DIGGING DEEP INTO THE FLASH SANDBOXES

Paul Sabanal
IBM X-Force Advanced Research
tsabanpm[at]ph.ibm.com, pv.sabanal[at]gmail.com
@polsab

Mark Vincent Yason
IBM X-Force Advanced Research
yasonmg[at]ph.ibm.com
@MarkYason

ABSTRACT

Lately we have seen how sandboxing technology is positively altering the software security landscape. From the Chrome browser, to Adobe Reader, to Mac and iOS applications, sandboxing has become one of the main exploit mitigation technologies that software has come to rely on. As with all critical security technologies, they need to be understood and scrutinized, mainly to see how effective they are, or at the very least, to satisfy one's curiosity. The sandbox implementations for Adobe's Flash Player certainly piqued ours.

Our talk will explore the internals of three sandbox implementations for Flash: Protected Mode Flash for Chrome, Protected Mode Flash for Firefox, and Pepper Flash. And of course, we will show that an exhaustive exploration of the Flash sandboxes will eventually yield gold as we discuss and demonstrate some Flash sandbox escape vulnerabilities we found along the way.

We start with a look at the high level architecture of each sandbox implementation. Here we will define the role of each process and the connections between them. In the second part, we will dive deep into the internal sandbox mechanisms at work such as the sandbox restrictions, the different IPC protocols in use, the services exposed by higher-privileged processes, and more. In the third part of our talk we will take a look at each sandbox's security and talk about the current limitations and weaknesses of each implementation. We will then discuss possible avenues to achieve a sandbox bypass or escape. Throughout all this we will be pointing out the various differences between these implementations.

1. CONTENTS

2. INTRODUCTION

During Black Hat USA last year, we gave a talk about Adobe Reader X's sandbox. In that talk we covered the sandbox implementation of one of the primary exploitation vectors used by malware. We also noted that ever since the Reader X sandbox's introduction there has been a remarkable decrease in PDF exploits released in the wild, and thankfully, this remains true up to this time. This year, we focus our sights on another popular exploitation vector - Adobe's Flash Player, and this time, we have three implementations of the sandbox to play with.

In doing this research, we asked ourselves the same things we did last year. What are the security implications with this new technology and what other things can an attacker do in spite of the restrictions imposed by the sandbox? What can still be done within these limits that, from an attacker's perspective, would still bring profit, or from a user's perspective, should be watched out for? Since we are investigating three different Flash sandbox implementations, we also asked ourselves how these implementations differ from each other.

To answer these questions, we dived deeply into the internals of the three Flash sandbox implementations. This paper documents our findings and discusses the internal mechanisms, limitations, and potential escape avenues for each sandbox implementation. We will also provide our thoughts and recommendations on the matter of sandbox security.

3. THE TARGETS

In this paper, we will discuss three different implementations of the Flash Player Sandbox. The targets are:

1. Flash Player Protected Mode For Firefox
2. Flash Player Protected Mode For Chrome
3. Flash Player Protected Mode For Chrome Pepper

Throughout this paper we will refer to them as Firefox Flash, Chrome Flash, and Pepper Flash, respectively.

Firefox Flash, an NPAPI [1] plugin, was first released as a beta on February 2012, and was officially released in June 2012. It is developed by Adobe in collaboration with Mozilla. It is based on the sandboxing code in Adobe Reader X, which we covered in our talk and paper [2] at Black Hat USA last year. Hence, there will be a lot of similarities between them. We will be using version 11.3.300.257 in this paper.

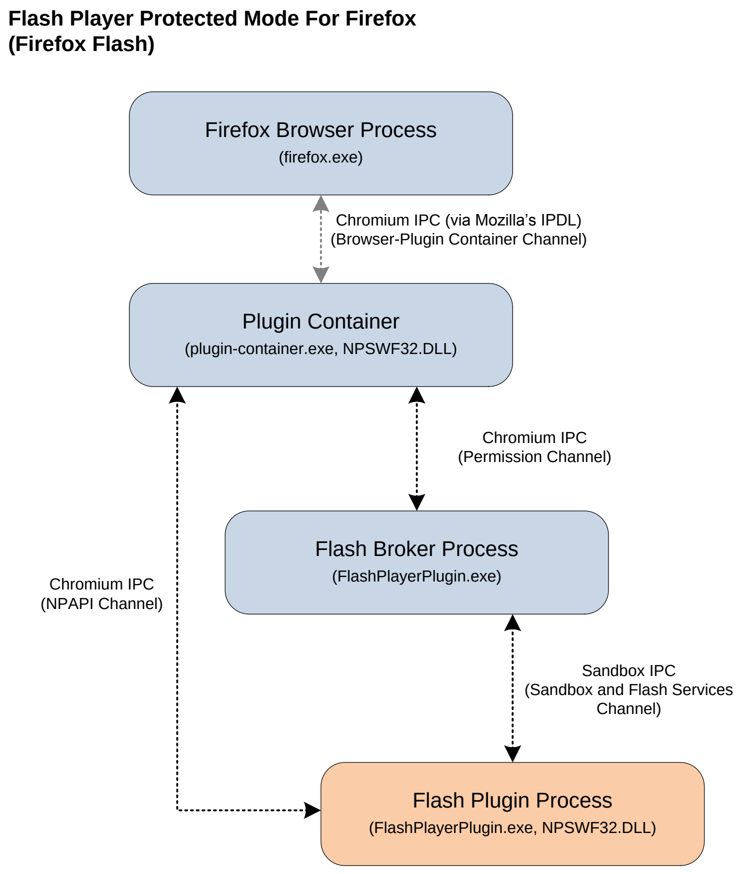
Chrome Flash, also an NPAPI plugin, has been around since December 2010 and is a result of collaboration between Adobe and Google. It is the default Flash player in Chrome. We will be using the version bundled with Chrome 20.0.1132.47 in this paper.

Pepper Flash is an implementation of Flash player using Google's Pepper Plugin API (PPAPI) [3]. It can be enabled through Chrome > Settings > Privacy > Content Settings > Plugins. The version covered in this paper is bundled with Chrome 20.0.1132.47 and is an experimental version. At the time of writing, Chrome Beta 21 has been released which includes Pepper Flash as the default Flash Player.

4. SANDBOX ARCHITECTURE

This section discusses the general architecture of each of the sandbox implementations. More details will be provided in the subsequent sections.

4.1. FLASH PLAYER PROTECTED MODE FOR FIREFOX


Flash Player Protected Mode For Firefox (Firefox Flash)
Firefox Browser Process (firefox.exe)
Chromium IPC (via Mozilla's IPDL) (Browser-Plugin Container Channel)
Plugin Container (plugin-container.exe, NPSWF32.DLL)
Chromium IPC (Permission Channel)
Flash Broker Process (FlashPlayerPlugin.exe)
Chromium IPC (NPAPI Channel)
Sandbox IPC (Sandbox and Flash Services Channel)
Flash Plugin Process (FlashPlayerPlugin.exe, NPSWF32.DLL)

The Flash Player Protected Mode for Firefox (Firefox Flash) sandbox configuration consists of the following components:

- Firefox Browser Process (*firefox.exe*) – The main Firefox browser process (*firefox.exe*). It launches *plugin_container.exe* when a web page with Flash content is opened.

- Plugin Container (*plugin_container.exe*) – facilitates communication between the Flash plugin process and the Firefox browser process. It is also responsible for launching the broker process.

- Flash Broker Process (*FlashPlayerPlugin.exe*) – spawned by *plugin_container.exe*. It is responsible for setting up the sandbox restriction and policies, and also for spawning the sandbox process. It also hosts an IPC service to communicate with the sandbox process.

- Flash Plugin Process (*FlashPlayerPlugin.exe*) – The sandboxed Flash plugin process. It is responsible for parsing and rendering Flash content.

The Firefox Flash sandbox is enabled by default but it can be disabled using the privacy and security configuration file for Firefox Flash [4]. This file, *mms.cfg*, should be placed in the following folder:

- *%WINDIR\System32\Macromed\Flash* for 32-bit Windows or
- *%WINDIR\SysWow64\Macromed\Flash* for 64-bit Windows

To disable the protected mode, set the *ProtectedMode* option in *mms.cfg* to 0:

```
ProtectedMode = 0
```

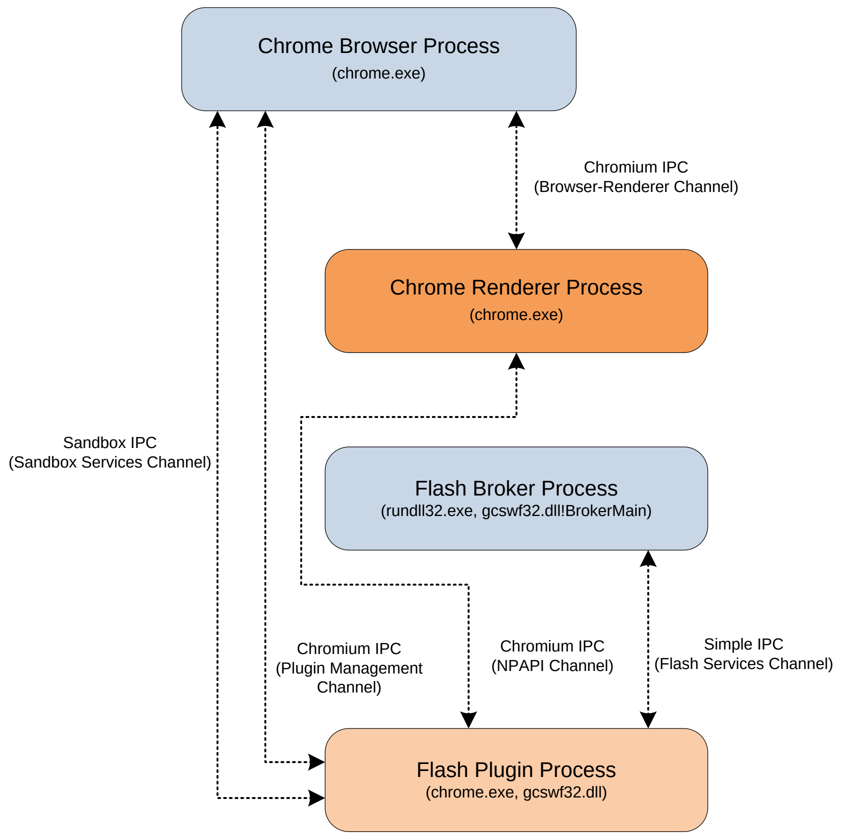
mms.cfg is also used to set up a policy file, which contains whitelist policies to bypass some of the default restrictions in the sandbox. To enable the whitelist policy file, the *ProtectedModeBrokerWhitelistConfigFile* option should be set to 1:

```
ProtectedModeBrokerWhitelistConfigFile = 1
```

A policy file with the file name *ProtectedModeWhitelistConfig.txt* should be placed in *%WINDIR%\System32\Macromed\Flash* for 32-bit Windows and *%WINDIR%\SysWow64\Macromed\Flash* for 64-bit Windows.

4.2. FLASH PLAYER PROTECTED MODE FOR CHROME

Flash Player Protected Mode For Chrome (Chrome Flash)



The Flash Player Protected Mode for Chrome (Chrome Flash) sandbox configuration consists of the following components:

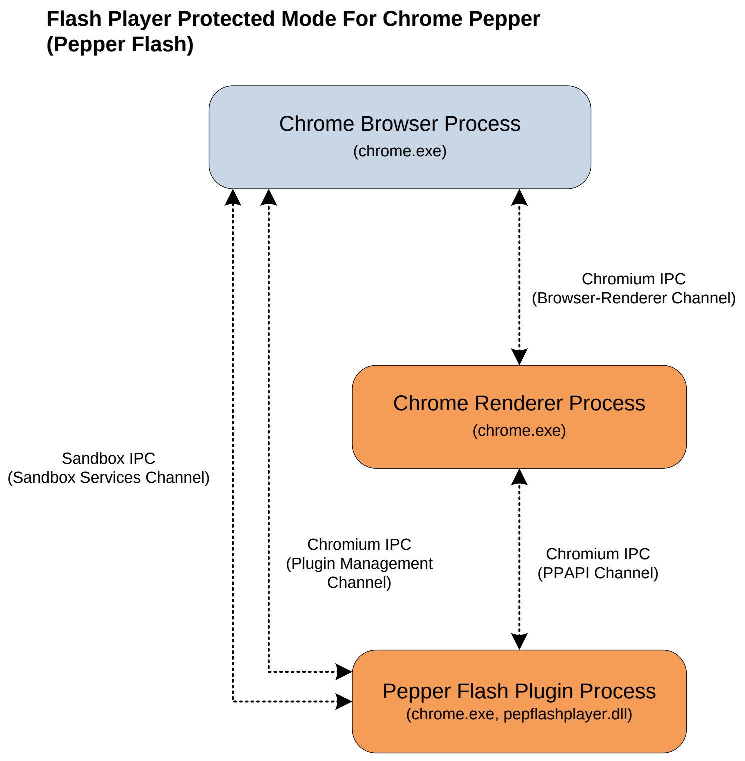
- Chrome Browser Process (*chrome.exe*) – The main Chrome browser process. It launches the Flash broker process and the Flash plugin process when a web page with Flash content is opened. It also exposes some browser-related services that the Flash plugin process connects to.

- Chrome Renderer Process (*chrome.exe*) – The renderer process for the page the Flash content is in. It also exposes browser-related services that the Flash plugin process connects to.

- Flash Broker Process (*rundll32.exe, gcswf32.dll!BrokerMain*) – *rundll32.exe* is used to run *gcswf32.dll*'s *BrokerMain* entry point, which act as the broker process. It also hosts Flash specific services for the Flash plugin process.

- Flash Plugin Process (*chrome.exe, gcswf32.dll*) – The sandboxed Flash plugin process. It is responsible for parsing and rendering Flash content.

4.3. FLASH PLAYER PROTECTED MODE FOR CHROME PEPPER


Flash Player Protected Mode For Chrome Pepper (Pepper Flash)
Chrome Browser Process
(chrome.exe)
Chromium IPC
(Browser-Renderer Channel)
Chrome Renderer Process
(chrome.exe)
Sandbox IPC
(Sandbox Services Channel)
Chromium IPC
(Plugin Management Channel)
Chromium IPC
(PPAPI Channel)
Pepper Flash Plugin Process
(chrome.exe, pepflashplayer.dll)

Flash Player Protected Mode for Chrome Pepper (Pepper Flash) sandbox configuration consists of the following components:

- Chrome Browser Process (*chrome.exe*) – The main Chrome browser process. It launches the Flash plugin process when a web page with Flash content is opened. It also exposes some browser-related services that the Flash plugin process connects to.

- Chrome Renderer Process (*chrome.exe*) – The renderer process for the page the Flash content is in. It also exposes browser-related and Pepper services that the Flash plugin process connects to.

- Flash Plugin Process (*chrome.exe, pepflashplayer.dll*) – The sandboxed Flash plugin process. It is responsible for parsing and rendering Flash content

5. SANDBOX MECHANISMS

After discussing the architecture of each Flash sandbox implementation, we will now dive deep into the internal mechanisms used by each Flash sandbox implementation. In this section, we will start with the discussion of the mechanisms used for sandboxing the Flash plugin process and then progressively move the discussion to the mechanisms used by the higher-privileged processes.

5.1. SANDBOX STARTUP SEQUENCE

In this section, we will discuss the steps each sandbox implementation takes when starting up.

5.1.1. FIREFOX FLASH

1. When a web page with Flash content is opened, *plugin_container.exe* is spawned.

2. *plugin_container.exe* then spawns the broker process *FlashPlayerPlugin_11_3_300_257.exe*.

3. The broker process sets up the sandbox restrictions for the sandbox process:
   a. Sets the job level to *JOB_RESTRICTED*, but with the following restrictions unset:
      - *JOB_OBJECT_UILIMIT_READCLIPBOARD*
      - *JOB_OBJECT_UILIMIT_WRITECLIPBOARD*
      - *JOB_OBJECT_UILIMIT_GLOBALATOMS*

   b. Sets the token level. It sets up two tokens, the initial token and the lockdown token. Both tokens will be active when the sandbox process is started. The sandbox process requires a more privileged token during startup, as it needs to access resources that are otherwise inaccessible due to the sandbox. The initial token allows the sandbox process to temporarily have an elevated privilege. It is only valid for the initial thread the process started with and will be discarded later. Other threads will only be using the less privileged lockdown token. The token levels assigned to each tokens are:
      - Initial token – *USER_RESTRICTED_SAME_ACCESS* for Vista or later, otherwise *USER_UNPROTECTED*
      - Lockdown token – *USER_LIMITED*

      Refer to section 5.2.1 for more details about the token restrictions.

   c. Sets the integrity level. It will be set to *INTEGRITY_LEVEL_LOW*.

   d. Adds a DLL eviction policy, which lists DLLs that are suspected or known to cause a sandboxed process to crash. These DLLs will be unloaded by the sandbox. Refer to section 11.1 for the list of evicted DLLs in Flash Player Protected Mode for Firefox.

4. The broker process sets up the sandbox policies, which are rules that describe exceptions from the restrictions imposed by the sandbox policy.

   a. Sets up admin-configurable policies. The policy rules are read from a file named *ProtectedModeWhiteList.txt* located in *%WINDIR%\System32\Macromed\Flash* for 32-bit Windows, or *%WINDIR%\SysWow64\Macromed\Flash* for 64-bit Windows.

   b. Sets up hard-coded policies for file, named pipes, process, registry, sync objects, mutant, and section access.

5. The broker process spawns the sandbox process in a suspended state. It will run the *FlashPlayerPlugin_11_3_300_257.exe* executable, the same as the broker, but with the "-*type=renderer*" parameter.

6. Set up and initialize interceptions on the sandbox process. Refer to section 5.3 for more details about the interceptions.

7. Resume execution of the sandbox process.

5.1.2. CHROME FLASH

1. When a web page with Flash content is opened, the Chrome browser process spawns *rundll32.exe* to launch the broker process via the *gcswf32.dll!BrokerMain* entrypoint.

2. The Chrome browser process sets up the sandbox policies for the Flash sandbox process:
   a. Sets the job level to *JOB_UNPROTECTED*

   b. Sets the token level to the following:
      - Initial token - *USER_RESTRICTED_SAME_ACCESS*
      - Lockdown token - *USER_INTERACTIVE*

3. Sets the integrity level to *INTEGRITY_LEVEL_LOW*

4. Adds a DLL eviction policy, which lists DLLs that are suspected or known to cause a renderer process to crash. These DLLs will be unloaded by the sandbox. Refer to section 11.2 for the list of evicted DLLs in Chrome.

5. Adds a plugin DLL eviction policy, which lists DLLs that are suspected or known to cause a plugin process to crash. These DLLs will be unloaded by the sandbox. Refer to section 11.3 for the list of evicted plugin DLLs in Chrome.

6. The Chrome browser process spawns the sandboxed Flash plugin process, which is chrome.exe with a "*type=plugin*" parameter and with *gcswf32.dll* loaded. This process is initially launched in a suspended state.

7. Set up and initialize interceptions on the sandbox process. Refer to section 5.3 for more details about the interceptions.

8. Resume execution of the sandboxed Flash plugin process.

5.1.3. PEPPER FLASH

Pepper plugins run under Chrome's renderer so the same restrictions apply with a minor difference.

1. When a web page with Flash content is opened, the Chrome browser process sets up the sandbox policies for the Pepper Flash plugin process:
    a. Sets the job level to *JOB_LOCKDOWN*

    b. Sets the token level to the following:
        - Initial token - *USER_RESTRICTED_SAME_ACCESS* for Vista or later, otherwise *USER_UNPROTECTED*
        - Lockdown token - *USER_LOCKDOWN*

    c. Sets the integrity level to *INTEGRITY_LEVEL_UNTRUSTED*

    d. Sets alternate window station and desktop

    e. Adds a DLL eviction policy, which lists DLLs that are suspected or known to cause a renderer process to crash. These DLLs will be unloaded by the sandbox. Refer to section 11.2 for the list of evicted DLLs in Chrome.

    f. Add policy for Pepper plugin. This simply adds full access to named pipes that match the following pattern "*\\.\pipe\chrome.**".

2. The Chrome browser process spawns the sandboxed Pepper Flash plugin process, which is *chrome.exe* with a "*type=ppapi*" parameter and with *pepflashplayer.dll* loaded. This process is initially launched in a suspended state.

3. Set up and initialize interceptions on the sandbox process. Refer to section 5.3 for more details about the interceptions.

4. Resume execution of the sandbox process.

5.2. SANDBOX RESTRICTIONS

Sandbox restrictions are the mechanisms in place to run the sandboxed Flash plugin process in a confined environment. In case the sandboxed Flash plugin is compromised, sandbox restrictions will prevent malicious code from making persistent changes to the system, and depending on the sandbox restrictions and sandbox policies in place, will also prevent malicious code from accessing confidential information from the system.

The sandbox restrictions in all Flash sandbox implementations are based on the Practical Windows Sandboxing recipe [5, 6, 7] which describes the use of the following Windows mechanisms for restricting the privileges and capabilities of a sandboxed process:

- Restricted Tokens
- Integrity Levels
- Job Objects

- Alternate Desktop and Alternate Window Station

5.2.1. RESTRICTED TOKENS

One of the fundamental ways to lower the privileges of a sandboxed process is by assigning it a restricted token [8]. In the case of Flash, the restricted token assigned to the sandboxed Flash plugin process is derived from the user's token and has the following restrictions set:

- Deny-Only Security Identifiers (SIDs) - Only pre-selected SIDs are left enabled, all other SIDs are set to deny-only. This limits the number of securable resources the sandboxed Flash plugin can access and the type of access it can be granted.
- Restricting SIDs - Adding pre-selected SIDs as restricting SIDs. This ensures that the sandboxed Flash plugin can only access securable resources which are also accessible to the pre-selected restricting SIDs.
- Limited Privileges – Enabling only a very limited number of privileges so that the sandboxed Flash plugin process is limited to the types of system operations it can perform (e.g. shutting down the system and debugging programs).

5.2.2. INTEGRITY LEVELS

A *Low* or *Untrusted* integrity level [9] is also set in the token assigned to the sandboxed Flash plugin process so that write access to most securable resources will be denied since the majority of securable resources in a Windows system are assigned a *Medium* or a higher integrity level. This also mitigates *shatter attacks* [10] as lower-integrity processes are prevented from sending write-type messages to windows owned by higher-integrity processes.

5.2.3. JOB OBJECTS

Additional restrictions are also enforced to the sandboxed Flash plugin process by associating it with a job object [11]. Examples of capabilities that can be restricted via job objects are access to the clipboard, modification to system settings and preventing the spawning of additional processes.

5.2.4. ALTERNATE WINDOW STATION AND ALTERNATE DESKTOP

By assigning the sandboxed Flash plugin process a separate window station and a separate desktop, it is isolated from windows in other desktops, and the clipboard and global atom table in other window stations - all of which are vectors for sandbox escape and/or information disclosure. Note that this is only effective if the token assigned to the sandboxed process is set up so that the sandboxed process does not have access to other window stations and desktops.

5.2.5. SANDBOX RESTRICTIONS COMPARISON TABLE

The comparison table below shows the sandbox restrictions in place in Chrome Flash, Firefox Flash and Pepper Flash.

Restriction	Chrome Flash	Firefox Flash	Pepper Flash
Integrity Level	Low	Low	Untrusted
Restricted Token: Enabled SIDs (Deny-Only SIDs Exceptions)	• User's SID • Logon SID • Everyone • Users	• User's SID • Logon SID • Everyone • Users	• Logon SID

	• INTERACTIVE • Authenticated Users	• INTERACTIVE	
Restricted Token: Restricting SIDs	• Logon SID • Everyone • RESTRICTED • Users • User's SID	• Logon SID • Everyone • RESTRICTED • Users	• NULL SID
Restricted Token: Enabled Privileges	• Bypass traverse checking	• Bypass traverse checking	(None)
Job Restrictions	• Kill on job close	• Kill on job close • Spawning additional processes • Desktop creation and switching via *CreateDesktop()* and *SwitchDesktop()* • Modifying display settings via *ChangeDisplaySettings()* • Logging off, shutting down or restarting the system via *ExitWindows()* or *ExitWindowsEx()* • Using USER handles owned by processes not associated with the job • Changing system settings via *SystemParametersInfo()*	• Kill on job close • Spawning additional processes • Desktop creation and switching via *CreateDesktop()* and *SwitchDesktop()* • Modifying display settings via *ChangeDisplaySettings()* • Logging off, shutting down or restarting the system via *ExitWindows()* or *ExitWindowsEx()* • Using USER handles owned by processes not associated with the job • Changing system settings via *SystemParametersInfo()* • Kill on unhandled exception • Read from the clipboard • Write to the clipboard • Accessing global atoms
Alternate Window Station and Alternate Desktop	No	No	Yes

Based on the table above, Pepper Flash has the most sandbox restrictions in place while Chrome Flash has the least restrictions in place. Pepper Flash is also the only sandboxed Flash running as an *Untrusted* integrity process.

The restricted token assigned to Chrome Flash can still obtain read access to resources accessible to the *Users* group, the *Everyone* group, and any resource accessible via the user's SID, such as private user files located under the user's *Documents* folder. On the other hand, the restricted token assigned to Firefox Flash can also be used to obtain read access to resources accessible to the *Users* group and the *Everyone* group but it cannot be used to access resources that accessible via the user's SID. The restricted token assigned to Pepper Flash is the most restrictive and very limited to what resources it can access; it cannot even access resources accessible to the *Users* and the *Everyone* group.

Lastly, Pepper Flash is the only sandboxed Flash that uses a separate window station and a separate desktop. Similar to Adobe Reader X, Firefox Flash mitigates attacks relating to shared desktop use (such as *shatter attacks* and DLL injection via *SetWindowsHookEx()*) via the other sandbox restrictions [12], primarily, via the USER handles (*UILIMIT_HANDLES*) job restriction and by running as a *Low* integrity process.

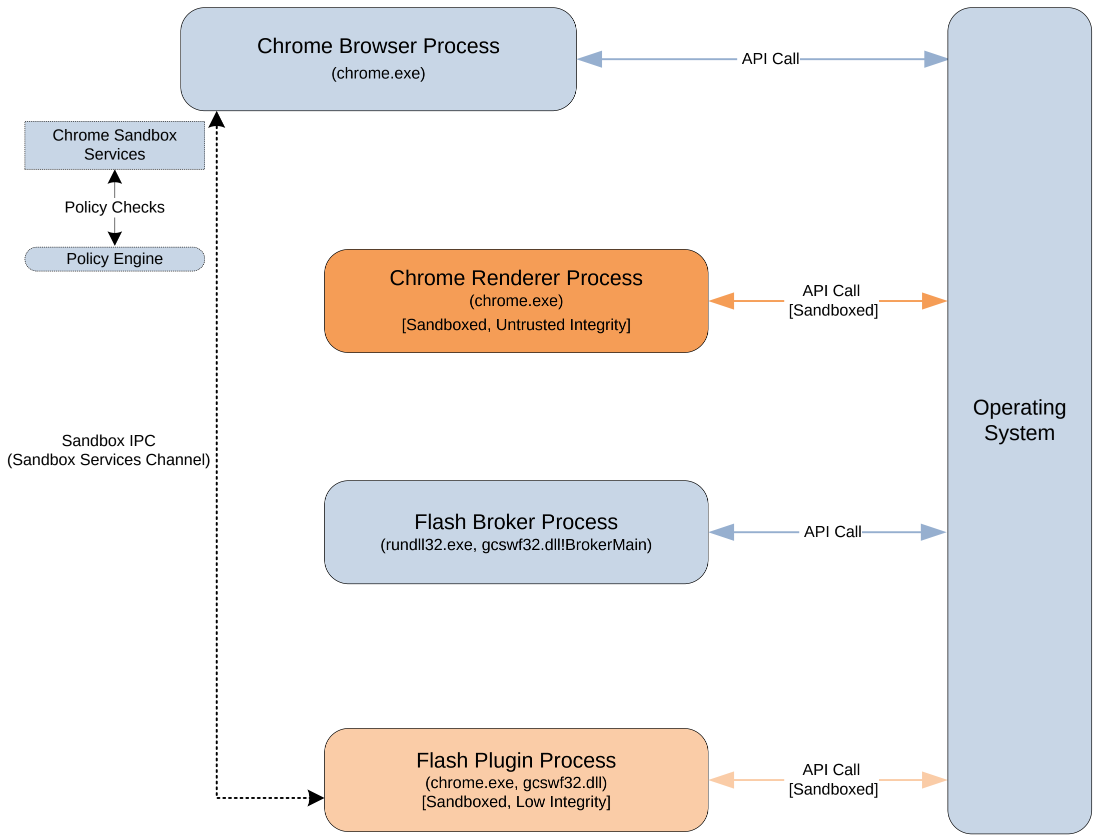
5.3. INTERCEPTION MANAGER

The purpose of the Interception Manager is to transparently forward API calls made by the sandboxed Flash plugin process to a higher-privileged process such as a browser or a broker process.

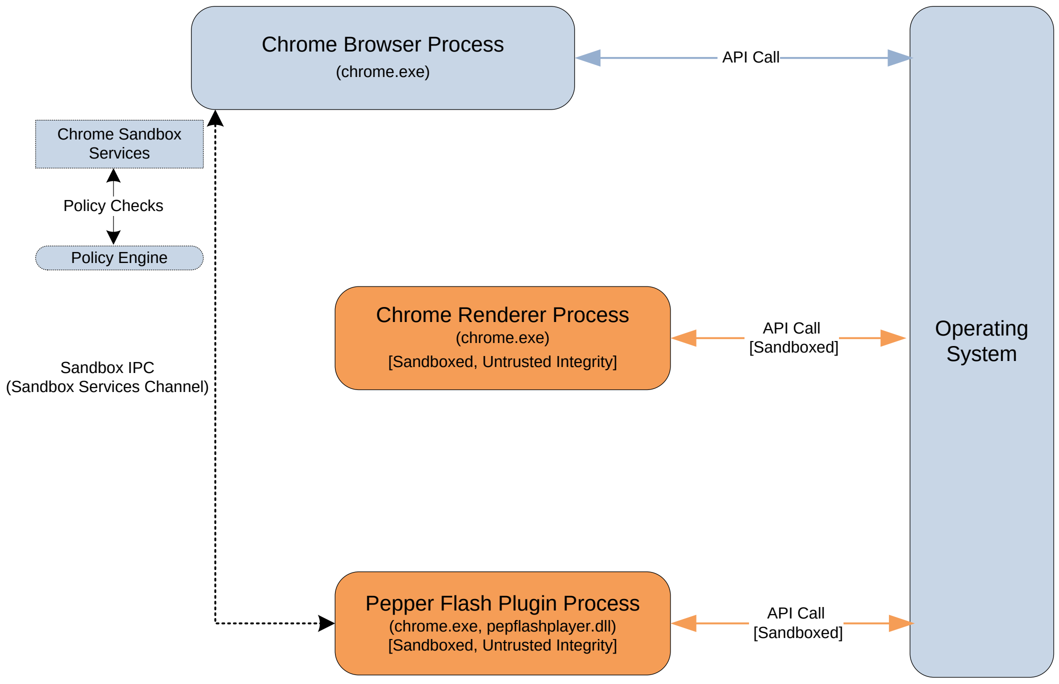
Generally, an API call is forwarded to the broker/browser process because the API call failed due to the sandbox restrictions in place. The broker/browser process on the other hand, evaluates the request against the sandbox policies and decides if the call will fail or succeed. In the latter case, the broker/browser process will generally perform the API call on behalf of the sandboxed process. An API call may also be automatically forwarded to the broker process because it just needs to be executed in the context of the higher-privileged process.

For Chrome Flash and Pepper Flash, the API calls are forwarded from the sandboxed Flash plugin process to the Chrome browser process via a Sandbox IPC connection (discussed in section 5.4.1) and are serviced by the Chrome Sandbox services hosted in the Chrome browser process (discussed in section 5.5.1):

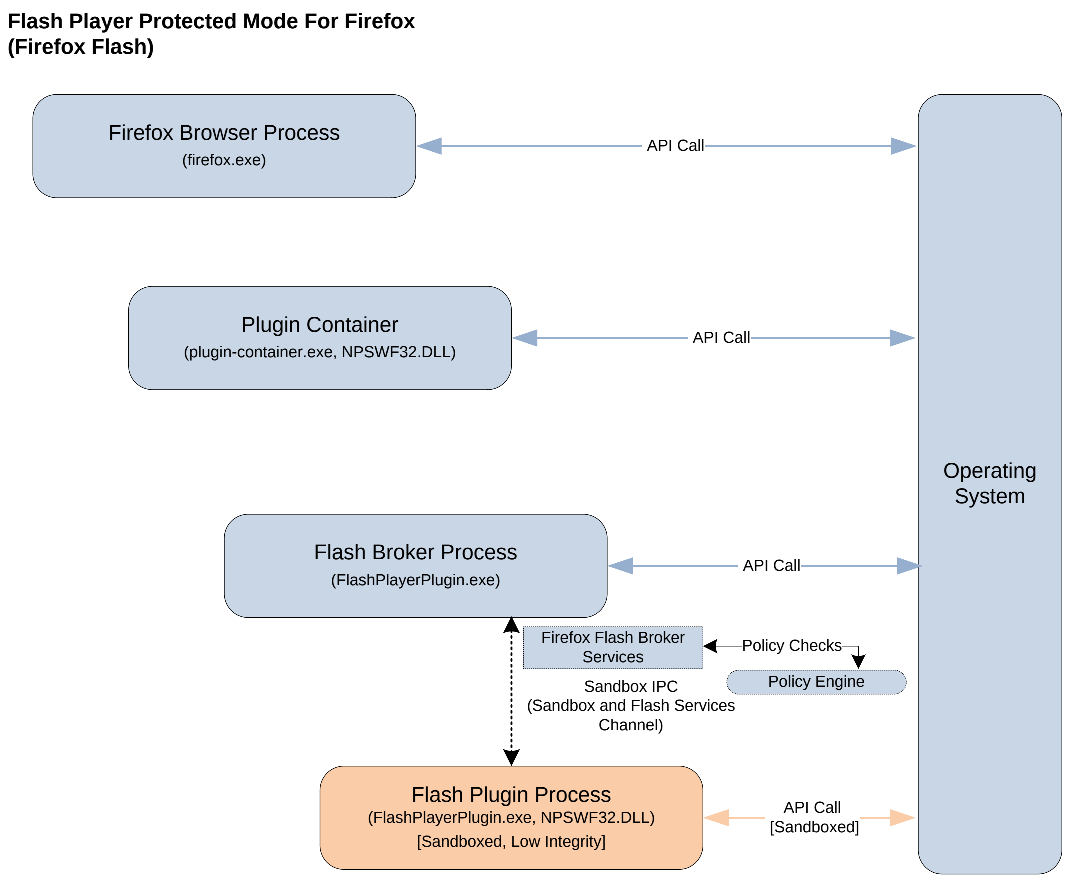
Flash Player Protected Mode For Chrome (Chrome Flash)



Flash Player Protected Mode For Chrome Pepper (Pepper Flash)



For Firefox Flash, the API calls are forwarded from the sandboxed Flash plugin process to the Flash broker process via a Sandbox IPC connection and are serviced by the Firefox Flash broker services hosted in the Flash broker process (discussed in 5.5.5):


Flash Player Protected Mode For Firefox (Firefox Flash)
Firefox Browser Process
(firefox.exe)
API Call
Plugin Container
(plugin-container.exe, NPSWF32.DLL)
API Call
Operating System
Flash Broker Process
(FlashPlayerPlugin.exe)
API Call
Firefox Flash Broker Services
Policy Checks
Policy Engine
Sandbox IPC
(Sandbox and Flash Services Channel)
Flash Plugin Process
(FlashPlayerPlugin.exe, NPSWF32.DLL)
[Sandboxed, Low Integrity]
API Call
[Sandboxed]

Forwarding API calls is done transparently via API interceptions (or *API hooking*) in the sandboxed process. Depending on the type of interception, the API interceptions are set early in the sandboxed process initialization or when the DLL where the API is located is mapped into the sandboxed process.

5.3.1. INTERCEPTION TYPES

The table below describes the different types of interceptions:

Interception Type	Constant Value	Description
INTERCEPTION_SERVICE_CALL	1	Interceptions of type *INTERCEPTION_SERVICE_CALL* are performed for *NTDLL* APIs. Interceptions are performed by the broker/browser process to the sandboxed process via *WriteProcessMemory()* when the sandboxed process is newly spawned but still in a suspended state. API Interceptions are done by patching the entry point of the API with a stub that starts with the following code sequence: `MOV EAX,<ServiceID>`

		`MOV EDX,<ThunkCodeAddress>`
		`JMP EDX`
		
		ThunkCodeAddress points to an allocated memory containing a thunk code that sets up the stack and for performing the actual control transfer to the interception handler.

INTERCEPTION_EAT	2	Interceptions of type *INTERCEPTION_EAT* are done by the sandboxed process to itself and are performed when the target DLL is mapped into the sandboxed process. DLL mapping is monitored via the interception of *NTDLL.DLL!NtMapViewOfSection()*. As its name suggests, interceptions of type *INTERCEPTION_EAT* are performed by patching the entry of the API in the export address table of the DLL.

INTERCEPTION_SIDESTEP	3	Interceptions of type *INTERCEPTION_SIDESTEP* are performed by the sandboxed process to itself when the target DLL is mapped into the sandboxed process.
		
		Interceptions of type *INTERCEPTION_SIDESTEP* are performed by patching the API entry point with a *JMP* instruction that transfers control to the thunk code:
		
		`JMP <ThunkCodeAddress>`
		`<original API code>`
		`<original API code>`
		`<. . .>`

INTERCEPTION_SMART_SIDESTEP	4	Appears to be currently unused, but Chrome's source code suggests that this interception type is similar to *INTERCEPTION_SIDESTEP* but with a different thunk code.
INTERCEPTION_UNLOAD_MODULE	5	This is a special interception type for DLLs which should be restricted from being loaded on the sandboxed process. Based on Chrome's source code, the DLLs that are set to be unloaded are those that are suspected or known to crash the sandboxed process.

Table 1. Interception Types. (Reference: http://src.chromium.org/viewvc/chrome/trunk/src/sandbox/src/sandbox_types.h?view=markup)

5.4. INTER-PROCESS COMMUNICATION

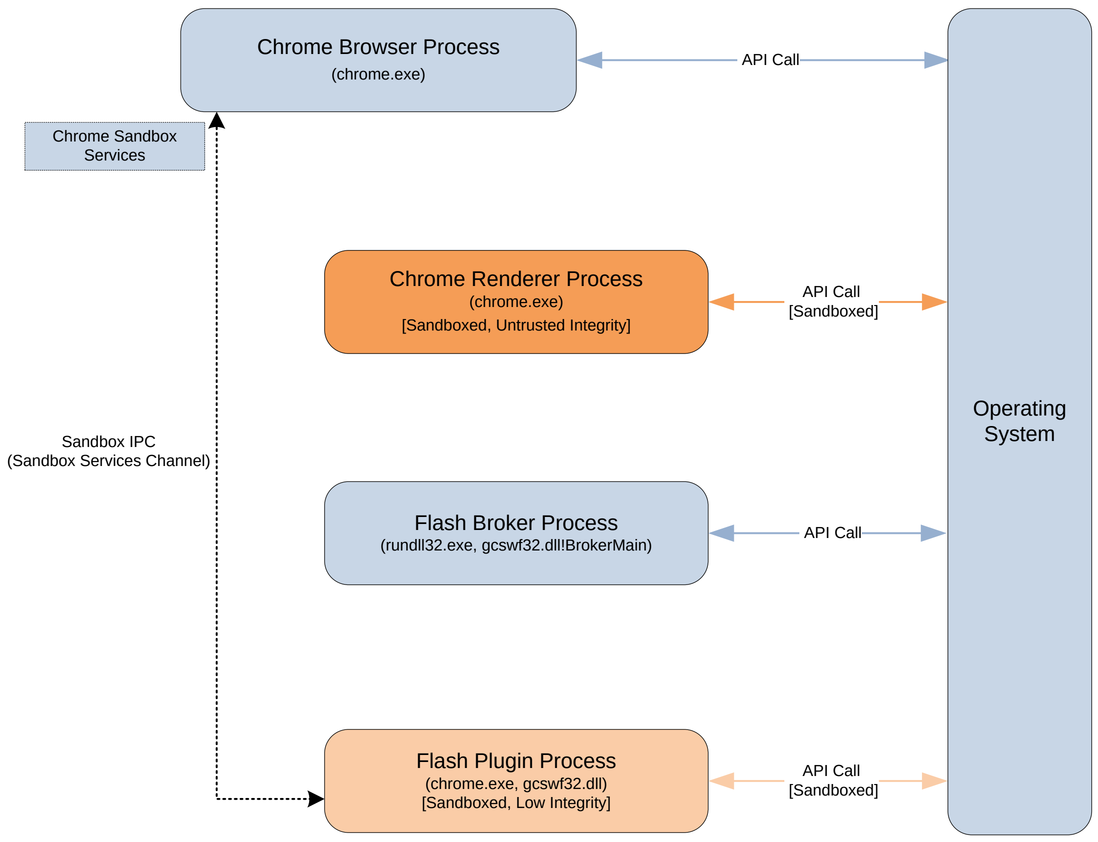
Inter-process communication (IPC) is the mechanism that allows processes to communicate with each other. In the case of the Flash sandboxes, IPC is used between processes with different privilege levels and between processes with the same privilege level. This section describes the different IPC implementations used in the Flash sandboxes.

5.4.1. SANDBOX IPC

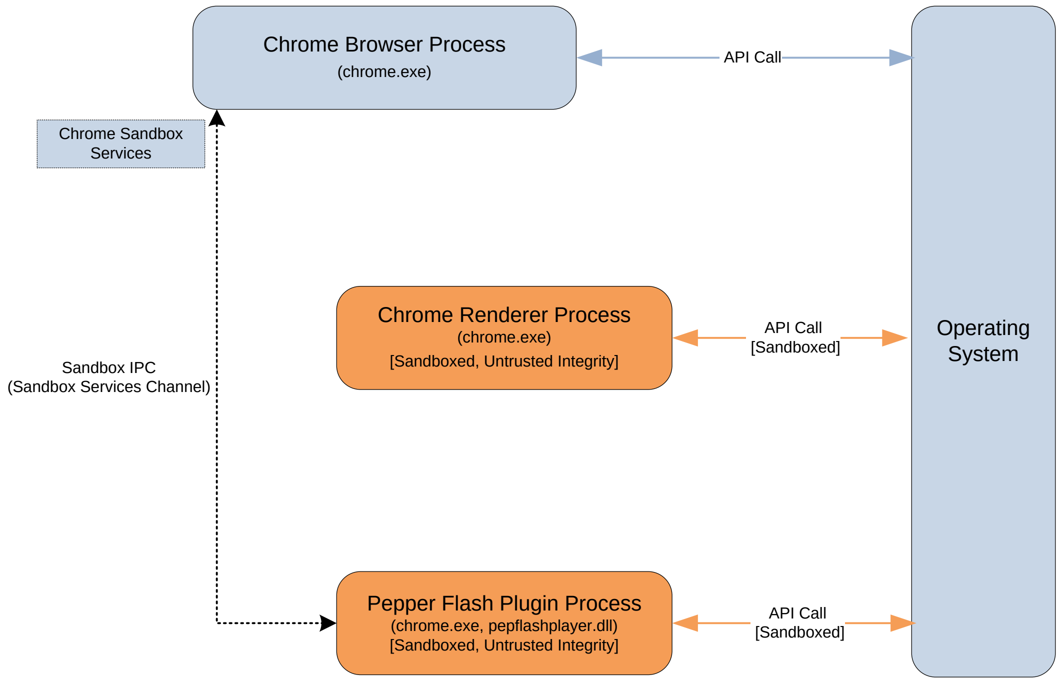
Originally from the Chromium project, all Flash sandbox implementations use the Sandbox IPC for communication between lower-privileged and higher-privileged processes.

In Chrome Flash and Pepper Flash, the Sandbox IPC is mainly used for forwarding API calls from the sandboxed Flash plugin process to the Chrome browser process:

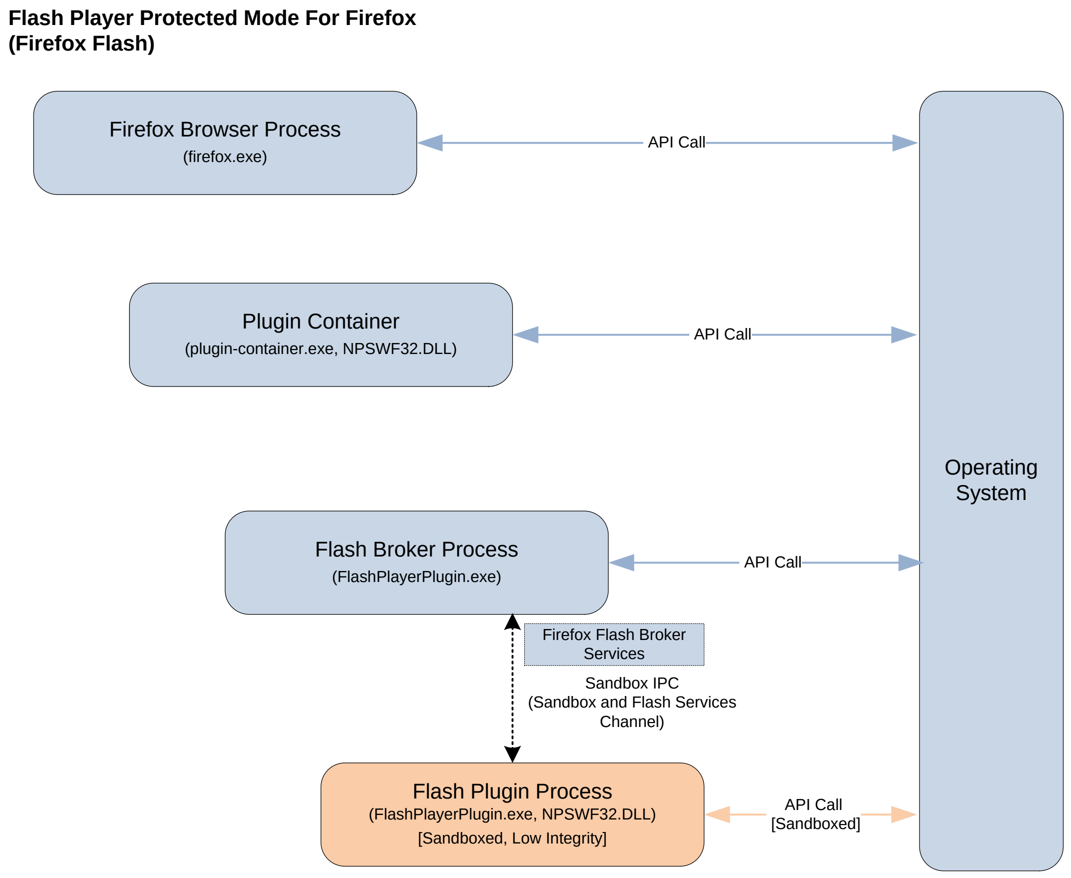
Flash Player Protected Mode For Chrome (Chrome Flash)



Flash Player Protected Mode For Chrome Pepper (Pepper Flash)



In Firefox Flash, aside from being used for forwarding API calls from the Flash plugin process to the Flash broker process, it is also used for invoking additional services, such as launching the Flash Player settings manager, which are exposed by the Flash broker process:


Flash Player Protected Mode For Firefox (Firefox Flash)
Firefox Browser Process (firefox.exe)
Plugin Container (plugin-container.exe, NPSWF32.DLL)
Flash Broker Process (FlashPlayerPlugin.exe)
Firefox Flash Broker Services
Sandbox IPC (Sandbox and Flash Services Channel)
Flash Plugin Process (FlashPlayerPlugin.exe, NPSWF32.DLL) [Sandboxed, Low Integrity]
API Call
API Call
API Call
API Call [Sandboxed]
Operating System

5.4.1.1. IMPLEMENTATION

Sandbox IPC is accomplished by shared memory and synchronization between the processes is done via Windows event objects.

To setup the Sandbox IPC connection, the Chrome browser process (in the case of Chrome Flash and Pepper Flash) or the Flash broker process (in the case of Firefox Flash) which hosts the IPC server creates a chunk of shared memory and uses the upper part of this shared memory for the Sandbox IPC. The browser/broker process then creates a duplicate of the handle to the shared memory and transfers the handle duplicate to the sandboxed Flash plugin process via a *WriteProcessMemory()* call. The IPC client hosted in the sandboxed Flash plugin process uses the transferred handle duplicate to map the shared memory to its address space.

5.4.1.2. MESSAGE STRUCTURE

IPC SHARED MEMORY STRUCTURE

The overall structure of the Sandbox IPC shared memory is consists of a top level *IPCControl* structure which is divided into channel control (*ChannelControl*) structures and channel buffer (*ActualCallParams*) structures.

The format of the *IPCControl* structure is as follows:

Offset	Size/Type	Name	Description
0x0000	0x04/size_t	channels_count	Number of IPC channels. Currently, 7 for Chrome Flash and Pepper Flash and 15 for Firefox Flash.
0x0004	0x04/HANDLE	server_alive	Mutex handle that will be used by the IPC client to

			determine if the IPC server is still alive.
0x0008	channels_count*0x14/ ChannelControl	channels	Channel control structures for *channels_count* IPC channels. Contains the control data for each channel.
?	channels_count* kIPCChannelSize / ActualCallParams	(channel buffers)	Channel buffer structures for *channel_count* IPC channels. Contains the actual serialized parameters for the IPC call. Each structure is *kIPCChannelSize* bytes each. Currently, *kIPCChannelSize* is 0x400 bytes for Chrome Flash and Pepper Flash and 0x2000 bytes for Firefox Flash.

Table 2. IPCControl Structure. (Reference: http://src.chromium.org/viewvc/chrome/trunk/src/sandbox/src/sharedmem_ipc_client.h?view=markup)

IPC CHANNEL CONTROL STRUCTURE

To allow multiple IPC connections, the IPC shared memory is divided into multiple IPC channels. Each IPC channel has its own corresponding channel control structure which contains the control data for each channel.

The format of the IPC channel control (*ChannelControl*) structure is as follows:

Offset	Size/Type	Name	Description
0x0000	0x04/size_t	channel_base	Offset (relative to the start of the IPC shared memory) of the corresponding IPC channel buffer for this channel.
0x0004	0x04/LONG	state	State of the IPC Channel: *kFreeChannel* (1), *kBusyChannel* (2), *kAckChannel* (3), *kReadyChannel* (4), kAbandonnedChannel (5).
0x0008	0x04/HANDLE	ping_event	Event handle used by the IPC client to notify the IPC server that an IPC message is ready in this channel.
0x000C	0x04/HANDLE	pong_event	Event handle used by the IPC client to receive notification from the IPC server that a response is already available in the IPC channel.
0x0010	0x04/uint32	ipc_tag	IPC Tag – a unique identifier that identifies what service the caller is requesting.

Table 3. ChannelControl Structure. (Reference: http://src.chromium.org/viewvc/chrome/trunk/src/sandbox/src/sharedmem_ipc_client.h?view=markup)

IPC CHANNEL BUFFER STRUCTURE

Each IPC channel has a corresponding IPC channel buffer which will contain the serialized IPC call parameters and IPC call results.

The format of the IPC channel buffer (*ActualCallParams*) structure is as follows:

Offset	Size /Type	Name	Description
0x0000	0x04/uint32	tag_	IPC Tag – a unique identifier that identifies what service the caller is requesting. Same value as *ChannelControl.ipc_tag*. The IPC server uses this value along with the *type_* field from the ParamInfo structures to select the handler routine that will service the request. An example IPC tag would be the value 0x03 for a request on the broker/browser process to invoke *NtCreateFile()* on behalf of the sandboxed process.
0x0004	0x04 /uint32	is_in_out_	Contains an in/out parameter.
0x0008	0x34/CrossCallReturn	call_return	IPC call result values filled out by the IPC server after servicing a request.
0x003C	0x04 /size_t	params_count_	Number of parameters.
0x0040	0x0C*(params_count+1)/ParamInfo	param_info_	Parameter information structures for *param_count_*+1 parameters.
?		parameters_	Actual parameter data. These are the serialized IPC call parameters. Example parameters are the object name and access mask used for the IPC call for *NtCreateFile()*.

Table 4. ActualCallParams Structure. (Reference: http://src.chromium.org/viewvc/chrome/trunk/src/sandbox/src/crosscall_params.h?view=markup)

The *param_info_* field is an array of *ParamInfo* structures which contains information about the type, size and offset (relative to the start of the IPC channel buffer) of each parameter. Note that *param_info_* has an extra element; the extra element is used to quickly get the total size (used part) of the IPC channel buffer because its *offset_* field points to the end of the used part of the IPC channel buffer.

The format of the *ParamInfo* structure is as follows:

Offset	Size (Type)	Name	Description
0x0000	0x04/ArgType	type_	Parameter type: *WCHAR_TYPE* (1), *ULONG_TYPE* (2), *UNISTR_TYPE* (3), *VOIDPTR_TYPE* (4), *INPTR_TYPE* (5), *INOUTPTR_TYPE* (6), *CHAR_TYPE* (7, specific to Firefox Flash), *REMOTEBUF_TYPE* (8, specific to Firefox Flash)
0x0004	0x04/ptrdiff_t	offset_	Offset of parameter data (relative to the start of the IPC channel buffer).
0x0008	0x04/size_t	size_	Size of the parameter.

Table 5. ParamInfo Structure. (Reference: http://src.chromium.org/viewvc/chrome/trunk/src/sandbox/src/crosscall_params.h?view=markup)

An interesting parameter type in Firefox Flash which is also found in Adobe Reader X [2] is *REMOTEBUF_TYPE*. *REMOTEBUF_TYPE* is a type that represents a buffer in a remote process, it has the following fields:

- Offset 0: *pid* (4 bytes) – process ID of the process where the buffer is located.
- Offset 4: *address* (4 bytes) – address of the buffer in the remote process.
- Offset 8: *size* (4 bytes) – size of the buffer in the remote process.

Return values from an IPC call are stored in the *ActualCallParams.call_return* field which in turn is a *CrossCallReturn* structure.

The format of the *CrossCallReturn* structure is as follows:

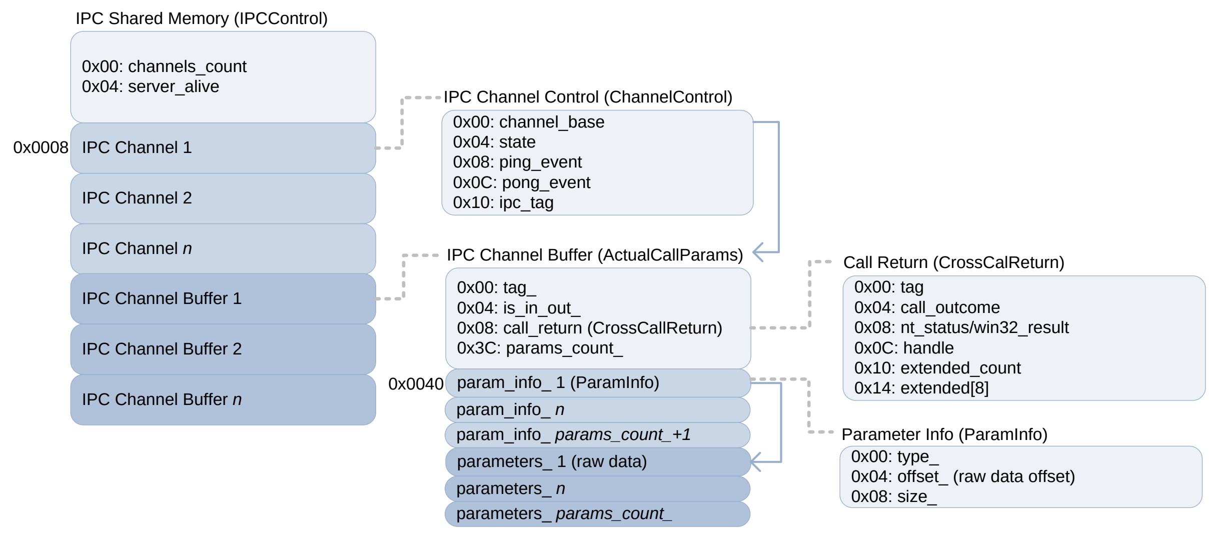
Offset	Size/Type	Name	Description
0x0000	0x04/uint32	tag	IPC Tag. Should be the same value as *ChannelControl.ipc_tag* but is not really set.
0x0004	0x04/ResultCode	call_outcome	Result code of the IPC call: *SBOX_ALL_OK* (0) or non-zero if an error occurred (see *ResultCode* enum in http://src.chromium.org/viewvc/chrome/trunk/src/sandbox/src/sandbox_types.h?view=markup).
0x0008	0x04/NTSTATUS, DWORD	nt_status/win32_result	Return value of an API call when executed on the broker/browser process. Used for API interceptions.
0x000C	0x04/HANDLE	handle	If the IPC call returns a handle, generally, the handle or the handle duplicate is stored here.
0x0010	0x04/uint32	extended_count	Number of extended return values.
0x0014	0x04*8/MultiType	extended	Array of 8 *MultiType* extended return values. *MultiType* is a union and its members are: *unsigned_int*, *pointer*, *handle* or *ulong_ptr*.

Table 6. CrossCallReturn Structure. (Reference: http://src.chromium.org/viewvc/chrome/trunk/src/sandbox/src/crosscall_params.h?view=markup)

IPC SHARED MEMORY ILLUSTRATION

Below is an illustration of the Sandbox IPC shared memory structure and the substructures it contains:

Sandbox IPC Shared Memory

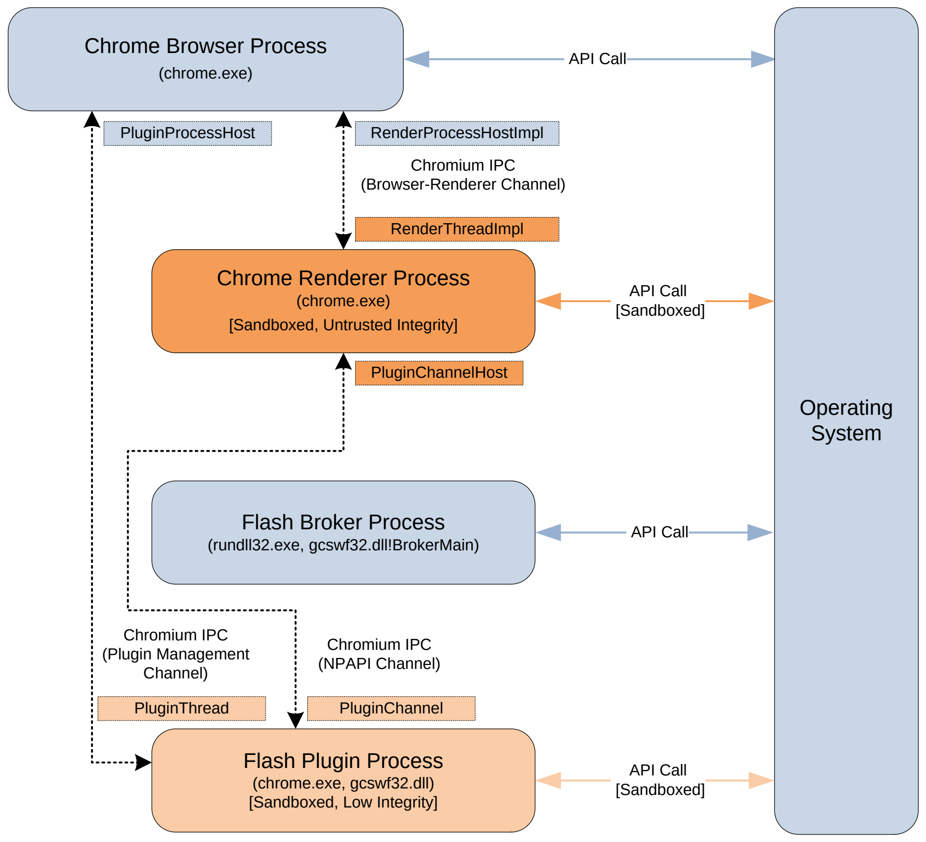


5.4.2. CHROMIUM IPC

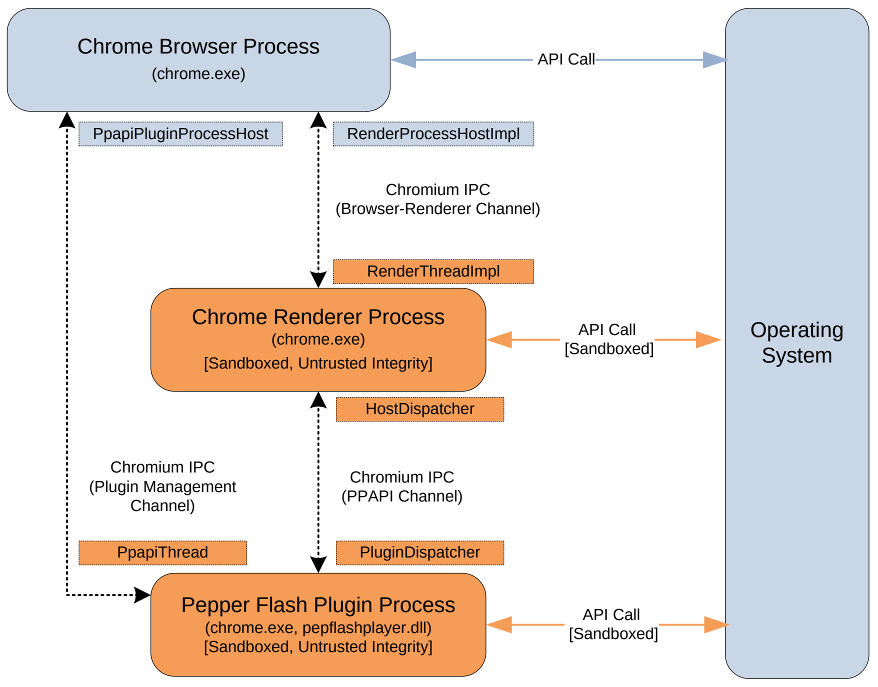
Originally from the Chromium project, the Chromium IPC is another IPC implementation used by all Flash sandbox implementations. The Chromium IPC is not only used for communication between processes with different privileges but it is also used for communication between processes with the same privileges.

In Chrome Flash and Pepper Flash, Chromium IPC is used by the sandboxed Flash plugin, the sandboxed Chrome renderer, and the Chrome browser process to communicate with each other:

Flash Player Protected Mode For Chrome (Chrome Flash)

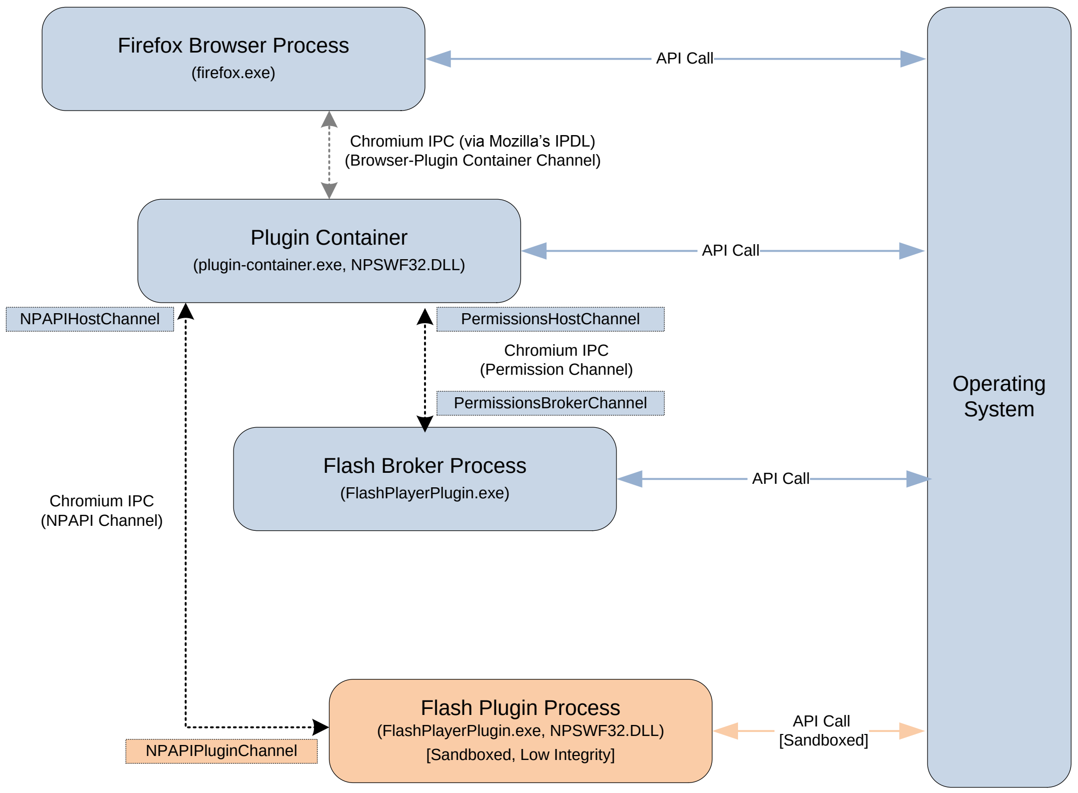


Flash Player Protected Mode For Chrome Pepper (Pepper Flash)



In Firefox Flash, Chromium IPC is used by the plugin container process to communicate with the sandboxed Flash plugin, Flash broker, and the Firefox browser process:

Flash Player Protected Mode For Firefox (Firefox Flash)



5.4.2.1. IMPLEMENTATION

On Windows, Chromium IPC is done via a named pipe. Chromium IPC pipes are named using the format "*\\.\pipe\chrome.%channel_id%*". *%channel_id%* varies in format depending on the type of connection the Chromium IPC is used for.

For setting up the Chromium IPC connection, the process hosting the IPC server generates a unique pipe name and creates the pipe. With the pipe name usually passed via command line switch, the other process hosting the IPC client then connects to the IPC server.

Two detailed articles also covering the Chromium IPC is the Chromium IPC design document [13] and Azimuth Security's blog post "*The Chrome Sandbox Part 2 of 3: The IPC Framework*" [14] .

LISTENERS

The square boxes in the diagrams are the names of the *Listener* class (http://src.chromium.org/viewvc/chrome/trunk/src/ipc/ipc_listener.h?view=markup) that handles the IPC connection in each process.

Listener classes have an *OnMessageReceived()* method which dispatches the IPC messages to the appropriate service handlers. There are also cases where the *Listener* class which handles the IPC connection do not process the IPC messages themselves but instead "*routes*" the IPC messages to instances of other *Listener* classes.

Chromium IPC messages which are processed by the *Listener* class that handles the IPC connection are called "*control*" messages while Chromium IPC messages which are routed to instances of other *Listener* class are called "*routed*" messages. Furthermore, for control messages, *Listener* classes that handles the IPC connection may also implement an *OnControlMessageReceived()* method which is specifically used for dispatching control messages.

MESSAGE FILTERS

Before the IPC messages are sent to a *Listener* class, the IPC messages may be first sent to a list of message filters (*MessageFilter*, http://src.chromium.org/viewvc/chrome/trunk/src/ipc/ipc_channel_proxy.h?view=markup). Similar to the *Listener class*, the *MessageFilter* class also have an *OnMessageReceived()* method which dispatches the IPC message to the appropriate service handlers.

MESSAGE CLASSES

Creating and dispatching IPC messages is made simpler in Chrome because there are available macros that automatically build IPC message classes. The following is an example use of the IPC message class generation macro to generate the *PpapiMsg_LoadPlugin* message class:

```
IPC_MESSAGE_CONTROL1(PpapiMsg_LoadPlugin, FilePath /* path */)
```

The above line expands to an *IPC::Message* class definition in which the generated class contains the code to build and dispatch the IPC message. For sending an IPC message, the generated IPC message class can just be instantiated and the resulting class instance can be passed to *Send(Message* message)* methods:

```
Send(new PpapiMsg_LoadPlugin(plugin_path_));
```

The macros related to IPC message class generation are:

- *IPC_MESSAGE_CONTROL**
- *IPC_MESSAGE_ROUTED**
- *IPC_SYNC_MESSAGE_CONTROL**
- *IPC_SYNC_MESSAGE_ROUTED**

MESSAGE DISPATCHING

To simplify dispatching of the IPC messages to the appropriate service handler routines, *Listener* classes use the previously discussed generated IPC message classes plus another set of Chrome macros on their *OnMessageReceived()* or *OnControlMessageReceived()* method:

```
bool PpapiThread::OnMessageReceived(const IPC::Message& msg) {
  IPC_BEGIN_MESSAGE_MAP(PpapiThread, msg)
    IPC_MESSAGE_HANDLER(PpapiMsg_LoadPlugin, OnMsgLoadPlugin)
    IPC_MESSAGE_HANDLER(PpapiMsg_CreateChannel, OnMsgCreateChannel)
    ...
  IPC_END_MESSAGE_MAP()
```

The above code expands to a switch statement which uses the type field of the IPC message to dispatch the IPC message to the appropriate service handler:

```
bool PpapiThread::OnMessageReceived(const IPC::Message& msg) {
  {
    typedef PpapiThread _IpcMessageHandlerClass;
    const IPC::Message& ipc_message__ = msg;
    bool msg_is_ok__ = true;
    switch (ipc_message__.type())
    case PpapiMsg_LoadPlugin::ID: {
        ...
        msg_is_ok__ = PpapiMsg_LoadPlugin::Dispatch(&ipc_message__, this, this,
                                      &_IpcMessageHandlerClass::OnMsgLoadPlugin);
      }
      break;
    case PpapiMsg_CreateChannel::ID: {
      ...
      msg_is_ok__ = PpapiMsg_CreateChannel::Dispatch(&ipc_message__, this, this,
                                      &_IpcMessageHandlerClass::OnMsgCreateChannel);
      }
      break;
```

The Chromium macros related to Chromium IPC message dispatching are:

- *IPC_BEGIN_MESSAGE_MAP_EX*
- *IPC_BEGIN_MESSAGE_MAP*
- *IPC_MESSAGE_HANDLER*
- *IPC_MESSAGE_HANDLER_DELAY_REPLY*
- *IPC_MESSAGE_HANDLER_GENERIC*
- *IPC_REPLY_HANDLER*
- *IPC_MESSAGE_UNHANDLED*
- *IPC_MESSAGE_UNHANDLED_ERROR*
- *IPC_END_MESSAGE_MAP*
- *IPC_END_MESSAGE_MAP_EX*

5.4.2.2. MESSAGE STRUCTURE

CHROMIUM IPC MESSAGE STRUCTURE

The structure of a Chromium IPC message is shown below. Note that the Chromium IPC message structure may vary depending on the operating system and build parameters. The structure shown below is based on release builds of Google Chrome for Windows.

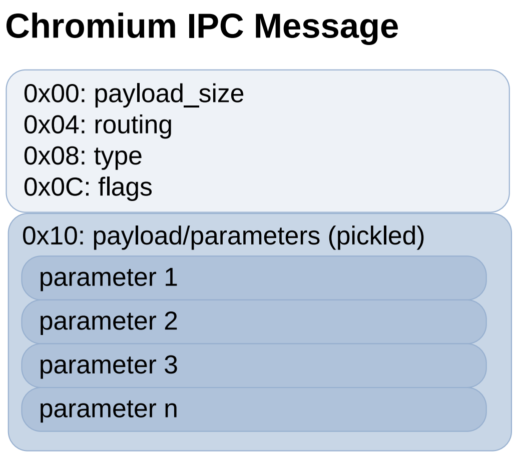
Offset	Size/Type	Name	Description
0x0000	0x04/uint32	payload_size	Total size of the payload (i.e. IPC call parameters).
0x0004	0x04/int32	routing	Routing ID. For control messages, the value of routing ID is *MSG_ROUTING_CONTROL* (0x7FFFFFFF). For routed messages, routing ID is a value identifying the instance of the *Listener* class which the IPC message will be routed to.
0x0008	0x04/uint32	type	Message Type. A unique ID specifying the service to be invoked. The upper 16 bit of the *type* field is the service group (discussed below) while the lower 16 bits is a unique value within a particular service group. The service group part of the *type* field can be used to

			identify the kind of service being invoked. For Chrome processes (which includes the Chrome Flash plugin and Pepper Flash plugin process), service group values can be derived from the *IPCMessageStart* enum in http://src.chromium.org/viewvc/chrome/trunk/src/ipc/ipc_message_utils.h?view=markup.
0x000C	0x04/uint32	flags	Message Flags. The first 3 bits are for message priority: *PRIORITY_LOW* (1), *PRIORITY_NORMAL* (2), PRIORITY_HIGH (3). The rest of the bits are for flags: *SYNC_BIT* (0x0004), *REPLY_BIT* (0x0008), *REPLY_ERROR_BIT* (0x0010), *UNBLOCK_BIT* (0x0020), *PUMPING_MSGS_BIT* (0x0040), *HAS_SENT_TIME_BIT* (0x0080).
0x0010		payload	Serialized IPC call parameters. The class responsible for serializing/deserializing basic types is the *Pickle* class (http://src.chromium.org/viewvc/chrome/trunk/src/base/pickle.h?view=markup). Complex types are serialized/deserialized by specializations of the struct template called *ParamTraits*. *ParamTraits* methods, on the other hand, invoke *Pickle* class methods to serialize/deserialize the basic types which the complex type is composed of.

Table 7. Chromium IPC Message. (Reference: http://src.chromium.org/viewvc/chrome/trunk/src/ipc/ipc_message.h?view=markup)

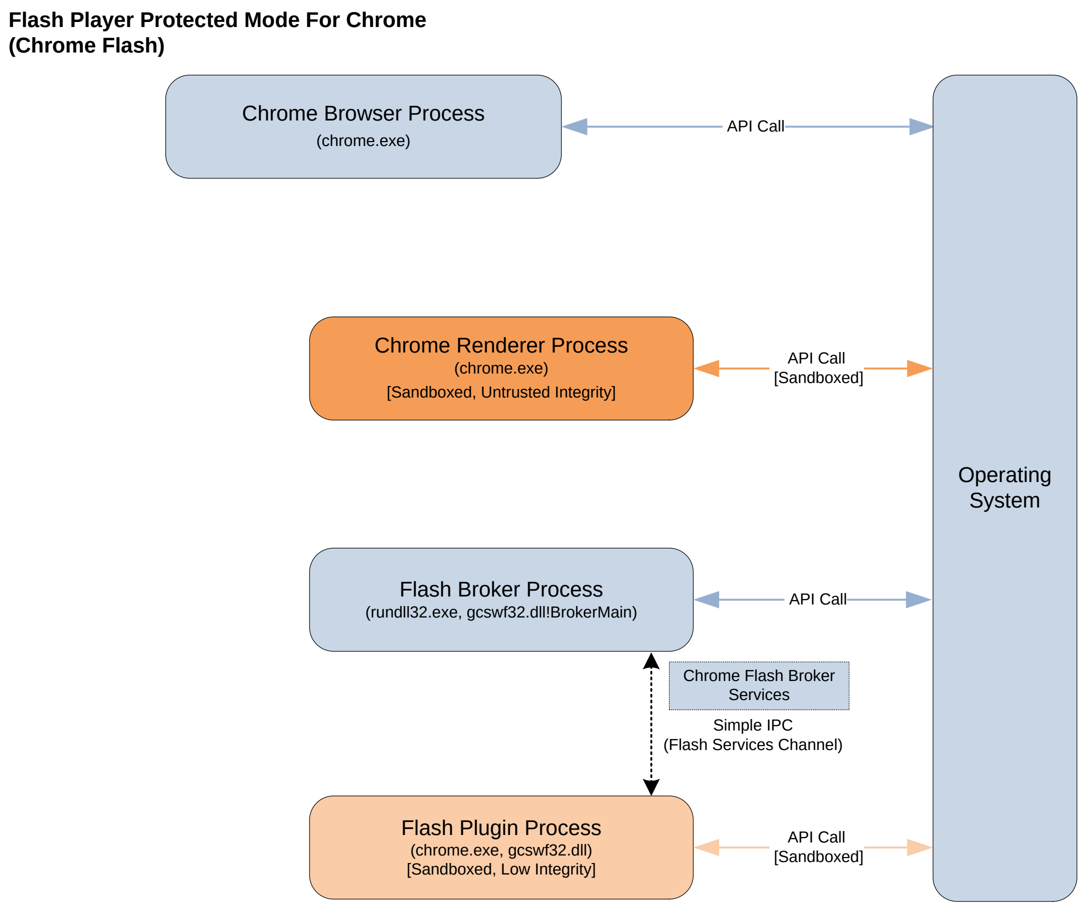
CHROMIUM IPC MESSAGE ILLUSTRATION

Below is an illustration of the Chromium IPC message structure:



5.4.3. SIMPLE IPC

Chrome Flash additionally uses an IPC implementation called "*Simple IPC*" for communication between the Flash plugin process and the Flash broker process, it is mainly used by the Flash plugin process to invoke the Chrome Flash broker services (discussed in 5.5.3) exposed by the Flash broker process:


Flash Player Protected Mode For Chrome (Chrome Flash)
Chrome Browser Process
(chrome.exe)
API Call
Chrome Renderer Process
(chrome.exe)
[Sandboxed, Untrusted Integrity]
API Call [Sandboxed]
Operating System
Flash Broker Process
(rundll32.exe, gcswf32.dll!BrokerMain)
API Call
Chrome Flash Broker Services
Simple IPC (Flash Services Channel)
Flash Plugin Process
(chrome.exe, gcswf32.dll)
[Sandboxed, Low Integrity]
API Call [Sandboxed]

5.4.3.1. Implementation

On Windows, Simple IPC is done via a named pipe. In Chrome Flash, Simple IPC pipes are named using the format "\\.*pipe\MacromediaKappa.%flash_broker_hex_pid%*".

For setting up the Simple IPC connection, the Flash broker process which hosts the IPC server creates the Simple IPC pipe. Then, using the Flash broker PID passed via the command line switch "*--flash-broker=*", the Flash plugin process which hosts the IPC client generates the Simple IPC pipe name and then uses it to connect to the IPC server.

Simple IPC is developed by Google and its source can be found in the *simple-ipc-lib* project hosted in http://code.google.com/p/simple-ipc-lib/.

5.4.3.2. Message Structure

Simple IPC Message Structure

The structure of a Simple IPC message is as follows:

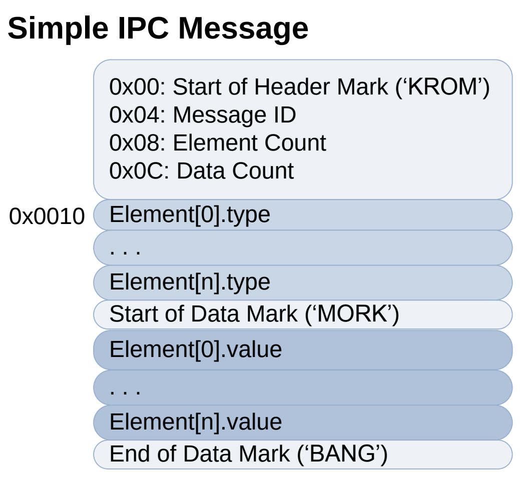
Offset	Size (Type)	Name	Description
0x0000	0x04/int32	Start of Header Mark	The value 'KROM'.
0x0004	0x04/int32	Message ID	A unique ID identifying the service to be invoked.
0x0008	0x04/int32	Element Count	Number of elements (i.e. IPC call parameters).

0x000C	0x04/size_t	Data Count	Total size of the IPC message (in 4 bytes). Messages should be aligned by 4 bytes.
0x0010	0x04/int32	Element[0].tag	Element Tag (i.e. parameter type): *TYPE_INT32* (0x01), *TYPE_UINT32* (0x02), *TYPE_LONG32* (0x03), *TYPE_ULONG32* (0x04), *TYPE_CHAR8* (0x05), *TYPE_CHAR16* (0x06), *TYPE_VOIDPTR* (0x07), *TYPE_NULLSTRING8* (0x08), *TYPE_NULLSTRING16* (0x09), *TYPE_NULLBARRAY* (0x0A), *TYPE_STRING8* (0x40000016), *TYPE_STRING16* (0x80000017), *TYPE_BARRAY* (0x40000018)
?	0x04/int32	Element[n].tag	…
?	0x04/int32	Start of Data Mark	The value 'MORK'.
?		Element[0].value	Serialized element (parameter) value.
?		Element[n].value	…
?	0x04/uint32	End of Data Mark	The value 'BANG'.

Table 8. Simple IPC Message Structure. (Reference: http://code.google.com/p/simple-ipc-lib/source/browse/trunk/src/ipc_codec.h)

SIMPLE IPC MESSAGE ILLUSTRATION

Below is an illustration of a Simple IPC message:



5.5. SERVICES

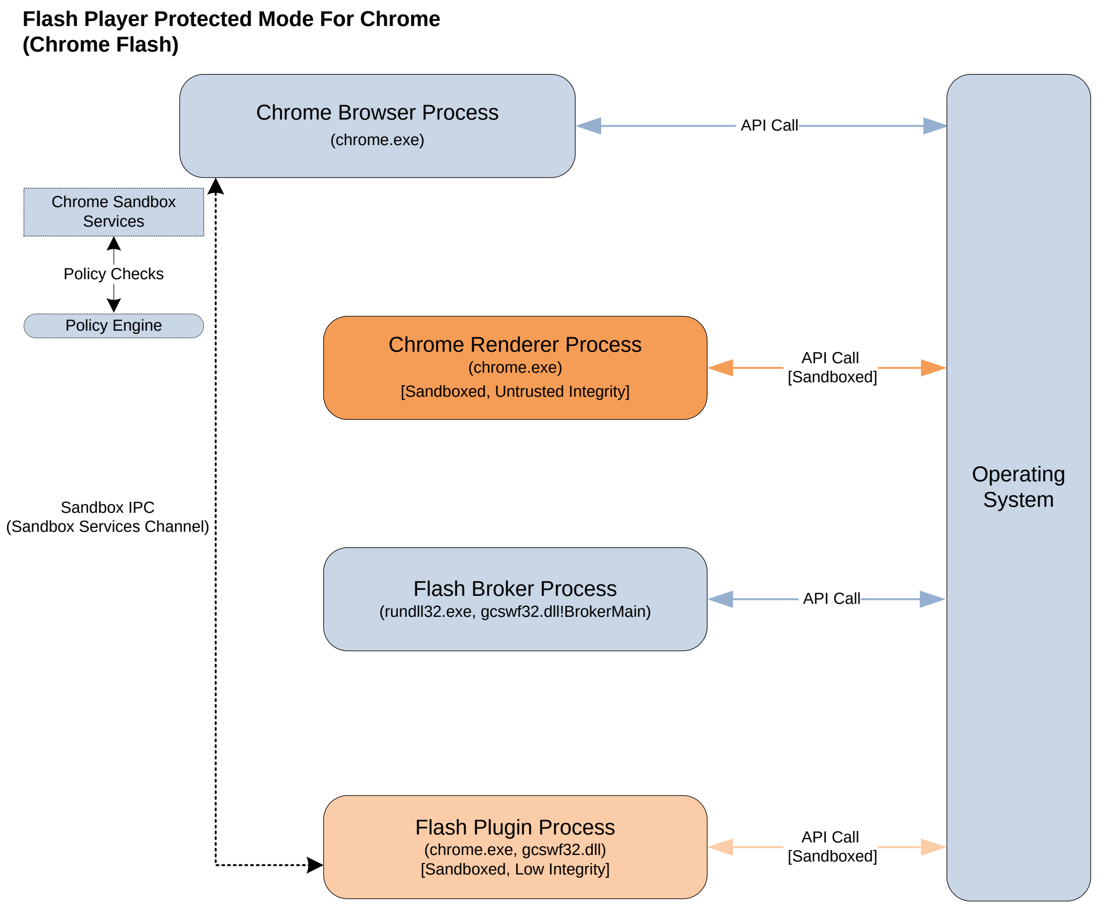
After discussing the connection between processes, we will now take a look at the services exposed by the different processes that are part of the Flash sandbox implementations.

5.5.1. CHROME SANDBOX SERVICES

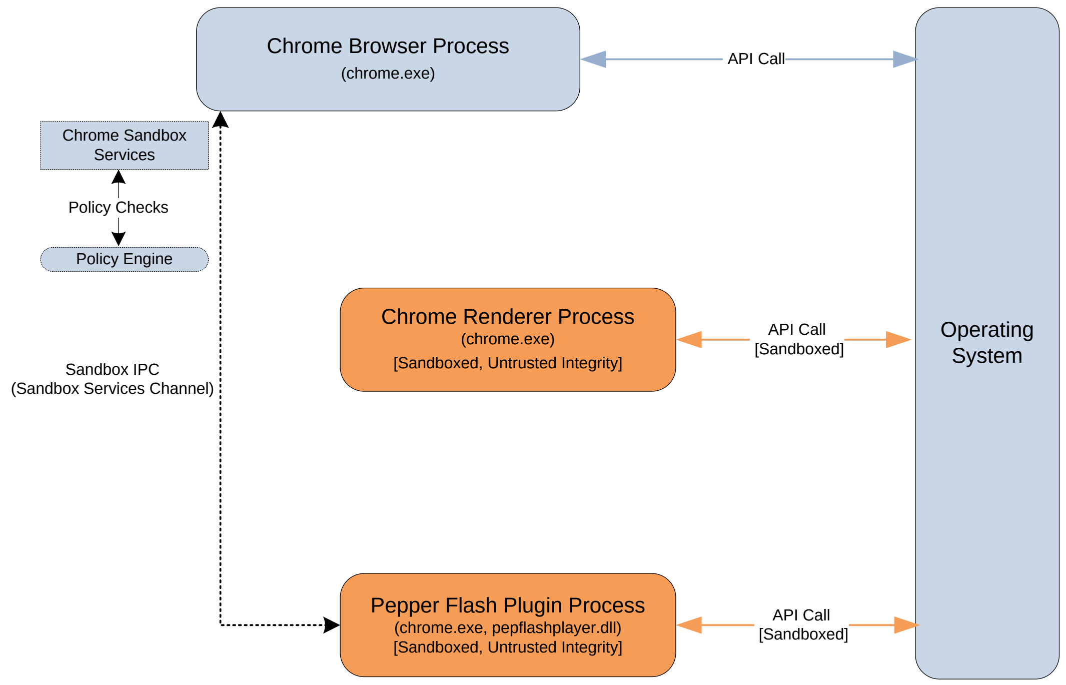
The Chrome browser process provides sandboxed processes services that allow them to call security sensitive APIs (such as opening a file, registry key, etc.) provided that the API call is allowed by the policy. Generally, these sandbox services are not directly invoked by the sandboxed process but are invoked indirectly via API interceptions in the sandboxed process.

5.5.1.1. CONNECTION

In the context of Chrome Flash and Pepper Flash, the Chrome Sandbox services are callable from the sandboxed Flash plugin process via a Sandbox IPC connection:



Flash Player Protected Mode For Chrome Pepper (Pepper Flash)



5.5.1.2. SERVICES

Chrome sandbox service handlers are methods of *Dispatcher* classes (http://src.chromium.org/viewvc/chrome/trunk/src/sandbox/src/crosscall_server.h?view=markup) and each *Dispatcher* class handles a specific set of APIs. The table below lists the different *Dispatcher* classes in Chrome:

Dispatcher Class	Purpose
FilesystemDispatcher	Handles file system services. Handles forwarded filesystem-related *NTDLL.DLL* API calls.
HandleDispatcher	Handles handle duplication services.
NamedPipeDispatcher	Handles named pipe services. Handles forwarded *CreateNamedPipeW()* API calls.
PolicyBase	Special *Dispatcher* class. Among other things, the Sandbox IPC server uses *PolicyBase* to resolve actual *Dispatcher* class and the actual service handler routine to service the request.
RegistryDispatcher	Handles registry services. Handles forwarded *NtOpenKey()* and *NtCreateKey()* API calls.
SyncDispatcher	Handles synchronization (events) services. Currently handles forwarded *CreateEventW()* and *OpenEventW()* API calls.
ThreadProcessDispatcher	Handles process and thread services. Currently handles forwarded *CreateProcessW()*, *CreateProcessA()*, and other thread/process-related API calls.

The source for the listed *Dispatcher* classes can be found in the "**_dispatcher.cc*" files in Chromium's sandbox tree (http://src.chromium.org/viewvc/chrome/trunk/src/sandbox/src/).

Each service handler is registered as an *IPCCall* structure which is initialized in the constructor of each *Dispatcher* class. An *IPCCall* call structure contains the following information:

- The IPC tag for the service handler - equivalent to a service ID, see Sandbox IPC section (5.4.1) for more information
- The sequence and type of parameters expected by the service handler
- The actual handler routine

Below is an example *IPCCall* structure initialization:

```
FilesystemDispatcher::FilesystemDispatcher(PolicyBase* policy_base)
    : policy_base_(policy_base) {

  static const IPCCall create_params = {
    {IPC_NTCREATEFILE_TAG, WCHAR_TYPE, ULONG_TYPE, ULONG_TYPE, ULONG_TYPE,
     ULONG_TYPE, ULONG_TYPE, ULONG_TYPE},
    reinterpret_cast<CallbackGeneric>(&FilesystemDispatcher::NtCreateFile)
  };

  ...
```

Consequently, enumerating the Chrome Sandbox service handlers involves finding initializations of *IPCCall* structures.

5.5.2. CHROME PLUGIN SERVICES

To support out-of-process NPAPI and PPAPI plugins such as Chrome Flash and Pepper Flash, the Chrome renderer and the Chrome browser process expose services which callable from out-of-process plugins.

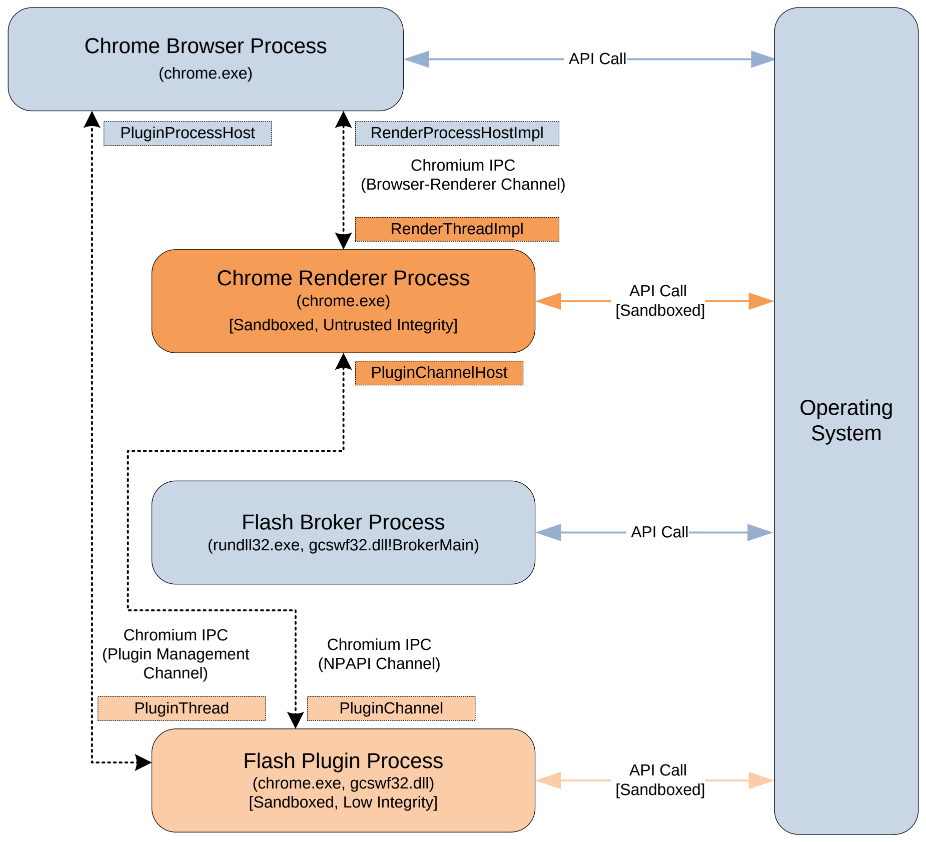
Generally, the services exposed by the browser process to the NPAPI/PPAPI plugin process are mostly for sending status or notification messages. On the other hand, the renderer process does most of the heavy lifting as it exposes services to the plugin process to support NPAPI and PPAPI calls.

5.5.2.1. CONNECTION

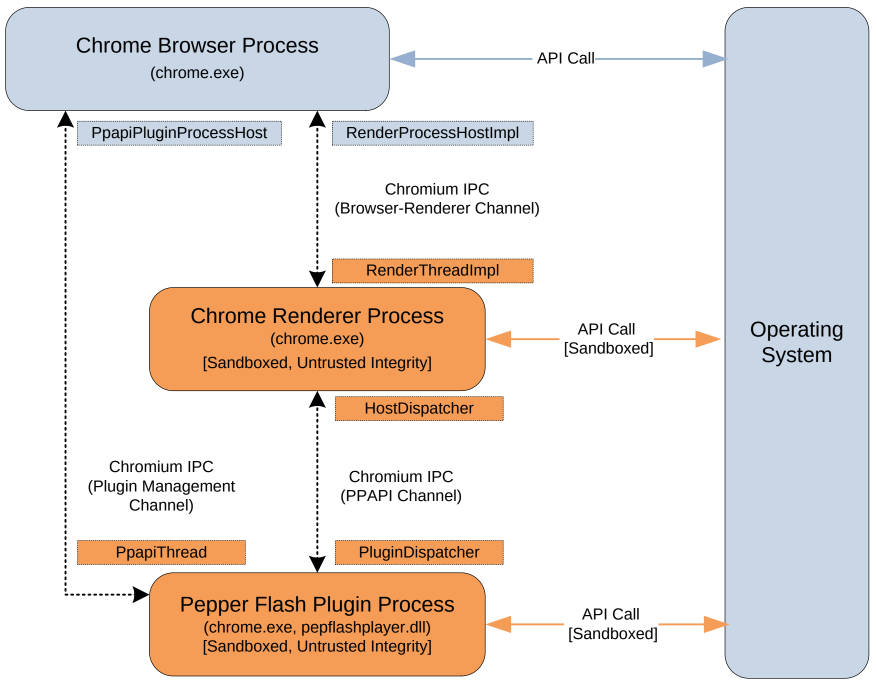
In the case of Chrome Flash and Pepper Flash, the services exposed by the Chrome browser and the Chrome renderer process to the Flash plugin processes are callable via two separate Chromium IPC connections. A third Chromium IPC connection is employed by the renderer process to communicate with the browser process. In the context of plugins, the browser-renderer channel is used for executing privileged plugin service requests in the context of the browser process.

The diagrams below show how the Flash plugin process, the Chrome browser and the Chrome renderer process are interconnected via Chromium IPC.

Flash Player Protected Mode For Chrome (Chrome Flash)



Flash Player Protected Mode For Chrome Pepper (Pepper Flash)



As mentioned earlier in the Chromium IPC section (5.4.2), the square boxes represent the *Listener* class which handles the IPC connection in each process. The *Listener* classes dispatch the IPC messages to the appropriate service handlers or routes the IPC message to other *Listener* classes which will handle the IPC message.

5.5.2.2. SERVICES (OUT-OF-PROCESS NPAPI PLUGIN)

The table below lists the different services exposed by the Chrome renderer and Chrome browser process to out-of-process NPAPI plugins. The listener column represents the name of the *Listener* class which handles the dispatching of the IPC message to the appropriate service handler. The message column represents names of the message classes which are used by the aforementioned *Listener* classes for selecting and invoking the appropriate service handler.

Message	Sent To	Listener	Purpose
PluginProcessHostMsg_*	Browser	PluginProcessHost	Sending plugin status or notifications to the browser process.
PluginHostMsg_*	Renderer	PluginChannelHost WebPluginDelegateProxy	Support services for NPAPI *NPN_** calls. The renderer uses the services exposed by the browser process (via the browser-renderer channel) to handle privileged NPAPI service requests.
NPObjectMsg_*	Renderer/ Plugin	NPObjectStub	Marshalling *NPObjects* [15] between the plugin process and the renderer process.

5.5.2.3. SERVICES (OUT-OF-PROCESSES PPAPI PLUGIN)

The table below lists the different services exposed by the Chrome renderer and Chrome browser process to out-of-process PPAPI plugin processes.

Message	Sent To	Listener	Purpose
PpapiHostMsg_*	Browser	PpapiPluginProcessHost	Sending plugin status or notification to the browser process.
PpapiHostMsg_*	Renderer	Subclasses of InterfaceProxy	PPAPI services. PPAPI services are exposed by a process via interface proxies (*InterfaceProxy, http://src.chromium.org/viewvc/chrome/trunk/src/ppapi/proxy/interface_proxy.h?view=markup).* *InterfaceProxy* is actually a subclass of the *Listener* class and thus have an *OnMessageReceived()* method which dispatches the PPAPI IPC messages to the appropriate service handler. The renderer uses the services exposed by

	the browser process (via the browser-renderer channel) to handle privileged PPAPI service requests.

Below are some examples of PPAPI interface proxies along with the corresponding messages classes used by the interface proxies for dispatching the PPAPI IPC messages to the actual service handlers:

Message	Interface Proxy	Purpose
PpapiHostMsg_PPBAudio_*	PPB_Audio_Proxy	Audio services
PpapiHostMsg_PPBFileChooser_*	PPB_FileChooser_Proxy	Open/save dialog services
PpapiHostMsg_PPBFileIO_*	PPB_FileIO_Proxy	File I/O services
PpapiHostMsg_PPBFlashClipboard_*	PPB_Flash_Clipboard_Proxy	Clipboard services
PpapiHostMsg_PPBVideoCapture_*	PPB_VideoCapture_Proxy	Video capture services

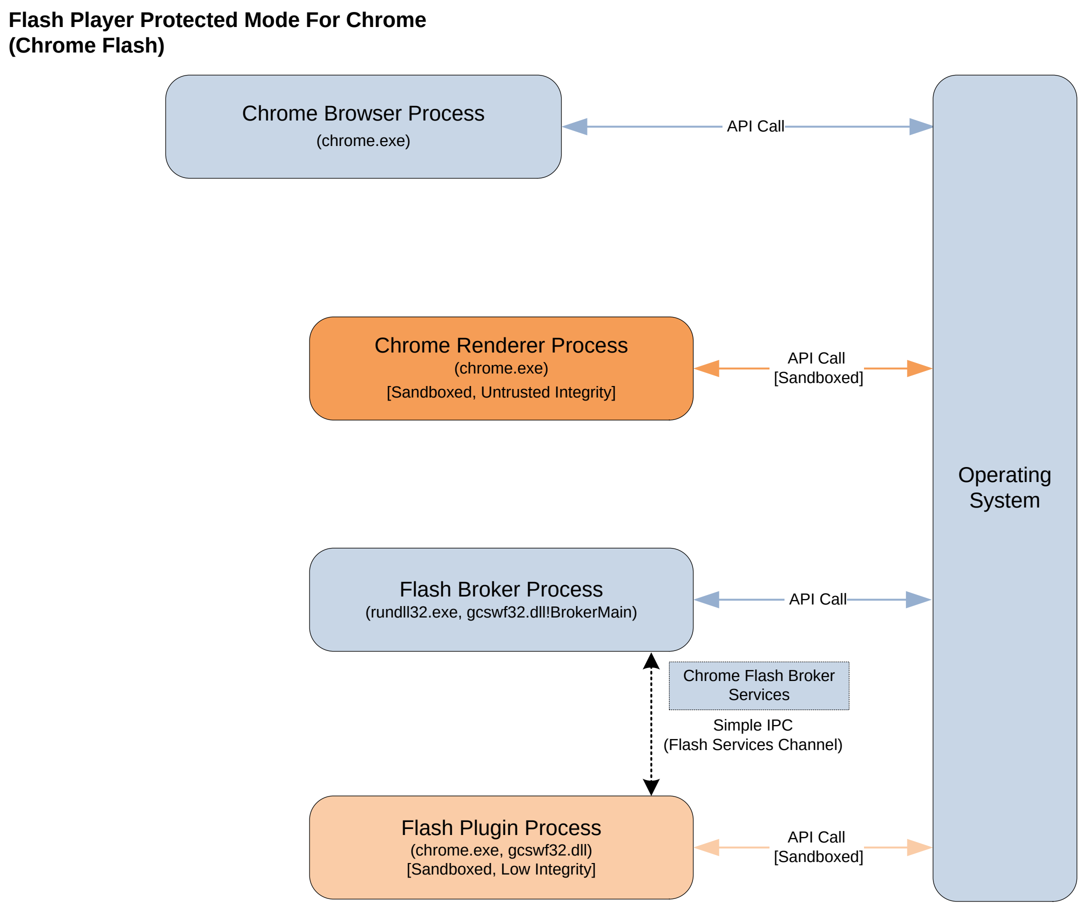
The sources of PPAPI interface proxies are found in the Chromium source tree under "*src/ppapi/proxy*" (http://src.chromium.org/viewvc/chrome/trunk/src/ppapi/proxy/) and their names are formatted as "**_proxy.cc*".

5.5.3. CHROME FLASH BROKER SERVICES

Chrome Flash includes a separate broker process which exposes additional services to the sandboxed Flash plugin process. Example of the services exposed by the Chrome Flash broker includes displaying an open/save file dialog and launching the Flash Player settings manager.

5.5.3.1. CONNECTION

The services exposed by the Flash broker process are callable from Flash plugin process via a Simple IPC connection:



5.5.3.2. SERVICES

The table below lists the type of services exposed by the Chrome Flash broker to the Flash plugin process. The code for these services is located in the *gcswf32.dll* binary.

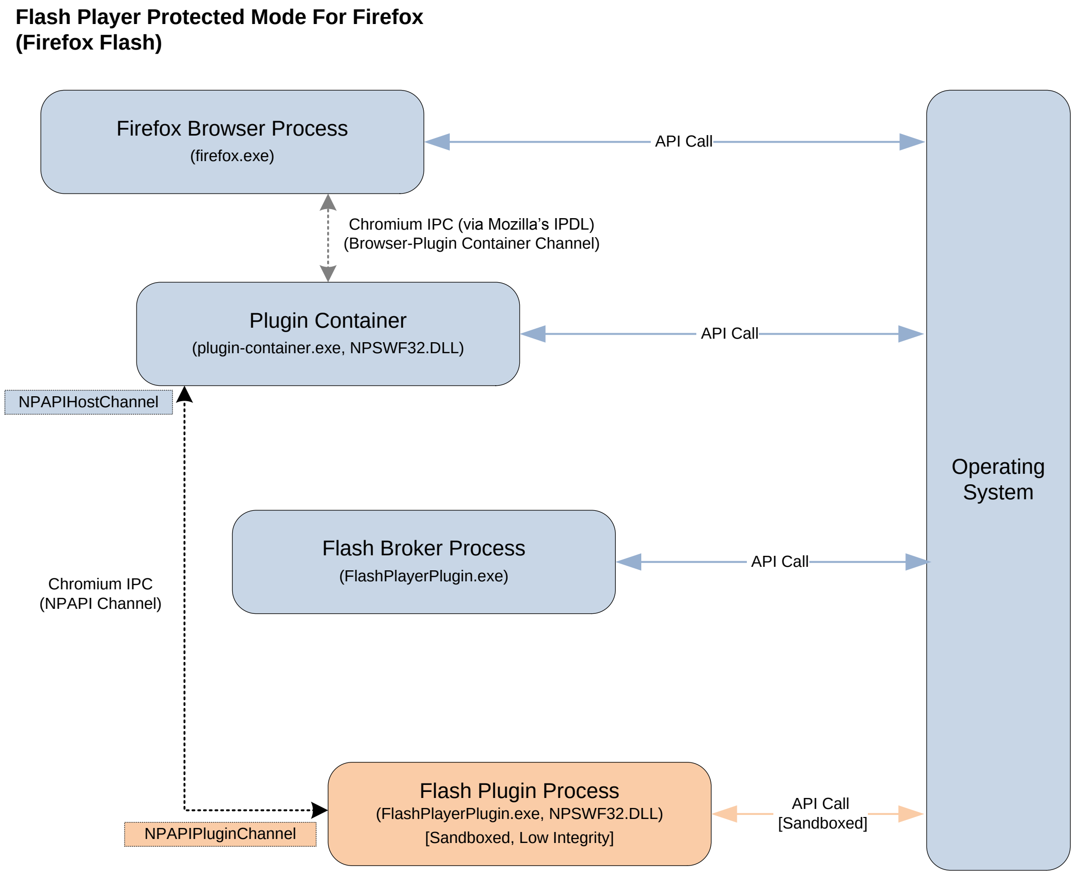
Service	Purpose
Dialog Services	Opening an open/save file dialog.
Filesystem Services	Brokering calls to *FindFirstFileW()*, *FindNextFileW()*, *CreateFileW()*, *MoveFileExW()* and *CreateDirectoryW()*.
'LM" Services	Services for downloading and launching signed content.
Memory mapping and Mutant Services	Brokering calls to *CreateFileMappingW()* and *CreateMutexW()*.
Network Services	Brokering calls to *WININET* APIs.
Miscellaneous Services	Such as launching the Flash settings manager.

5.5.4. FIREFOX FLASH PLUGIN CONTAINER SERVICES

The privileged plugin container process in Firefox exposes services that are callable from the sandboxed Flash plugin process. These exposed services are mainly used for proxying NPAPI calls between the Firefox browser process and the sandboxed Flash plugin process.

5.5.4.1. CONNECTION

The services exposed by the plugin container process are callable from the Flash plugin process via a Chromium IPC connection:



The pipe name used for the Chromium IPC connection is formatted as "*\\.\pipe\chrome.Flash%plugin_container_pid%.%p.%d*". The IPC connection is handled by the *NPAPIHostChannel Listener* class in the plugin container process, while the IPC connection is handled by the *NPAPIPluginChannel Listener* class in the Flash plugin process.

Additionally, as shown in the diagram, a separate Chromium IPC connection exists between the Firefox browser process and the plugin container. This Chromium IPC connection is used for forwarding NPAPI calls between the browser process and the plugin container process. On a high-level, communication via this IPC connection are governed by Mozilla's Inter-process communication Protocol Definition Language (*IPDL*) [16], on a low-level, the actual messages are also sent via Chromium IPC. More information about Firefox's multi-process architecture can be found on Mozilla's *Electrolysis* project page [17].

5.5.4.2. SERVICES

The table below lists the services exposed by the privileged plugin container to the sandboxed Flash plugin process. The code for these services is located in the *NPSWF32.dll* binary.

Message Type	Sent To	Listener	Purpose
NPAPIHostChannel Messages	Plugin Container	NPAPIHostChannel	Mostly for proxying NPAPI *NPN_** APIs from the Flash plugin process to the browser process. *NPN_** APIs proxied by *NPAPIHostChannel* are those that do not require a plugin instance (e.g. *NPN_GetIntIdentifier()* and *NPN_ReloadPlugins()*).
NPAPIPluginProxy Messages	Plugin Container	NPAPIPluginProxy	Mostly for proxying NPAPI *NPN_** APIs from the Flash plugin process to the browser process. *NPN_** APIs proxied by *NPAPIPluginProxy* are those that require a plugin instance (e.g. *NPN_GetURL()* and *NPN_PostURL()*). Various window-related services are also exposed by the plugin container to the Flash plugin process via the *NPAPIPluginProxy* messages.
NPObject Messages	Plugin Container Flash Plugin	NPObjectStub	Proxying *NPObject* calls between the Flash plugin process and the browser process.

5.5.5. FIREFOX FLASH BROKER SERVICES

The services exposed by the Firefox Flash broker can be divided into three groups:

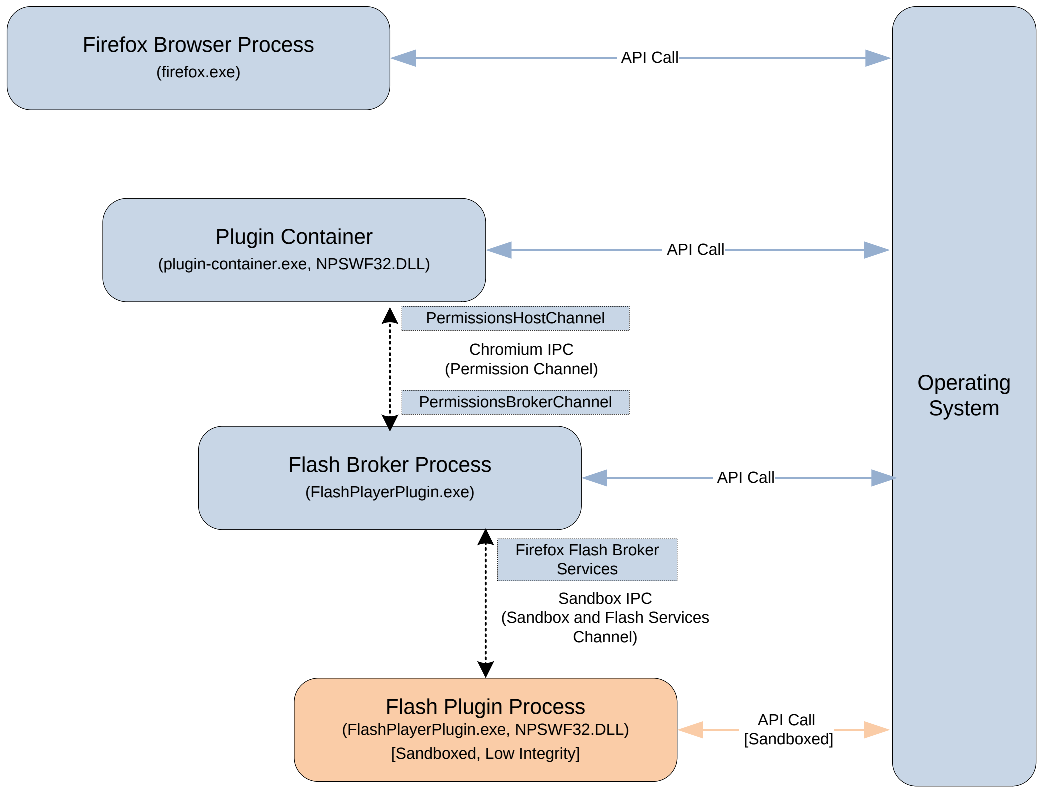
1. Sandbox Services
2. Flash Services
3. Permission Services

The Sandbox services are equivalent to the Chrome Sandbox services (discussed in 5.5.1). Specifically, they handle forwarded API calls from the Flash plugin process. Flash services are additional services exposed to the Flash plugin such as displaying an open/save dialog and launching the Flash Player settings manager. Permission services are services for managing access permissions of the sandboxed Flash plugin process.

5.5.5.1. CONNECTION

The Sandbox and Flash services exposed by the Flash broker process are callable from the sandboxed Flash plugin process via a Sandbox IPC connection. The permission services exposed by the Flash broker are callable from the plugin container via a Chromium IPC connection:

Flash Player Protected Mode For Firefox (Firefox Flash)



The name of the IPC pipe used in the Chromium IPC connection between the plugin container and the Flash broker is formatted as "\\.*pipe\chrome.Flash%plugin_container_pid%.%p.%d*". The Chromium IPC connection is handled by the *PermissionsHostChannel Listener* class in the plugin container process, while the Chromium IPC connection is handled by the *PermissionsBrokerChannel Listener class* in the Flash broker process.

5.5.5.2. SERVICES (SANDBOX AND FLASH SERVICES)

The table below lists the Sandbox and Flash services exposed by the Flash broker to the Flash plugin process. The code for these services is located in the *FlashPlayerPlugin.exe* binary. By using the RTTI information embedded in *FlashPlayerPlugin.exe* binary in addition to referencing the Chromium code, we were able to recover the names of the *Dispatcher* classes.

Dispatcher Class	Purpose
FilesystemDispatcher	Handles file system services. Handles forwarded filesystem-related *NTDLL.DLL API* calls.
MutantDispatcher	Handles synchronization (mutant) services. Handles forwarded *NtCreateMutant()* and *NtOpenMutant()* API calls.
NamedPipeDispatcher	Handles named pipe services. Handles forwarded *CreateNamedPipeW()* API calls.
PolicyBase	Special dispatcher class. Among other things, the Sandbox IPC server uses *PolicyBase* to resolve actual dispatcher class and the actual handler

	routine to service the request.
RegistryDispatcher	Handles registry services. Handles forwarded *NtOpenKey()* and *NtCreateKey()* API calls.
SandboxBrokerServerDispatcher	Miscellaneous broker services. Handles services which are not covered by the other dispatcher classes such as launching the Flash Player settings manager. Has interesting service handler routines and is a large attack surface due to the number of its service handler routines.
SandboxVideoCaptureDispatcher *(possible name)*	Handles video capture services.
SandboxClipboardDispatcher	Handles clipboard services. Mostly handles forwarded clipboard-related *USER32.DLL* API calls.
SandboxCryptDispatcher	Handles cryptographic services. Mostly handles forwarded crypto-related *SECUR32.DLL/SSPICLI.DLL* and *CRYPT32.DLL API* calls.
SandboxPrintDispatcher	Handles printing services.
SandboxWininetDispatcher	Handles *WININET* services. Mostly handles forwarded *WININET.DLL* API calls.
SectionDispatcher	Handles section object services. Currently handles forwarded *NtCreateSection()* and *NtOpenSection()* API calls.
SyncDispatcher	Handles synchronization (events) services. Currently handles forwarded *CreateEventW()* and *OpenEventW()* API calls.
ThreadProcessDispatcher	Handles process and thread services. Currently handles forwarded *CreateProcessW()*, *CreateProcessA()*, and other thread/process-related API calls.

5.5.5.3. SERVICES (PERMISSION SERVICES)

The table below lists the permission services exposed by the Flash broker process to the plugin container process. The code for these services is located in the *FlashPlayerPlugin.exe* binary.

Message	Sent to	Listener	Purpose
PermissionsBrokerChannel Messages	Flash Broker	PermissionsBrokerChannel	As of Firefox Flash 11.3, *PermissionBrokerChannel* messages are used for granting or denying the job object that the sandboxed Flash plugin process is associated to access to a window handle.

5.6. POLICY ENGINE

The policy engine allows the broker to specify exceptions to the default restrictions imposed in the sandbox. These exceptions, or whitelist rules, allows the broker to grant the sandbox process access to certain named objects, bypassing the sandbox restrictions.

5.6.1. ADDING POLICIES

The policy engine for all three implementations is derived from Chrome sandbox's policy engine so they share a lot in common. One of these is the way a policy is added. Policies are added programmatically using the *sandbox::PolicyBase::AddRule()* function. This function takes the following format:

```
AddRule(subsystem, semantics, pattern)
```

The subsystem parameter indicates the Windows system the rules apply. The semantics parameter indicates the permission that will be applied to the file name/path, registry name, etc. that matches the pattern expression.

The subsystems and semantics used are different for Firefox Flash and the sandboxes from Chrome, and are derived from that of Adobe Reader X's sandbox. Here are the subsystems and semantics for Firefox Flash:

Subsystem	Description
SUBSYS_FILES	Creation and opening of files and pipes.
SUBSYS_NAMED_PIPES	Creation of named pipes.
SUBSYS_PROCESS	Creation of child processes.
SUBSYS_REGISTRY	Creation and opening of registry keys.
SUBSYS_SYNC	Creation of named sync objects.
SUBSYS_MUTANT	Creation and opening of mutant objects.
SUBSYS_SECTION	Creation and opening of section objects.

Semantics	Description
FILES_ALLOW_ANY	Allows open or create for any kind of access that the file system supports.
FILES_ALLOW_READONLY	Allows open or create with read access only.
FILES_ALLOW_QUERY	Allows access to query the attributes of a file.
FILES_ALLOW_DIR_ANY	Allows open or create with directory semantics only.
NAMEDPIPES_ALLOW_ANY	Allows creation of a named pipe.
PROCESS_MIN_EXEC	Allows creation of a process with minimal rights over the resulting process and thread handles. No other parameters besides the command line are passed to the child process.

PROCESS_ALL_EXEC	Allows the creation of a process and return fill access on the returned handles. This flag can be used only when the main token of the sandboxed application is at least INTERACTIVE.
EVENTS_ALLOW_ANY	Allows the creation of an event with full access.
EVENTS_ALLOW_READONLY	Allows opening an event with synchronize access.
REG_ALLOW_READONLY	Allows read-only access to a registry key.
MUTANT_ALLOW_ANY	Allows creation of a mutant object with full access.
SECTION_ALLOW_ANY	Allows read and write access to a section.
REG_ALLOW_ANY	Allows read and write access to a registry key.

Subsystems and semantics for Chrome Flash and Pepper Flash are derived directly from Chrome sandbox's. Here are the subsystems and semantics for both Chrome Flash and Pepper Flash:

Subsystem	Description
SUBSYS_FILES	Creation and opening of files and pipes.
SUBSYS_NAMED_PIPES	Creation of named pipes.
SUBSYS_PROCESS	Creation of child processes.
SUBSYS_REGISTRY	Creation and opening of registry keys.
SUBSYS_SYNC	Creation of named sync objects.
SUBSYS_HANDLES	Duplication of handles to other processes.

Semantics	Description
FILES_ALLOW_ANY	Allows open or create for any kind of access that the file system supports.
FILES_ALLOW_READONLY	Allows open or create with read access only.
FILES_ALLOW_QUERY	Allows access to query the attributes of a file.
FILES_ALLOW_DIR_ANY	Allows open or create with directory semantics only.
NAMEDPIPES_ALLOW_ANY	Allows creation of a named pipe.

PROCESS_MIN_EXEC	Allows creation of a process with minimal rights over the resulting process and thread handles. No other parameters besides the command line are passed to the child process.
PROCESS_ALL_EXEC	Allows the creation of a process and return fill access on the returned handles. This flag can be used only when the main token of the sandboxed application is at least INTERACTIVE.
EVENTS_ALLOW_ANY	Allows the creation of an event with full access.
EVENTS_ALLOW_READONLY	Allows opening an event with synchronize access.
REG_ALLOW_READONLY	Allows read-only access to a registry key.
REG_ALLOW_ANY	Allows read and write access to a registry key.
HANDLES_DUP_ANY	Allows duplicating of handles opened with any access permissions.
HANDLES_DUP_BROKER	Allows duplicating handles to the broker process.

5.6.2. ADMIN-CONFIGURABLE POLICIES

In Firefox Flash, it is possible to add custom policies through a configuration file. This allows the administrator to set whitelists that bypasses the restrictions imposed by the sandbox. The policy file is enabled by setting the *ProtectedModeBrokerWhitelistConfigFile* option to 1 in mms.cfg. The policy file is named *ProtectedModeWhitelistConfig.txt* and should be placed in *%WINDIR%\System32\Macromed\Flash* for 32-bit *Windows, and %WINDIR%\SysWow64\Macromed\Flash* for 64-bit Windows.

Each policy rule takes the format of:

```
POLICY_RULE_TYPE = pattern string
```

The *POLICY_RULE_TYPE* is derived from the semantics and can be any of the following:

Policy Rule	Description
FILES_ALLOW_ANY	Allows open or create for any kind of access that the file system supports.
FILES_ALLOW_DIR_ANY	Allows open or create with directory semantics only.
NAMEDPIPES_ALLOW_ANY	Allows creation of a named pipe.
PROCESS_ALL_EXEC	Allows the creation of a process and return full access on the returned handles. This flag can be used only when the main token of the sandboxed application is at least INTERACTIVE.
EVENTS_ALLOW_ANY	Allows the creation of an event with full access.
REG_ALLOW_ANY	Allows read and write access to a registry key.

MUTANT_ALLOW_ANY	Allows creation of a mutant object with full access.
SECTION_ALLOW_ANY	Allows read and write access to a section.

The pattern string is similar to the pattern parameter in the hard-coded policies and denotes file names, paths, registry locations, etc.

5.7. SUMMARY: SANDBOX MECHANISMS

After discussing each Flash sandbox mechanism, as a summary, this section will discuss how all the sandbox mechanisms are interconnected.

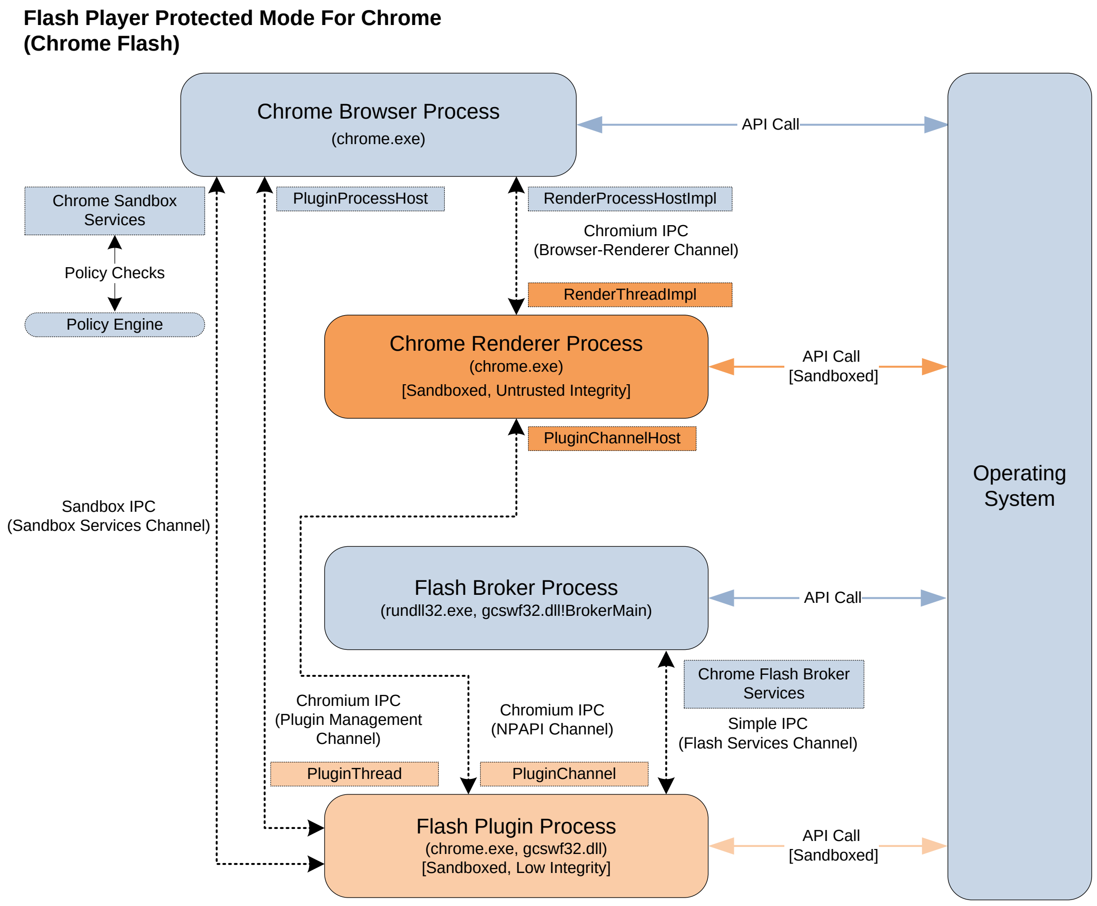
5.7.1. FLASH PLAYER PROTECTED MODE FOR CHROME (CHROME FLASH)

In Chrome Flash, the sandboxed Flash plugin process connects to the following processes:

- Chrome browser process via Sandbox IPC and Chromium IPC
- Chrome renderer process via Chromium IPC
- Flash broker process via Simple IPC

API calls are intercepted in the Flash plugin process and are forwarded to the Chrome browser process via the Sandbox IPC connection; these forwarded API calls are first evaluated by the policy engine against the sandbox policies. For NPAPI calls, the Flash plugin process invokes the NPAPI support services exposed by the renderer process. Furthermore, the Flash plugin process also uses the additional services exposed by the Flash broker process.

The diagram below shows how all the sandbox mechanisms are interconnected in Chrome Flash.


Flash Player Protected Mode For Chrome (Chrome Flash)
Chrome Browser Process
(chrome.exe)
API Call
Chrome Sandbox Services
Policy Checks
Policy Engine
PluginProcessHost
RenderProcessHostImpl
Chromium IPC (Browser-Renderer Channel)
RenderThreadImpl
Chrome Renderer Process
(chrome.exe)
[Sandboxed, Untrusted Integrity]
API Call [Sandboxed]
PluginChannelHost
Sandbox IPC (Sandbox Services Channel)
Operating System
Flash Broker Process
(rundll32.exe, gcswf32.dll!BrokerMain)
API Call
Chrome Flash Broker Services
Chromium IPC (Plugin Management Channel)
Chromium IPC (NPAPI Channel)
Simple IPC (Flash Services Channel)
PluginThread
PluginChannel
Flash Plugin Process
(chrome.exe, gcswf32.dll)
[Sandboxed, Low Integrity]
API Call [Sandboxed]

5.7.2. FLASH PLAYER PROTECTED MODE FOR CHROME PEPPER (PEPPER FLASH)

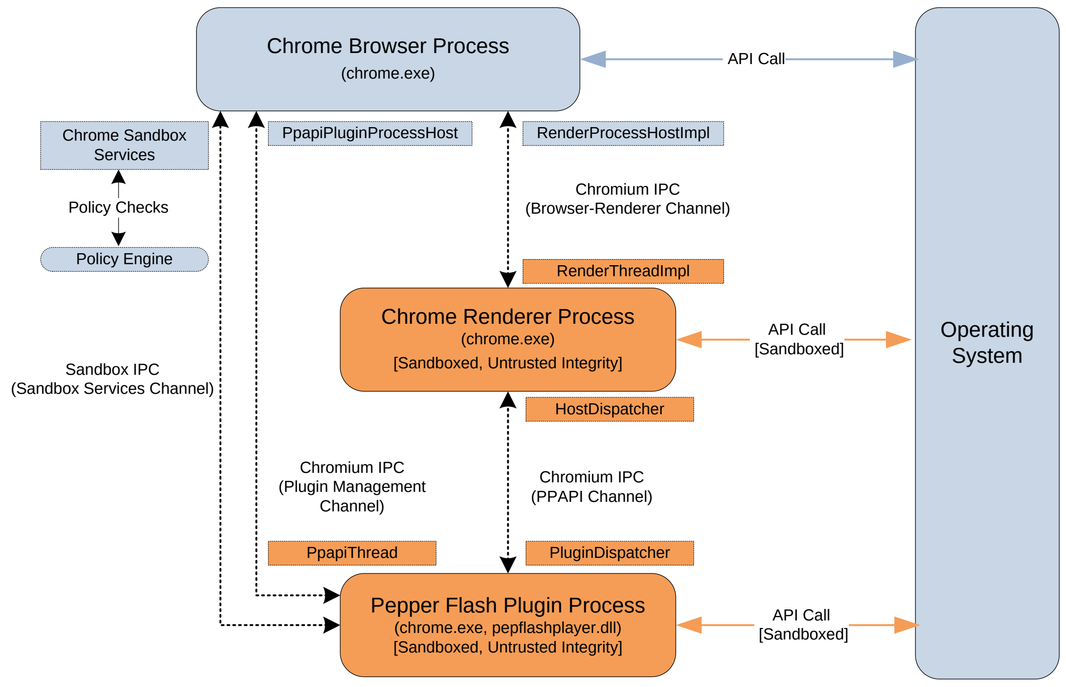
In Pepper Flash, the sandboxed Flash plugin process connects to the following processes:

- Chrome browser process via Sandbox IPC and Chromium IPC
- Chrome renderer process via Chromium IPC

APIs calls are intercepted in the Flash plugin process and are forwarded to the Chrome browser process via the Sandbox IPC connection; these forwarded API calls are first evaluated by the policy engine against the sandbox policies. For PPAPI calls, the Flash plugin process invokes the PPAPI services exposed by the renderer process. Since the PPAPI services provide adequate capabilities to the Flash plugin process, there is no need for an additional Flash broker.

The diagram below shows how all the sandbox mechanisms are interconnected in Pepper Flash.

Flash Player Protected Mode For Chrome Pepper (Pepper Flash)



5.7.3. FLASH PLAYER PROTECTED MODE FOR FIREFOX (FIREFOX FLASH)

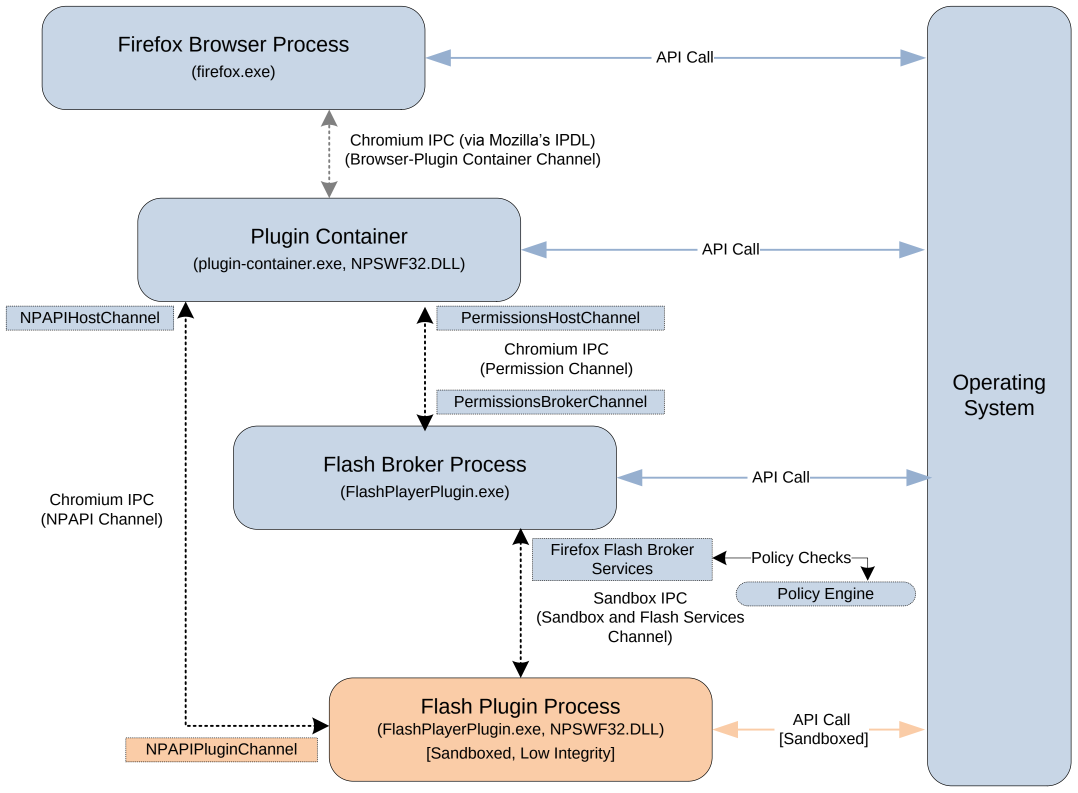
In Firefox Flash, the sandboxed Flash plugin process connects to the following processes:

- Plugin container process via Chromium IPC
- Flash broker process via Sandbox IPC

APIs calls are intercepted in the Flash plugin process and are forwarded to the Flash broker process via the Sandbox IPC connection; these forwarded API calls are first evaluated by the policy engine against the sandbox policies. In addition to servicing forwarded API calls, the Flash broker process also exposes additional services to the Flash plugin process. Furthermore, the Flash broker process additionally exposes permission services which grant/deny the sandboxed Flash plugin process access to resources. These permission services are exposed by the Flash broker process to the plugin container process. And for NPAPI calls, the Flash plugin process invokes the NPAPI services exposed by the plugin container process which in turn proxies the NPAPI calls to the browser process.

The diagram below shows how all the sandbox mechanisms are interconnected in Firefox Flash.

Flash Player Protected Mode For Firefox (Firefox Flash)



6. SANDBOX LIMITATIONS

In this section, we will discuss the limitations and weaknesses of the three sandbox implementations. We will focus on the things that malicious code can still do when running inside the sandbox, in spite of the sandbox restrictions.

6.1. FILE SYSTEM READ ACCESS

Firefox Flash allows read access to all files that are accessible from the user's account. This is partly a result of the sandbox process token still having access to some files (such as those accessible to the *Everyone* and *Users* group), but more importantly, there is a hard-coded policy rule that allows read access to all files:

```
SubSystem=SUBSYS_FILES
Semantics=FILES_ALLOW_READONLY
Pattern="*"
```

We can assume that above policy rule was added for compatibility reasons. However, the security implication of this weakness is that it would allow malicious code running in the sandbox to read the user's documents, source codes, application configuration/data files (which may contain encrypted password or password hashes), and other sensitive files.

This weakness also allows an attacker to read the policy file "*ProtectedModeWhitelistConfig.txt*", which contains user configured custom policies. This can give attackers an idea of what exceptions from the sandbox restrictions are in effect, and may allow them to craft more effective attacks subsequently.

Chrome Flash also allows read access as much as Firefox Flash does, not by any explicit policy rules but due to the sandbox process token still having read access to files.

In contrast to the other two implementations, Pepper Flash does not allow any direct file access at all.

6.2. REGISTRY READ ACCESS

Firefox Flash also allows read access to registry keys that are accessible from the user's account. This is partly a result of the sandbox process token still having access to some registry keys (such as those accessible to the *Everyone* and *Users* group), but more importantly, there are several hard-coded policy rules that allow read access to major registry hives:

```
SubSystem=SUBSYS_REGISTRY
Semantics=REG_ALLOW_READONLY
Pattern="HKEY_CLASSES_ROOT*"

SubSystem=SUBSYS_REGISTRY
Semantics=REG_ALLOW_READONLY
Pattern="HKEY_CURRENT_USER*"

SubSystem=SUBSYS_REGISTRY
Semantics=REG_ALLOW_READONLY
Pattern="HKEY_LOCAL_MACHINE*"

SubSystem=SUBSYS_REGISTRY
Semantics=REG_ALLOW_READONLY
Pattern="HKEY_USERS*"

SubSystem=SUBSYS_REGISTRY
```

```
Semantics=REG_ALLOW_READONLY
Pattern="HKEY_CURRENT_CONFIG*"
```

Again, we can speculate that the above policies were added for compatibility reasons. This weakness would allow malicious code running in the sandbox to read system configuration information, get a list of installed applications, and retrieve application configuration/data (which may contain encrypted passwords or password hashes) and other sensitive information. In addition to the policies above, Firefox Flash also allows full access to certain Flash related registry keys. For example, full access is granted for any registry access to a key that matches the pattern *HKEY_CURRENT_USER\Software\Macromedia\FlashPlayer**.

Chrome Flash also allows read access to the major registry hives mentioned above. However, it does not explicitly allow any other registry access.

Pepper Flash on the other hand does not allow any registry access at all.

6.3. NETWORK ACCESS

A notable limitation in both Chrome Flash and Firefox Flash is their inability to restrict network access. This limitation would allow malicious code running in the sandbox process to send stolen information to a remote server. Another way to leverage this limitation is by connecting to and possibly exploiting internal systems which are accessible to the affected machine but are otherwise inaccessible if accessed from outside the internal network. This type of attack was already known in the pre-sandbox days, but it is interesting that it is still possible. Pepper Flash doesn’t have this limitation as it disallows socket creation.

6.4. POLICY ALLOWED WRITE ACCESS TO FILES/FOLDERS

A weakness we found researching Firefox Flash are the permissive policy rules that grant the sandbox process write access to certain folders and files, some of which are used by third party applications. If the writable folder/file is used by a certain application to store configuration information, it may be possible for malicious code running in the sandbox process to control the behavior of the application, which in turn could lead to further compromise of the system.

The ability to create a local file with a controllable file name can also be leveraged by an attacker in the following use-cases:

- Can be leveraged in exploiting a vulnerability in which successful exploitation requires the creation an attacker-controlled file with a predictable file name.
- For multi-stage exploit payloads, an attacker can use the writable locations as a location to temporarily store a second stage payload library file which can then be loaded to the sandbox process.

6.5. CLIPBOARD READ/WRITE ACCESS

In Firefox Flash, programmatic read/write access is permitted on the clipboard which could be abused. This is because in addition to the clipboard read/write access restriction not being placed in the job object the sandbox process is assigned to, the *SandboxClipboardDispatcher* dispatcher class in the broker process also provides clipboard-related services which allow clipboard access in the context of the broker process.

Chrome Flash also allows read/write access to the clipboard because no clipboard read/write access restriction is placed on the sandbox’s job object.

Thus, for these two implementations it is possible for code running in the sandbox process to read the clipboard contents, which may contain sensitive information (such as passwords – if the user uses an insecure password manager that does not regularly clears the clipboard). Clipboard write access may also lead to other security issues such as arbitrary command injection, and if an application trusts the clipboard contents, it could also become an avenue for a sandbox escape - these are described by Tom Keetch in the paper "*Practical Sandboxing on the Windows Platform*" [18].

As usual, Pepper Flash does not have this weakness.

6.6. WRITE ACCESS TO FAT/FAT32 PARTITIONS

Common to all three implementations is that they all allow write access to FAT/FAT32 partitions. Since FAT/FAT32 partitions (still mostly used in USB flash drives) do not support security descriptors, it is possible for code running in the sandbox process to create or modify files located in partitions of these types. As a consequence, malicious code running in the sandbox process will be able to drop malicious files on FAT/FAT32 partitions which could in turn lead to propagation behaviors. An example of such propagation behavior is dropping a malicious EXE file and an autorun.inf file.

6.7. SANDBOX LIMITATION COMPARISON TABLE

Here's a comparison table for each sandbox's restrictions against some important objects:

	Chrome Flash	Firefox Flash	Pepper Flash
File System READ Access (%USERPROFILE%)	**GRANTED**	**GRANTED**	**DENIED**
File System READ Access (%USERPROFILE%\Documents)	**GRANTED**	**GRANTED**	**DENIED**
File System READ Access (%APPDATA%)	**GRANTED**	**GRANTED**	**DENIED**
File System READ Access (%LOCALAPPDATA%)	**GRANTED**	**GRANTED**	**DENIED**
File System READ Access (%LOCALAPPDATA%\Temp)	**GRANTED**	**GRANTED**	**DENIED**
File System READ Access (%USERPROFILE%\AppData\LocalLow)	**GRANTED**	**GRANTED**	**DENIED**
File System READ Access (%SystemDrive%)	**GRANTED**	**GRANTED**	**DENIED**
File System READ Access (%SystemDrive%\Program Files)	**GRANTED**	**GRANTED**	**DENIED**
File System READ Access (%SystemDrive%\Program Files (x86))	**GRANTED**	**GRANTED**	**DENIED**
File System READ Access (%SystemRoot%)	**GRANTED**	**GRANTED**	**DENIED**
File System WRITE Access (%USERPROFILE%)	**DENIED**	**DENIED**	**DENIED**
File System WRITE Access (%USERPROFILE%\Documents)	**DENIED**	**DENIED**	**DENIED**
File System WRITE Access (%APPDATA%)	**DENIED**	**DENIED**	**DENIED**
File System WRITE Access	**DENIED**	**DENIED**	**DENIED**

(%LOCALAPPDATA%)			
File System WRITE Access (%LOCALAPPDATA%\Temp)	**DENIED**	**DENIED**	**DENIED**
File System WRITE Access (%USERPROFILE%\AppData\LocalLow)	**GRANTED**	**DENIED**	**DENIED**
File System WRITE Access (%SystemDrive%)	**DENIED**	**DENIED**	**DENIED**
File System WRITE Access (%SystemDrive%\Program Files)	**DENIED**	**DENIED**	**DENIED**
File System WRITE Access (%SystemDrive%\Program Files (x86))	**DENIED**	**DENIED**	**DENIED**
File System WRITE Test (%SystemRoot%)	**DENIED**	**DENIED**	**DENIED**
Registry READ Access (HKEY_CLASSES_ROOT)	**GRANTED**	**GRANTED**	**DENIED**
Registry READ Access (HKEY_CURRENT_USER)	**GRANTED**	**GRANTED**	**DENIED**
Registry READ Access (HKEY_LOCAL_MACHINE)	**GRANTED**	**GRANTED**	**DENIED**
Registry READ Access (HKEY_USERS)	**GRANTED**	**GRANTED**	**DENIED**
Registry READ Access (HKEY_CURRENT_CONFIG)	**GRANTED**	**GRANTED**	**DENIED**
Registry WRITE Access (HKEY_CLASSES_ROOT)	**DENIED**	**DENIED**	**DENIED**
Registry WRITE Access (HKEY_CURRENT_USER)	**DENIED**	**DENIED**	**DENIED**
Registry WRITE Access (HKEY_LOCAL_MACHINE)	**DENIED**	**DENIED**	**DENIED**
Registry WRITE Access (HKEY_USERS)	**DENIED**	**DENIED**	**DENIED**
Registry READ Access (HKEY_CURRENT_CONFIG)	**DENIED**	**DENIED**	**DENIED**
Network Access	**GRANTED**	**GRANTED**	**DENIED**
Clipboard READ Access	**GRANTED**	**GRANTED**	**DENIED**
Clipboard WRITE Access	**GRANTED**	**GRANTED**	**DENIED**
BaseNamedObjects WRITE Access	**GRANTED**	**GRANTED**	**DENIED**

6.8. SUMMARY: SANDBOX LIMITATIONS

In this section we showed the various limitations and weaknesses of each sandbox. We can also safely conclude that among the three implementations, Pepper Flash offers the most restriction.

While code running in the Flash sandboxes is severely limited in terms of what it can do, it is still possible to carry out information theft attacks by leveraging their current weaknesses and limitations. This is an important point to remember because the result of an information theft attack can be devastating especially for businesses and governments. And by conveying the security implication of each limitation or weakness, we hope that we can create an awareness that users should still continue to be cautious when opening unsolicited documents even if the application they are using has sandboxing capabilities.

7. SANDBOX ESCAPE

In this section, we will discuss the different ways that malicious code might use to escape the Flash sandboxes. This section starts with a discussion of generic escape methods and then moves to the discussion of attacking specific sandbox mechanisms to conduct a sandbox escape. Furthermore, most of the items discussed in this section are not specific to the Flash sandbox and can be applied to other sandbox implementations as well.

As a side note, one important advantage for code already running in the sandbox process is that it already has the necessary information to perform a Data Execution Prevention (*DEP*) and Address Space Layout Randomization (*ASLR*) bypass when exploiting other applications running on the same system. The reason is that a system-wide value called the *image bias* which dictates the load address of DLLs is shared across processes and is computed only once per boot [19]. This means, that for example, code running in the sandboxed Flash plugin process can use the *NTDLL.DLL* and *KERNEL32.DLL* base in the sandboxed process when crafting a ROP sequence for the exploit to be used against a higher-privileged process.

7.1. LOCAL ELEVATION OF PRIVILEGE (EOP) VULNERABILITIES

The first option when performing a sandbox escape is by exploiting local elevation of privilege (EoP) vulnerabilities. Exploiting local EoP vulnerabilities, especially those can result in arbitrary code execution in kernel mode are an ideal way to bypass all the restrictions set on sandboxed code.

With multiple available interfaces to kernel-mode code such as system calls and Device objects which are accessible to a sandboxed process, we can expect that local EoP vulnerabilities will become more valuable as more and more critical applications are being sandboxed. An interesting discussion on kernel-mode vulnerabilities can be found in the presentation "*There's a party at Ring0, and you're invited*" [20] by Tavis Ormandy and Julien Tinnes.

7.2. NAMED OBJECT SQUATTING ATTACKS

Named object squatting is an attack involving a malicious application creating a named object that a target application is known to trust. In the context of a sandbox escape, code running in a compromised sandbox process can create a malicious named object and wait until a higher-privileged process uses the malicious named object.

An example of named object squatting is an attack against Protected Mode Internet Explorer discussed by Tom Keetch in his presentation "*Practical Sandboxing on the Windows Platform*" [18].

7.3. IPC MESSAGE PARSER VULNERABILITIES

Running in a privileged context and being the first code that touches potentially untrusted data from an IPC connection, the IPC message parser is one of the primary attack vectors in a sandbox implementation for conducting a sandbox escape. From an auditing perspective, the routines of interest are those that parse IPC message and those that deserialize IPC call parameters (especially complex parameters types).

In section 5.4, we had discussed the different IPC implementations used by the different Flash sandboxes, all of them are open source, and thus, a source audit is possible. An interesting exercise is looking at security updates made in the public sources and then checking whether or not those security updates are propagated to the binaries released by Adobe – this also applies to other sandbox mechanisms as well.

An example of IPC message parser vulnerability is the *SkBitmap* deserialization vulnerability discovered by Mark Dowd [14] in the Chrome sandbox.

7.4. POLICY VULNERABILITIES

Permissive policies, particularly those that allow write access to a resource can also be used as a vector for sandbox escape.

An example scenario would be write-allowed policies for registry keys – if a particular key has sub keys or entries that are used for sandbox configuration, the sandboxed code can potentially control the initialization of the sandbox and use it directly or indirectly to perform a sandbox escape. The same applies to write-allowed policies for files or folders – if a particular folder contains a configuration file used by a higher-privileged process, the behavior of the higher-privileged process can possibly be controlled by a sandboxed code and could possibly lead to a sandbox escape. Also, if a particular folder contains executable files, a sandboxed code can potentially overwrite the contents of the executable file so that when the executable file is spawned, an attacker-controlled code will be executed in a privileged context.

Auditing for policy vulnerabilities involves enumerating all the policies in a sandbox implementation and specifically looking for policies that allow write access. Once all write allowed policies are enumerated, each write policy is then audited for potential escape by checking if the writable resource can affect the behavior of a higher-privileged application.

7.5. POLICY ENGINE VULNERABILITIES

Acting as a gatekeeper who decides which potentially sensitive actions are allowed or disallowed, the policy engine is another sandbox mechanism that can be leveraged to conduct a sandbox escape.

Auditing for policy engine vulnerabilities involves looking for ways to bypass policy checks such as checking if there is lack of or insufficient canonicalization of resource names when performing policy checks. Additionally, auditing for memory corruption bugs may also yield results if there is complex preprocessing of resource names during policy checks.

An excellent example of a policy engine vulnerability is CVE-2011-1353 [21, 22], a vulnerability we discovered (and also independently discovered by Zhenhua Liu [23] of Fortinet's Fortiguard Labs) in the policy engine of the Adobe Reader X sandbox. The vulnerability (since patched) is due to lack of canonicalization of a resource name when evaluating registry deny-access policies. In Reader X, there is a policy allowing full access to the sub keys of the registry key *HKEY_CURRENT_USER\Software\Adobe\Acrobat Reader\10.0*:

```
SubSystem: SUBSYS_REGISTRY
Semantics: REG_ALLOW_ANY
Pattern: HKEY_CURRENT_USER\Software\Adobe\Acrobat Reader\10.0\*
```

However, one of the sub keys of the previously mentioned registry key is used by Reader X to store sandbox configuration. An example is the registry entry *bProtectedMode* under the *Privileged* subkey which is used by Reader X to determine whether to enable or disable the Reader X sandbox:

```
HKEY_CURRENT_USER\Software\Adobe\Acrobat Reader\10.0\Privileged
bProtectedMode = 0 (disabled), non-zero (enabled)
```

Accordingly, a registry deny-access policy also exists to prevent access to the *Privileged* sub key:

```
SubSystem: SUBSYS_REGISTRY
```

```
Semantics: REG_DENY
Pattern: HKEY_CURRENT_USER\Software\Adobe\Acrobat Reader\10.0\Privileged*
```

However, since there is lack of canonicalization of registry resource name when performing policy checks, simply modifying the registry resource name to contain additional back slashes such as:

```
HKEY_CURRENT_USER\Software\Adobe\Acrobat Reader\10.0\\Privileged
```

Will result in a bypass of the registry deny-access policy, thereby, allowing a sandboxed code to disable the Reader X sandbox.

7.6. SERVICE VULNERABILITIES

In the Services section (5.5), we discussed the different services exposed by different Flash sandbox processes. The services exposed by higher-privileged processes make up a large part of the attack surface in a sandbox implementation since they are many and run in a privileged context.

Two examples of service vulnerability in a Flash sandbox implementation is CVE-2012-0724 and CVE-2012-0725 [24, 25], two vulnerabilities we discovered (and also independently discovered by Fermin J. Serna of the Google Security Team). Both vulnerabilities exist in the Chrome Flash Broker and are similar in nature.

CVE-2012-0724 and CVE-2012-0725 exist in two service handlers which are part of the Chrome Flash Broker Services (5.5.3) and are callable to the sandboxed Flash plugin process via Simple IPC (5.4.3). Essentially, the vulnerabilities were due to the service handlers accepting and fully trusting a pointer they received from the sandboxed Flash plugin process. The vulnerable service handlers including their corresponding Simple IPC message ID and parameters are as follows:

Message ID	Parameters	Purpose
0x2B	VOIDPTR sec_func_table	Broker a call to *AcquireCredentialHandlesA()*
0x2D	VOIDPTR sec_func_table, ULONG32 cred_handle_lower, ULONG32 cred_handle_upper	Broker a call to *FreeCredentialsHandle()*

Notice that in both service handlers, the first parameter is a pointer (*sec_func_table)*, and specifically, a pointer to a *SecurityFunctionTableA* [26] structure. Inside the service handlers, *sec_func_table* is fully trusted (no checks or further transformation is done) and is dereferenced to invoke *SecurityFunctionTableA.AcquireCredentialsHandleA()* and *SecurityFunctionTableA.FreeCredentialsHandle()*:

```
Service_0x2B_AcquireCredentialsHandleA:
   ...
   mov   reg, [sec_func_table]   ; sec_func_table is fully controllable
   ...
   call  [reg+0Ch]               ; sec_func_table->AcquireCredentialsHandleA()
                                 ; reg+0Ch is fully controllable!
```

```
Service_0x2D_FreeCredentialsHandle:
   ...
   mov   reg, [sec_func_table]  ; sec_func_table is fully controllable
   call  [reg+10h]              ; sec_func_table->FreeCredentialsHandle()
                                ; reg+10h is fully controllable!
```

Since *sec_func_table* is fully controllable, the address used in the call instruction is also fully controllable. With additional research work of figuring out how to implant data in the stack of the Flash broker process, and using one of the vulnerabilities to leak the stack address of the Flash broker to the sandboxed Flash plugin, and finally, with the possibility of predicting the address of APIs in the Flash broker process (as explained in the *image bias* side note), we were able to achieve arbitrary code execution in the Chrome Flash broker process.

We expect that in the future, to accommodate new features, new services will continually need to be added to accommodate the new functionalities.

7.7. SUMMARY: SANDBOX ESCAPE

Sandbox escape involves a sandboxed code exploiting a weakness in a higher-privileged process. The higher-privileged process can be a process which is a part of the operating system, another application or part of the sandbox implementation itself. Two prime examples of weaknesses that can lead to a sandbox escape are setting of permissive policies and improper handling of untrusted data provided by the sandboxed code.

As sandboxes becomes more prevalent in critical applications, an attacker would now need a separate vulnerability, or in some cases, a few more vulnerabilities [27, 28] to execute code in an elevated privilege in order to install persistent malware in a target system. That being said, sandbox escape vulnerabilities will become more important and exploiting them will become a major part of sophisticated attacks targeting critical applications.

8. CONCLUSION

In this paper we discussed three different implementations of the Flash Player sandbox. We have concluded that out of the three implementations, the one that offers the most security is Pepper Flash. As the current implementation still in the experimental phase, it still remains to be seen how well it would fare in the real world with its extreme restrictions. In our experience, it is still not stable enough for day to day use (e.g. YouTube videos). Fortunately, even the less restrictive Firefox Flash and Chrome Flash still offers substantial cost of exploitation. In fact, we haven't encountered any public exploits that fully exploits a Flash vulnerability through Firefox and Chrome since these sandbox implementations were released.

However, it does not mean that users should be complacent. As we have shown in this paper, there are still ways in which an attacker or malicious code can do damage in spite of running within the sandbox restrictions, and also that there are still many ways in which a determined attacker can possibly discover how to escape or bypass the sandboxes. We hope that through the discussion of the internal mechanisms, limitations, and weaknesses of these sandbox implementations, we have increased the reader's awareness regarding this topic.

Finally, we hope that this paper will serve as an inspiration and useful resource for our fellow security researchers for digging more deeply into current and future sandbox implementations.

9. ACKNOWLEDGEMENTS

We would like to thank Robert Freeman, Justin Schuh, and Peleus Uhley for reviewing and giving feedback on this paper.

10. BIBLIOGRAPHY

[1] Mozilla Foundation, “NPAPI,” [Online]. Available: https://wiki.mozilla.org/NPAPI.

[2] P. Sabanal and M. V. Yason, “Playing In The Reader X Sandbox,” [Online]. Available: https://media.blackhat.com/bh-us-11/Sabanal/BH_US_11_SabanalYason_Readerx_WP.pdf.

[3] The Chromium Authors, “Pepper Plugin Implementation,” [Online]. Available: https://sites.google.com/a/chromium.org/dev/developers/design-documents/pepper-plugin-implementation.

[4] Adobe Systems Incorporated, "Adobe Flash Player 11.3 Administration Guide," [Online]. Available: http://wwwimages.adobe.com/www.adobe.com/content/dam/Adobe/en/devnet/flashplayer/pdfs/flash_player_11_3_admin_guide.pdf.

[5] D. LeBlanc, “Practical Windows Sandboxing – Part 1,” [Online]. Available: http://blogs.msdn.com/b/david_leblanc/archive/2007/07/27/practical-windows-sandboxing-part-1.aspx.

[6] D. LeBlanc, “Practical Windows Sandboxing, Part 2,” [Online]. Available: http://blogs.msdn.com/b/david_leblanc/archive/2007/07/30/practical-windows-sandboxing-part-2.aspx.

[7] D. LeBlanc, “Practical Windows Sandboxing – Part 3,” [Online]. Available: http://blogs.msdn.com/b/david_leblanc/archive/2007/07/31/practical-windows-sandboxing-part-3.aspx.

[8] Microsoft, “Restricted Tokens,” [Online]. Available: http://msdn.microsoft.com/en-us/library/windows/desktop/aa379316(v=vs.85).aspx.

[9] Microsoft, "Windows Integrity Mechanism Design," [Online]. Available: http://msdn.microsoft.com/en-us/library/bb625963.aspx.

[10] Wikipedia, “Shatter Attack,” [Online]. Available: http://en.wikipedia.org/wiki/Shatter_attack.

[11] Microsoft, “Job Objects,” [Online]. Available: http://msdn.microsoft.com/en-us/library/windows/desktop/ms684161(v=vs.85).aspx.

[12] L. McQuarrie, A. Mehra, S. Mishra, K. Randolph and B. Rogers, "Inside Adobe Reader Protected Mode – Part 2 – The Sandbox Process," [Online]. Available: http://blogs.adobe.com/asset/2010/10/inside-adobe-reader-protected-mode-%E2%80%93-part-2-%E2%80%93-the-sandbox-process.html.

[13] The Chromium Authors, “Inter-process Communication (IPC),” [Online]. Available: http://www.chromium.org/developers/design-documents/inter-process-communication.

[14] Azimuth Security Pty Ltd, “The Chrome Sandbox Part 2 of 3: The IPC Framework,” [Online]. Available: http://blog.azimuthsecurity.com/2010/08/chrome-sandbox-part-2-of-3-ipc.html.

[15] Mozilla Foundation, “NPObject,” [Online]. Available: https://developer.mozilla.org/en/NPObject.

[16] Mozilla Foundation, "Inter-process communication Protocol Definition Language (IPDL)," [Online]. Available: https://developer.mozilla.org/en/IPDL.

[17] Mozilla Foundation, “Electrolysis Project,” [Online]. Available: https://wiki.mozilla.org/Electrolysis.

[18] T. Keetch, “Practical Sandboxing on the Windows Platform,” [Online]. Available: http://www.tkeetch.co.uk/slides/HackInParis_2011_Keetch_-_Practical_Sandboxing.ppt.

[19] M. Russinovich, D. Solomon and A. Ionescu, Windows® Internals: Including Windows Server 2008 and Windows Vista, Fifth Edition.

[20] T. Ormandy and J. Tinnes, “There's a party at Ring0, and you're invited,” [Online]. Available: http://www.cr0.org/paper/to-jt-party-at-ring0.pdf.

[21] IBM Corporation, “Adobe Reader X Sandbox Bypass Vulnerability,” [Online]. Available: http://www.iss.net/threats/433.html.

[22] Adobe Systems Incorporated, “Security updates available for Adobe Reader and Acrobat (APSB11-24),” [Online]. Available: http://www.adobe.com/support/security/bulletins/apsb11-24.html.

[23] Z. Liu and G. Lovet, "Breeding Sandworms: How To Fuzz Your Way Out of Adobe Reader's Sandbox," [Online]. Available: https://media.blackhat.com/bh-eu-12/Liu_Lovet/bh-eu-12-Liu_Lovet-Sandworms-WP.pdf.

[24] Adobe Systems Incorporated, “Security update available for Adobe Flash Player (APSB12-07),” [Online]. Available: http://www.adobe.com/support/security/bulletins/apsb12-07.html.

[25] IBM Corporation, “Adobe Flash Player For Chrome Sandbox Bypass Vulnerabilities,” [Online]. Available: http://iss.net/threats/446.html.

[26] Microsoft, “SecurityFunctionTable structure,” [Online]. Available: http://msdn.microsoft.com/en-us/library/windows/desktop/aa380125(v=vs.85).aspx.

[27] J. L. Obes and J. Schuh, "A Tale of Two Pwnies (Part 1)," [Online]. Available: http://blog.chromium.org/2012/05/tale-of-two-pwnies-part-1.html.

[28] K. Buchanan, C. Evans, C. Reis and T. Sepez, "A Tale Of Two Pwnies (Part 2)," [Online]. Available: http://blog.chromium.org/2012/06/tale-of-two-pwnies-part-2.html.

11. APPENDIX A: EVICTED DLLS AND PLUGINS

11.1. EVICTED DLLS IN FIREFOX FLASH

(As of version 11.3.300.257)

adialhk.dll
acpiz.dll
avgrsstx.dll
babylonchromepi.dll
btkeyind.dll
cmcsyshk.dll
cooliris.dll
dockshellhook.dll
googledesktopnetwork3.dll
fwhook.dll
hookprocesscreation.dll
hookterminateapis.dll
hookprintapis.dll
imon.dll
ioloHL.dll
kloehk.dll
lawenforcer.dll
libdivx.dll
lvprcinj01.dll
madchook.dll
mdnsnsp.dll
moonsysh.dll
npdivx32.dll
npggNT.des
npggNT.dll
oawatch.dll
pavhook.dll
pavshook.dll
pavshookwow.dll
pctavhook.dll
pctgmhk.dll
prntrack.dll
radhslib.dll
radprlib.dll
rapportnikko.dll
rlhook.dll
rooksdol.dll
rpchromebrowserrecordhelper.dll
rpmainbrowserrecordplugin.dll
r3hook.dll
sahook.dll
sbrige.dll
sc2hook.dll

sguard.dll
smum32.dll
smumhook.dll
ssldivx.dll
syncor11.dll
systools.dll
tfwah.dll
ycwebcamerasource.ax
wblind.dll
wbhelp.dll
winstylerthemehelper.dll

11.2. Evicted DLLs In Chrome Flash and Pepper Flash

(As of version 20.0.1132.43)

adialhk.dll
acpiz.dll
avgrsstx.dll
babylonchromepi.dll
btkeyind.dll
cmcsyshk.dll
cmsetac.dll
cooliris.dll
dockshellhook.dll
easyhook32.dll
googledesktopnetwork3.dll
fwhook.dll
hookprocesscreation.dll
hookterminateapis.dll
hookprintapis.dll
imon.dll
ioloHL.dll
kloehk.dll
lawenforcer.dll
libdivx.dll
lvprcinj01.dll
madchook.dll
mdnsnsp.dll
moonsysh.dll
mpk.dll
npdivx32.dll
npggNT.des
npggNT.dll
oawatch.dll
owexplorer-10513.dll
owexplorer-10514.dll
owexplorer-10515.dll
owexplorer-10516.dll
owexplorer-10517.dll

owexplorer-10518.dll
owexplorer-10519.dll
owexplorer-10520.dll
owexplorer-10521.dll
owexplorer-10522.dll
owexplorer-10523.dll
pavhook.dll
pavlsphook.dll
pavshook.dll
pavshookwow.dll
pctavhook.dll
pctgmhk.dll
prntrack.dll
protector.dll
radhslib.dll
radprlib.dll
rapportnikko.dll
rlhook.dll
rooksdol.dll
rpchromebrowserrecordhelper.dll
r3hook.dll
sahook.dll
sbrige.dll
sc2hook.dll
sdhook32.dll
sguard.dll
smum32.dll
smumhook.dll
ssldivx.dll
syncor11.dll
systools.dll
tfwah.dll
wblind.dll
wbhelp.dll
winstylerthemehelper.dll

11.3. EVICTED PLUGIN DLLS IN CHROME FLASH

rpmainbrowserrecordplugin.dll
rpchromebrowserrecordhelper.dll
rpchrome10browserrecordhelper.dll
ycwebcamerasource.ax
CLRGL.ax