interceptor In recent years, C# has gained great popularity among IE plugin developers. With Visual Studio, developing plugins like BHOs and toolbars with C# is easy. A .NET-based plugin is implemented by the Common Intermediate Language (CIL) [8]. Similar to the Java Virtual Machine, CLR's just-in-time compiler converts CIL code into code native to the underneath operating system [4].

A plugin developed with the .NET platform such as C# (which we call the *.NET-based plugin*) differs from a plugin developed with C++ (which we call the *native plugin*). Due to the different execution mechanisms, the COM interceptor and the system call interceptor do not work for .NET-based plugins. Hence, a new mechanism is needed specifically for .NET-based plugins. For CIL, although DynamicProxy.NET developed from the castle project [10] can generate lightweight .NET proxies at runtime, we find it difficult to use on IE. We thus take an alternative approach in sePlugin: we apply static code weaving tools like RAIL [17] to weave the interceptor code into the .NET-based plugin. Eventually, it loads only instrumented .NET-based plugins rather than original ones.

4 sePlugin Implementation

4.1 Sandbox Manager

In sePlugin, the sandbox manager is implemented as a BHO plugin. The sandbox manager is responsible for managing the life cycles of all in-process sandboxes, assigning target plugins to proper in-process sandboxes (or creating new ones if necessary), and configuring security policies for each in-process sandbox.

To apply security policies, the sandbox manager must be aware of the navigation status of the browser. The necessary information includes the plugin type, the current webpage content information including origins, and the security level (sePlugin defines different level of security policies which are explained in Section 5) from user configuration. In order to obtain all such information, as a BHO plugin, the sandbox manager registers two browser navigation events *DocumentComplete* and *BeforeNavigate2*. In section 5 we will further discuss how in-process sandbox enforces security policies on plugins.

4.2 Proxy and COM interceptor

Figure 2 depicts the interfaces provided by a proxy. As aforementioned, both the proxy and the target plugin are COM objects, the proxy creates the target plugin and forwards COM calls to the target plugin. Although *CoCreateInstance* is the general function to create a COM object, the proxy takes a different approach as follows to create the target plugin so that it shares the same COM apartment with the proxy.

- The proxy calls *LoadLibrary* to load the target plugin DLL into the memory.
- The proxy calls *GetProcAddress* with parameter *IID_IClassFactory* to get the address of procedure *DllGetClassObject*.
- The proxy calls *IClassFactory::CreatInstance* to create the COM instance.
- After the target plugin is created, the proxy will forward all related COM calls to the target plugin.

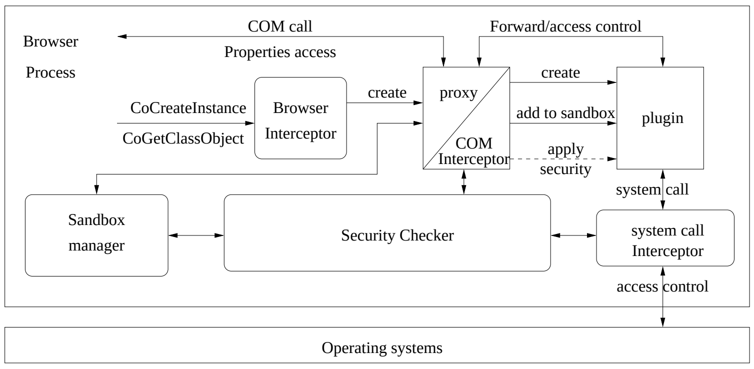

Fig. 2. Proxy Interfaces

Due to the variety of browser plugins, we have implemented a proxy that implements all the plugin interfaces for each type of plugins. The most commonly used plugins are BHO, toolbar and ActiveX Control. The toolbar proxy is implemented in a similar manner as BHO.

BHO Proxy: Normally, when IE creates a BHO plugin, it calls *CoCreateInstance*, which creates a single uninitialized object of the class associated with a specified globally unique identifier of a COM class object (CLSID). With sePlugin, when IE tries to load a BHO plugin, a proxy BHO is created instead. In many cases, by intercepting system call *CoCreateInstance*, sePlugin takes over the task of creating a COM object including BHO. sePlugin then reads the plugin information and executes the plugin in an in-process sandbox with proper security policies.

sePlugin creates a proxy for a BHO and the proxy has the same interfaces as the target BHO. The only interface required for the target BHO is *IObjectWithSite*. All requests to the target BHO are forwarded by the proxy. In this way, sePlugin successfully creates a chain of proxy objects that intercept, monitor, and control all COM communications of the target BHO.

ActiveX Control Proxy: If a plugin is an ActiveX control, IE calls *CoGetClassObject* instead of *CoCreateInstance* to obtain the class object of the ActiveX Control and create the ActiveX Control plugin from it. With a slightly changed procedure, sePlugin intercepts function *CoGetClassObject* and returns an ActiveX Control proxy class object. In this way, IE finally creates a proxy ActiveX Control. Other procedures are similar to that of the BHO proxy.

4.3 System Call Interceptor

As mentioned in the threat model, malware usually make system calls via Windows API rather than calling kernel code directly to attack a larger user space. In the following discussion, when we mention system calls, they refer to system calls made through Windows APIs. There are many approaches for Windows API interception such as the proxy DLL, code overwriting, and altering the DLL import address table [3]. sePlugin has some system calls that need to be intercepted for the entire process because those calls are usually related to the program flow. Those system calls include process creation calls such as *CreateProcess*, COM creation calls such as *CoCreateInstance* and *CoGetClassObject*. Moreover, system calls that directly access web resources like *IESetProtectedModeCookie*, access system resources like *readFile* and *WriteFile*, or perform network communications like *bind* and *connect* are also intercepted. For such system calls, we use Detours [5] for code overwriting. After the code is overwritten by Detours, all these system calls from the entire process are intercepted.

On the other hand, to reduce the performance impact, for other system calls, we only need to enforce access control on plugins within the in-process sandboxes. As plugins are in the form of DLL, we use the DLL import table alteration approach to add system call interceptors to plugins [3] for performance considerations. More specifically, when a proxy loads the target plugin by calling *LoadLibrary* which has already been intercepted by Detours when the browser process was launched, the Import Address Table [16] of the target plugin is altered and the function addresses are replaced with the interceptor's function addresses. The interceptor forwards system calls to the original system call after sePlugin performs security checking. With this method, only system calls from in-process sandboxes are intercepted while the rest is not affected.

4.4 .NET Interceptor

In Section 3, we have shown how sePlugin handles .NET-based plugins. To add an interceptor to .NET-based plugins, we need a CIL code weaving tool. sePlugin uses RAIL [17]. Here is an example of a .NET-based plugin interceptor. Suppose that we need to add an interceptor to the method *void BeforeNavigate2(object, ref object)*, which a BHO calls to receive browser events before a web navigation occurs. RAIL requires an *epilogue* method and a *prologue* method with the same return value and parameters as the *BeforeNavigate2* method. With RAIL, we can weave these two methods into the plugin CIL code. The functions *proBeforeNavigate2* and *epiBeforeNavigate2* will be called before and after the *BeforeNavigate2* method is called, respectively. Hence, all resource accesses made by .NET-based plugins can be intercepted by in-process sandboxes and further be checked by the security checker. The instrumented .NET-based plugin is loaded at runtime as shown in Figure 3.

4.5 Security Checker

The security checker is the component in the in-process sandbox that actually enforces security policies. The security checker saves a complete resource access record of each plugin and performs security checking with security policies. We will present more details about security policies in following sections.

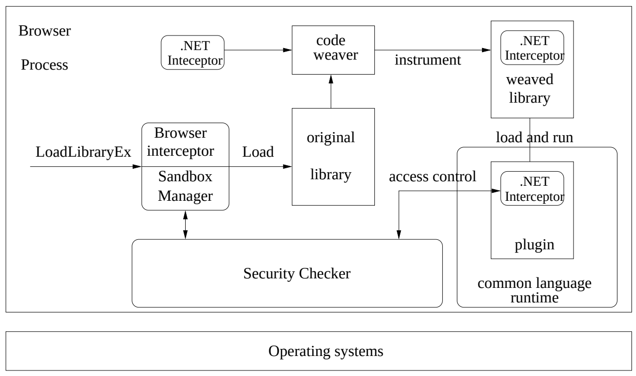

Fig. 3. .NET-based plugin overview

5 Securing IE with sePlugin

In this section, we explain how to deploy sePlugin in IE with a set of security policies. Note that even with a single page tab process like IE, there can be more than one plugins loaded, each of which has an in-process sandbox to enforce a different set of security policies from others. Furthermore, a single web page may have multiple origins as it can have contents loaded from different URLs.

5.1 Same Origin Policy for Content Extensions

Content extension plugins extend content formats that can be parsed, rendered, and displayed within a browser. Both Gazelle and OP browsers enforce a strong SOP for this type of plugins. Following the same security requirements, we also enforce the SOP with the in-process sandbox mechanism in sePlugin. Similar to Gazelle which puts plugins of the same origin in a single process, sePlugin puts plugin instances of the same origin into a single sandbox.

When a content extension plugin is created, it does not belong to any sandbox until a resource is assigned to it, e.g., by HTML tags or JavaScript code. For example, an ActiveX document server needs the moniker that represents the target document source. When function `IPersistMoniker::Load` is called by IE, the ActiveX knows the URL of the document resource and thus requests the sandbox manager for access privilege. A URL defines an origin. The sandbox manager applies the SOP to the calling ActiveX plugin. The SOP states that a plugin can only access objects with the same origin, and any access to objects of different origins from a plugin's origin is denied. For example, if the document is hosted in the same origin of the web page, the ActiveX document server can access all of its objects. However, if a third-party document is linked in the page, the ActiveX plugin can only access the document while it cannot read or write any object in the main browser space, including HTML and JavaScript, and other browser objects (e.g., cookies) of other origins.

5.2 Policies for Browser Extensions

Unlike content extensions, a browser extension works in the global context of the browser. Thus they are designed to be able to access all the browser resources rather than a specific type of contents in the browser. Thus, the restrictive SOP, however, cannot be directly applied to a browser extension because there is no origin for it by default. Neither OP nor Gazelle provides an effective approach to controlling the behavior of browser extensions. SpyShield forbids BHOs to access a web page with sensitive information. It is, however, difficult to request end-users to classify sensitive documents. Therefore, SpyShield's approach is conservative by directly forbidding plugins to access the entire web page.

We adopt an optimistic approach with three levels of security policies. By default, under any of these policies, a browser extension cannot directly modify any browser or web page contents, such as document URLs, DOM trees, JavaScript code, and cookies. However, all these three policies allow a browser extension to read information from the browser and the web page, while they have different levels of restriction on possible information leakage after the reading operation. In the following discussion, we describe these policies from the most restrictive to the most relaxed.

Relaxed Same Origin Policy In the most restrictive manner, the SOP-like control can be applied to browser extensions, similar to that for content extensions. As a browser extension does not have an origin because it does not process contents from a particular origin, we assign an origin for it when a new web page is loaded. More specifically, a single web page may contain a set of origins. When it is loaded by the browser, a new sandbox is created by the sandbox manager in sePlugin with its origin name set to be the concatenation of all the origins of this web page. A loaded browser extension is then dynamically assigned to this sandbox. Therefore, the SOP for this sandbox is: the browser extension can read any data and information in all the origins of the loaded web page, and it can talk to any of the origins via network connections or HTTP requests. Therefore, this policy can be essentially regarded as a relaxed SOP: the origin of the browser extension is the set of all origins in the web page; a browser extension can access information in any of the origins but cannot leak any data to the outside, and a browser extension cannot change or add new origins to the web page, e.g., change the HTML contents or insert/link JavaScript code from a third-party web page.

Protecting Sensitive Browser Information Although the relaxed SOP policy prevents any sensitive information from being leaked out of the origins of a loaded web page, it also prohibits normal behaviors by many benign plugins. For example, many toolbars are deployed on IE and other browsers for quick search, which should be allowed to communicate with web servers different from the loaded web page.

Therefore, we propose a less restrictive policy, which allows a browser extension to communicate with other origins out of the loaded web page but prevent possible leakage of sensitive information. Without control, a malicious browser extension can read browser information such as the current document URL, the browsing history, and data from the web page that is either loaded from the hosting web server or typed by the user. This *second* policy protects sensitive browser information as follows: a browser extension can read any information from browser objects and origins in a web session, but after it reads any sensitive information from the browser such as URLs, browsing history, cookies, and sensitive web form data, it cannot send the data to any origin outside of the web page, including saving them to a file and sending them to the network.

Protecting Sensitive Web Data The second policy is effective in preventing sensitive data leakage, including those maintained by a browser such as user browsing history, search keywords, cookies, and sensitive web data. There are, however, many existing BHOs or toolbars that work with some browser information. For example, Google and Yahoo toolbars collect user browsing history and search keywords and do statistic analysis to improve their search services. Also, many toolbars help users manage bookmarks on a dedicated web site. Therefore, blocking any released browser information affects the benign behavior of these widely used browser extensions.

For this purpose, we deploy a *third* policy which is further relaxed from the second one. Under this policy, a browser extension can read usual browser information such as URLs and bookmarks and send it outside of the sandbox, upon possible user authorization if necessary. However, a browser extension cannot leak sensitive web data to the outside at any time. Usually, a web page can have both sensitive and non-sensitive data, which are represented in the DOM tree and identified by tagName such as `password`. Accessing (both reading and writing) non-sensitive data should always be allowed. Under this policy, a browser extension plugin is allowed to read sensitive data, but any attempt to modify or leak sensitive data out of the sandbox is forbidden including saving it into the file system or sending it to the outside over network.

6 sePlugin Evaluation

We conduct a set of experiments to evaluate detection rates on malicious plugins, false positive rates on benign plugins, and its performance overhead. For these experiments, we run sePlugin-enhanced IE8.0 on Windows XP Professional on a machine with an Intel 2.79 GHz CPU and 2 GB RAM.

6.1 Detecting Malicious Plugins

We collected a number of malicious IE plugin samples and in this paper, we use sample names that are used by Symantec. In this set of experiments, over 40 samples have been tested, which fall into 16 different types of malicious plugins as shown in Table 1. For clarity, we show only one example for each type in the table. Regarding malicious .NET-based plugins, we find it difficult to obtain

Table 1. Malicious Plugins Detection with P1 (Relaxed Same Origin Policy), P2 (Protecting Sensitive Browser Information) and P3 (Protecting Sensitive Web Data)

name	format	type	main behaviors	P1	P2	P3
Adware.CPush	BHO	adware	checking update and display advertisement	✓	✗	✗
Adware.CWSIEFeats	BHO	adware	change IE default page, download file and display advertisement	✓	✓	✓
Internet Optimizer	BHO	adware	redirect navigation	✓	✓	✗
Adware.Rugo	BHO	adware	display advertisement and information leak	✓	✓	✓
Trojan.Vundo	BHO	spyware	download file and display advertisement	✓	✗	✗
Trojan.Linkoptimizer	BHO	adware	display advertisement, block connection	✓	✓	✓
Spyware.CWSMi	BHO	spyware	leak browse history	✓	✓	✓
Adware.Begin2search	toolbar	adware	display advertisement and download file	✓	✓	✗
Trojan. Elitebar	toolbar	spyware	redirect use search, change IE home page and access system files	✓	✓	✓
Spyware. Dotcomtoolbar	toolbar	spyware	forward HTTP requests to predetermined web	✓	✓	✓
Spyware.ISearch	toolbar	spyware	track user activities and forward search	✓	✓	✗
Adware.Mirar	toolbar	spyware	periodically collect and leak user activities and display advertisement	✓	✓	✓
Adware.IEDriver	toolbar	spyware	change home page and display advertisement	✓	✓	✓
Spyware.IEToolbar	toolbar	spyware	leak user activities and display advertisement	✓	✓	✗
888bar	toolbar	spyware	change browser configuration and display advertisement	✓	✓	✓
Adware.MaxSearch	toolbar	spyware	hijack search and change registry	✓	✓	✓
.NET plugin	BHO	spyware	search sensitive user information	✓	✓	✓

real-world instances for testing. To circumvent the problem, we slightly modify a popular BHO with C# source code [6]. This BHO reads passwords from a web page. In our experiment, we observe through sePlugin-enhanced IE8 that two actions have been performed by this plugin: (1) read sensitive data with tagName *password* through a COM interface from the web; (2) send sensitive data through a system call outside of the browser.

In the experiments, we test the three security policies that provide different levels of security protection. The evaluation results are illustrated in Table 1. From the table, we note that under the strictest policy (i.e., *Relaxed Same Origin Policy*), sePlugin successfully detects all malicious plugins. Under the policy of *Protecting Sensitive Browser Information*, sePlugin-enhanced IE8 achieves a 88.2% detection rate. If the most relaxed policy (i.e., *Protecting Sensitive Web Data*) is applied, sePlugin-enhanced IE8 detects 64.7% of the malicious plugins. On the other side, we find that although neither of the *Protecting Sensitive Web Data* and the *Protecting Sensitive Browser Information* policies achieves a 100% detection rate, those undetected malicious plugins actually do not leak sensitive user information out of the browser. Moreover, the malicious .NET-based plugin can be detected by each of the three policies.

6.2 False Positive Tests

We also conduct experiments to test 18 benign plugins to see whether sePlugin will raise false alarms. These benign plugins include 6 toolbars and BHO from renowned sources like Google, Yahoo! and Microsoft, and other popular plugins like spybot, T-Online toolbar, Airoboform, Microgarden, LostGoggles and Security Software Search Bar. The false positive rates of sePlugin are shown in round 1 of Table 2.

Table 2. False Positive Rate

false positive rate	Relaxed SOP	Protecting Sensitive Browser Information	Protecting Sensitive Web Data
round 1	11.1%	5.5%	5.5%
round 2	22.2%	16.7%	16.7%
round 3	5.5%	0%	0%

Some benign plugins also collect user or browser information. For example, some Yahoo! toolbars allow users to share information with Yahoo! about the sites a user has visited to improve Yahoo!'s services. This feature requires the user's consent. To analyze its security threat, we repeat the experiments by enabling the information sharing feature and the results are shown in round 2 of Table 2. Clearly, the results show that if the user agrees to share his browsing information, sePlugin raises more security alarms. This implies that malicious and benign plugins may adopt similar technologies and sometimes behave similarly. Their critical difference lies in their intentions.

To reduce the false positive rate, we improve the security policy by labeling communications with some well known web sites such as Yahoo!, Google, Microsoft as benign. In terms of the SOP, this step would add these trusted web sites to the origin list that allows plugins to access. After this new configuration, we repeat the experiments after information sharing is enabled. From the results demonstrated in round 3 of Table 2, having a whitelist of known sources of benign plugins can greatly help reduce the false alarm rates of sePlugin. It is noted that two thirds of the benign plugins are not from the companies associated with the whitelisted websites and even for the other remaining plugins, they may visit websites other than Yahoo!, Google, and Microsoft.

6.3 Operational Overhead

To evaluate the execution overhead of sePlugin accurately, we try to minimize the impact of measurement noise. As web browsing on the real Internet introduces noise due to end-to-end data transfer delays and jitters, we perform our experiments on a local web server that hosts downloaded web page contents. As in the experiment IE has installed some benign plugins, when the web page is loaded, these plugins will be loaded as well.

To measure execution overhead of sePlugin, we have developed a watching BHO, which is installed in IE8. This watching BHO listens to all types of web browser events like *DocumentComplete* and *BeforeNavigate2*. Once these events

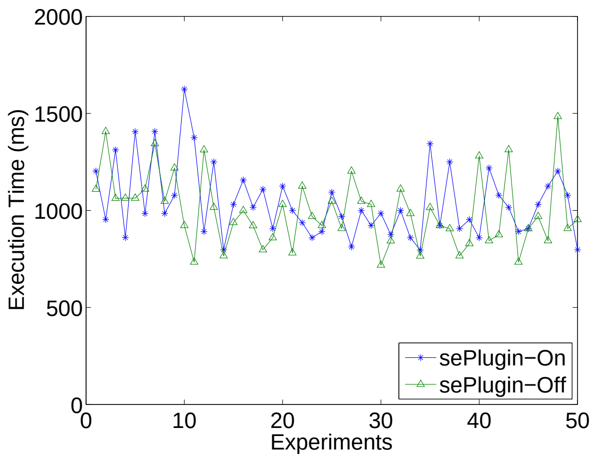

Fig. 4. Execution overhead of sePlugin

are triggered, the watching BHO records the time of these events. To evaluate sePlugin's overhead, we conduct web browsing with and without sePlugin installed in IE8. For each case (sePlugin enabled or disabled), we repeat the overhead tests for 50 times. The web loading times are illustrated in Figure 4. Comparing the average web page loading time with and without sePlugin, we find the average web page loading time with sePlugin installed is only about 4.46% longer than the time without sePlugin, which suggests that the execution of sePlugin takes only a small portion of the overall webpage loading time.

7 Conclusion

These days plugins are the main vectors for web-based attacks. While a great amount of endeavors have been made by both the research community and the industry to improve the web security, existing solutions are far from effective. We have designed and implemented sePlugin to enhance the security of plugin execution in Windows IE. The key idea is a lightweight in-process sandboxing mechanism that provides fine-grained yet flexible access control. We have deployed sePlugin in IE8.0 and the experimental results show that it enforces intended security policies with only about 4.46% execution overhead.

8 Acknowledgment

We thank the anonymous referees for providing constructive comments. The work has been supported in part by U.S. AFOSR under grant FA9550-09-1-0071, and by U.S. National Science Foundation under grant CNS-0746649.

References

1. http://www.google.com/chrome/intl/en/features.html.
2. http://www.cwsandbox.org/.
3. http://www.codeproject.com/KB/system/hooksys.aspx.
4. http://msdn.microsoft.com/en-us/library/ddk909ch%28VS.71%29.aspx.
5. http://research.microsoft.com/sn/detours/.
6. http://www.codeproject.com/KB/cs/Attach_BHO_with_C_.aspx.
7. Adsafe: Making javascript safe for advertising. http://www.adsafe.org.
8. Common intermediate language. http://en.wikipedia.org/wiki/Common_Intermediate_Language.
9. Component object model. http://msdn.microsoft.com/en-us/library/ee663262%28VS.85%29.aspx.
10. Dynamicproxy. http://www.castleproject.org/dynamicproxy/index.html.
11. Earthlink and webroot release second spyaudit report. http://www.earthlink.net/about/press/pr_spyAuditReport/.
12. Firefox add-ons. https://addons.mozilla.org/en-US/firefox/.
13. Internet explorer architecture. http://msdn.microsoft.com/en-us/library/aa741312%28VS.85%29.aspx.
14. Same origin policy. http://en.wikipedia.org/wiki/Same_origin_policy.
15. State of spyware q2 2006, consumer report. http://www.webroot.com/resources/stateofspyware/excerpt.html.
16. Understanding the import address table. http://sandsprite.com/CodeStuff/Understanding_imports.html.
17. B. Cabral, P. Marques, and L. Silva. Rail: code instrumentation for .net. In *Proceedings of the 2005 ACM symposium on Applied computing*, 2005.
18. C. Grier, S. Tang, and S. King. Secure web browsing with the op web browser. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, 2008.
19. Aaron Hackworth. Spyware. us-cert publication, 2005.
20. Aaron Hackworth. Known vulnerabilities in mozilla products. http://www.mozilla.org/projects/security/known-vulnerabilities.html, 2006.
21. E. Kirda, C. Kruegel, G. Banks, G. Vigna, and R. Kemmerer. Behavior-based spyware detection. In *Proceedings of 15th USENIX Security Symposium*, August 2006.
22. Z. Li, X. Wang, and J. Choi. Spyshield: Preserving privacy from spy add-ons. In *Proceedings of the 10th International Symposium, RAID*, 2007.
23. M. Louw, J. Lim, and V.N. Venkatakrishnan. Extensible web browser security. In *Proceedings of DIMVA'07*, 2007.
24. H. Wang, C. Grier, A. Moshchuk, S. King, P. Choudhury, and H. Venter. The multi-principal os construction of the gazelle web browser. In *Proceedings of USENIX Security'09*, 2009.
25. Y. Wang, R. Roussev, C. Verbowski, A. Johnson, M. Wu, Y. Huang, and S. Kuo. Gatekeeper: Monitoring auto-start extensibility points(aseps) for spyware management. In *Proceedings of Large Installation System Administration Conference (LISA) USENIX*, November 2004.
26. B. Yee, D. Sehr, G. Dardyk, B. 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 IEEE Symposium on Security and Privacy*, 2009.