PLAYING IN THE READER X SANDBOX

Paul Sabanal
IBM X-Force Advanced Research
sabanap[at]ph.ibm.com, polsab78[at]gmail.com
@polsab

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

ABSTRACT

In an effort to mitigate the effects of successful exploitation of Adobe Reader vulnerabilities, Adobe announced Adobe Reader Protected Mode back in July 2010. Since its release on November 2010, very little in-depth technical information is available about how the Adobe Reader Protected Mode sandbox works and how it was implemented.

The first part of this talk attempts to close this information gap by diving deep into the implementation details of the Adobe Reader Protected Mode sandbox. We will discuss the results of our reversing efforts to understand the mechanisms and data structures that make up the sandbox.

Using the knowledge gained in the first part, the second part then focuses on the security of the Adobe Reader Protected Mode sandbox. First, we will discuss the limitations and weaknesses of its earlier releases and their security implications, then we will discuss possible avenues to achieve privilege escalation.

At the end of our talk, we will demonstrate how an attacker could leverage the limitations and weaknesses of the Adobe Reader Protected Mode sandbox to carry out information theft or corporate espionage. We will be demonstrating a proof-of-concept information stealing exploit payload bootstrapped by exploiting a publicly known Adobe Reader X vulnerability.

1. CONTENTS

2. INTRODUCTION

Since its release last year, the Adobe Reader X sandbox has been successful in mitigating exploitation attempts. Judging from the number of exploits affecting it so far - zero, there is no doubt that it is a success. It even allowed Adobe to buy some time in fixing vulnerabilities in Reader X. The Reader X sandbox has definitely impacted the Reader exploitation landscape, and for the better. As proof, none of the two 0-day exploits that came out this year that could affect Reader X can successfully exploit it, as currently there is no publicly known method to escape the sandbox.

Having said that, we wondered what the security implications are with this new technology and if there are still some things an attacker could do in spite of the restrictions imposed by the sandbox. So, how does this sandbox differ from other ones? 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?

These are the core questions that we asked ourselves and we will answer in this paper. Along the way, we will discuss the nature of the sandbox internals, limitations and potential escapes. Furthermore, we will also provide our thoughts and recommendations on the matter of sandbox security.

3. SANDBOX INTERNALS

Before we discuss Reader X's sandbox security, we need to understand how it is implemented. In this section we will discuss the results of our reversing efforts. Please note that we are using the latest version of Adobe Reader X as of this writing - 10.1.0.

3.1. Relationship with Google Chrome's Sandbox

We knew that the Reader X sandbox is based on Google's Chrome sandbox, but we didn't know to what extent. Is it just the design, or are there code shared between them? As Chrome (or to be more specific the project it is based on, Chromium) is open-source, we thought it would be interesting to see if Reader X reuses code from Chrome.

We built a release version of Chrome with debugging symbols and, using a combination of binary diffing tools such as the patchdiff2 IDA Pro plugin, homegrown scripts, and manual analysis, we managed to map Chrome's sandbox-related code to similar code in the Acrord32.exe binary.

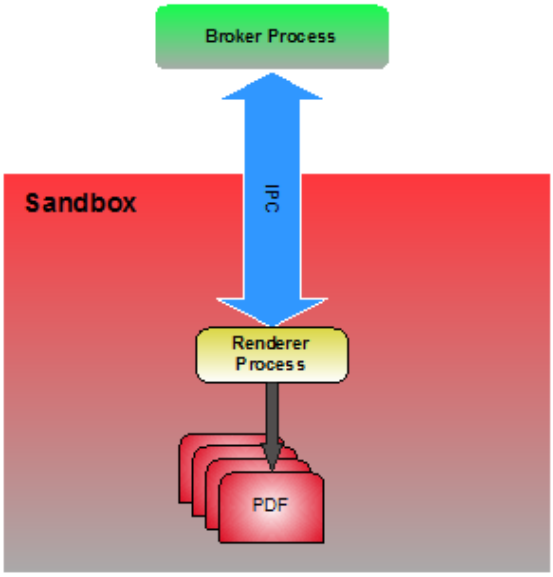
As it turns out, a lot of the sandbox-related code in Chromium is reused in Reader X. For instance, out of the 291 functions under the "sandbox" namespace found in the chrome.exe binary, we found 276 similar functions in the AcroRd32.exe binary. The rest are either heavily modified that we couldn't match them, or not implemented in Reader X at all. We then ported the names of the matched functions from our Chrome IDB to our AcroRd32.exe IDB.

This tremendously helped in reversing the Reader X sandbox and allowed us to look at the differences in the implementation between Chrome and Reader X to get a big picture of the sandbox implementation, as opposed to reversing everything from scratch.

Another technique that was useful in our reversing efforts is a tool we developed in house to reconstruct C++ objects at runtime. The tool was built using the PIN dynamic binary instrumentation tool and allows us to identify virtual function call targets and the object/class that they belong to.

3.2. Sandbox Architecture

The Reader X sandbox configuration consists of two main components, the broker process and the sandbox process. The broker is the first process executed when starting up Reader, and is responsible for setting up the restrictions and policies, and also for spawning the sandbox process. It is also responsible for intercepting API calls from the sandbox process and evaluates these calls against the sandbox policy. It does this by hosting an IPC service to communicate with the sandbox process. The sandbox process is responsible for parsing and rendering the PDF file. In the following sections we will go into more detail about each component of the sandbox.



READER X SANDBOX ARCHITECTURE

3.3. Sandbox Mechanisms

3.3.1. Sandbox Restrictions

Based on the Practical Windows Sandboxing recipe (1; 2), Reader X leverages the following Windows mechanisms for restricting the privilege of the sandbox process:

1. Restricted Tokens
2. Windows Integrity Mechanism (for Windows Vista and later)
3. Job Objects

As noted by Adobe, Reader X currently does not use a separate desktop (3) for the sandbox process because it would require significant changes and would delay the of the release of the Reader X sandbox. Instead, the Reader X sandbox attempts to mitigate "shatter" attacks and DLL injection via SetWindowsHookEx() using the other previously listed Windows mechanisms (4).

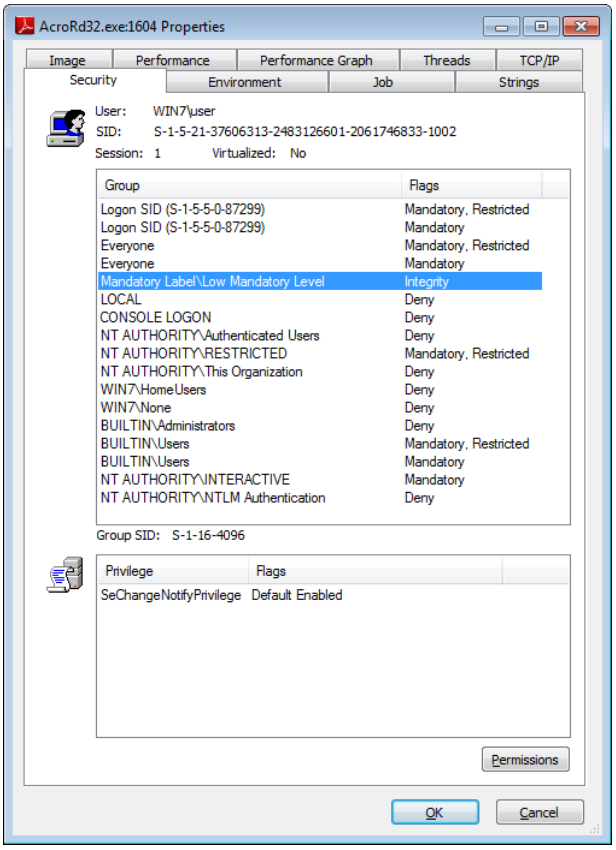
Restricted Tokens

By using restricted tokens (5), the sandbox process is limited to what it can do in terms of accessing securable objects and performing privileged operations. The restricted token assigned to the sandbox process is derived from the user's token and have the following form:

1. Deny-Only Security Identifiers (SIDs)
    - All, except Logon SID, Everyone, BUILTIN\Users, NT AUTHORITY\INTERACTIVE
2. Restricting SIDs
    - Logon SID
    - Everyone
    - BUILTIN\Users
    - NT AUTHORITY\RESTRICTED

3. Disabled Privileges
    - All, except SeChangeNotifyPrivilege

Among other things, the above restricted token effectively prevents the sandbox process from modifying critical system files and registry keys; it also prevents access to files in the %USERPROFILE% folder (e.g. C:\Users\<user>\) which contains private user files. However, compared to the restricted token used by the Chrome sandbox, the restricted token assigned to the Reader X sandbox process is less restrictive since still have access to a number of resources such those accessible to the Everyone and Users group.



SANDBOX PROCESS - RESTRICTED TOKEN AND INTEGRITY LEVEL

Windows Integrity Mechanism

Available in Windows Vista and later systems, the Windows Integrity Mechanism provides a way to limit the permissions of less trustworthy applications (6). An integrity level is assigned to processes and securable objects and is inspected first when an access check is performed (7). Depending on the integrity policy stored in the security descriptor of objects, a lower integrity process will have restricted read or restricted write access to higher integrity objects.

In the case of Reader X, the broker process assigns a Low integrity level to the sandbox process. Since the majority of securable objects in a Windows system are assigned a Medium or a higher integrity level, write access to most securable objects will be denied.

Job Objects

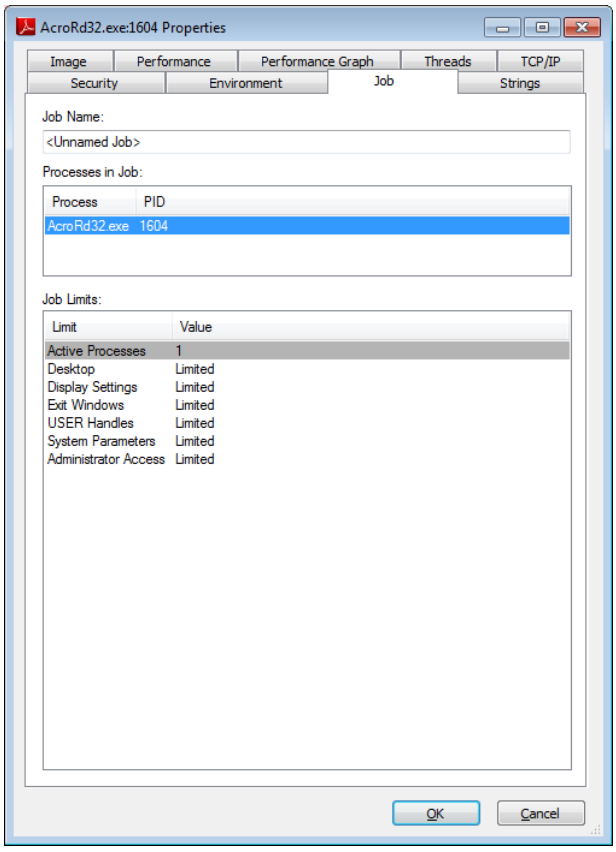
Reader X also enforces additional restrictions to the sandbox process by associating it to a job object (8). In addition to limiting resources to a group of processes, job objects can also restrict the capabilities of a process assigned to them. Below are the restrictions set to the job object which the sandbox process is assigned to:

- Limit the active process to 1 (JOB_OBJECT_LIMIT_ACTIVE_PROCESS).
- All process associated with the job object is terminated when the last handle to the job object is closed (JOB_OBJECT_LIMIT_KILL_ON_JOB_CLOSE).
- Restrict the process from using USER handles (handles to user interface objects) (9) owned by processes not associated with the same job (JOB_OBJECT_UILIMIT_HANDLES).
- Restrict the process from changing display settings via the ChangeDisplaySettings() API (JOB_OBJECT_UILIMIT_DISPLAYSETTINGS).
- Restrict the process from logging off, shutting down or restarting the system via the ExitWindowsEx() API (JOB_OBJECT_UILIMIT_EXITWINDOWS).
- Restrict the process from creating desktops and switching desktops via the CreateDesktop() and SwitchDesktop() API (JOB_OBJECT_UILIMIT_DESKTOP).

- Restrict the process from changing system parameters via the SystemParametersInfo() API (JOB_OBJECT_UILIMIT_SYSTEMPARAMETERS).

In contrast to the job object restrictions set by the Chrome sandbox, the Reader X sandbox currently does not set the following job object restrictions:

- Restrict access to global atoms (JOB_OBJECT_UILIMIT_GLOBALATOMS).
- Restrict read access to clipboard (JOB_OBJECT_UILIMIT_READCLIPBOARD).
- Restrict write access to clipboard (JOB_OBJECT_UILIMIT_WRITECLIPBOARD).



SANDBOX PROCESS - JOB OBJECT RESTRICTIONS

3.3.2. Sandbox Startup Sequence

When a PDF file is opened, the following happens:

1. AcroRd32.exe is spawned. This is the broker process.

2. 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

      Refer to the previous section for more details about the job restrictions.

   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 the previous section for more details about the token restrictions.

c. Set the integrity level. It will be set to INTEGRITY_LEVEL_LOW. Refer to the previous section for more details about integrity levels.

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 ”Appendix C: Evicted DLLs” for the list of evicted DLLs in Reader 10.1.0.

3. The broker process sets up the generic policies. These policies 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 in the Reader install directory.

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

   Refer to section 3.3.6 for more details about the policy engine.

4. The broker process spawns the sandbox process in a suspended state. It will run the ArcoRd32.exe executable, the same as the broker, but with the “type=renderer” parameter.

5. Set up and initialize interceptions on the sandbox process. Refer to section 3.3.3 for more details about the interceptions.

6. Resume execution of the sandbox process.

3.3.3. Interception Manager

After discussing the sandbox restrictions and startup sequence, we will now discuss the major components of the Reader X sandbox. The first component that we will talk about is the Interception Manager.

The purpose of the Interception Manager is to transparently forward API calls made by the sandbox process to the broker process. Generally, an API call is forwarded to the broker process because the API call failed due to the restrictions set on the sandbox process. The broker 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 will generally perform the API call on behalf the sandbox 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 broker process.

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

Finally, the API interception list (list of APIs that will be intercepted) is populated in the SetupService() virtual function of dispatcher classes - see section 3.3.5.1 and “Appendix A: Dispatcher Classes”. Note however, other sandbox initialization routines also populate the API interception list.

3.3.3.1. Interception Types

The following are the different types of API interceptions:

Interception Type	Constant Value
INTERCEPTION_SERVICE_CALL	1
INTERCEPTION_EAT	2
INTERCEPTION_SIDESTEP	3
INTERCEPTION_SMART_SIDESTEP	4
INTERCEPTION_UNLOAD_MODULE	5

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

INTERCEPTION_SERVICE_CALL. Interceptions of type INTERCEPTION_SERVICE_CALL are performed for NTDLL APIs. Interception is performed by the broker process to the sandbox process via WriteProcessMemory() when the sandbox process is newly spawned but still in a suspended state.

On a 32-bit Windows 7 system, INTERCEPTION_SERVICE_CALL interceptions are performed by patching the entry point of the APIs with the following code sequence:

```
MOV EAX,<ServiceID>
MOV EDX,<ThunkCodeAddress>
JMP EDX
RETN <StackSize>
NOP
```

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.

Below is an example of a patched NTDLL.DLL!NtCreateFile():

```
77CA55C8 > B8 42000000      MOV EAX,42
77CA55CD   BA 28000700      MOV EDX,70028
77CA55D2   FFE2             JMP EDX
77CA55D4   C2 2C00          RETN 2C
77CA55D7   90               NOP
```

The original entry point of NTDLL.DLL!NtCreateFile() is follows:

```
77CA55C8 > B8 42000000      MOV EAX,42
77CA55CD   BA 0003FE7F      MOV EDX,7FFE0300
77CA55D2   FF12             CALL DWORD PTR DS:[EDX]
77CA55D4   C2 2C00          RETN 2C
77CA55D7   90               NOP
```

INTERCEPTION_EAT. Interceptions of type INTERCEPTION_EAT are done by the sandbox process to itself and are

performed when the target DLL is mapped into the sandbox process. DLL mapping is monitored via the interception of NTDLL.DLL!NtMapViewOfSection(). And 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. Similar to INTERCEPTION_EAT, interceptions of type INTERCEPTION_SIDESTEP are performed by the sandbox process to itself when the target DLL is mapped into the sandbox process. Interceptions are performed by patching the API entry point with a JMP instruction that transfers control to the thunk code. Below is an example INTERCEPTION_SIDESTEP interception performed to KERNEL32.DLL!CreateProcessA():

```
77B82082 >-E9 E9DF4888      JMP 00010070  ;Jump to thunk code
77B82087   6A 00            PUSH 0
77B82089   FF75 2C          PUSH DWORD PTR SS:[EBP+2C]
77B8208C   FF75 28          PUSH DWORD PTR SS:[EBP+28]
77B8208F   FF75 24          PUSH DWORD PTR SS:[EBP+24]
```

Below is the original entry point of KERNEL32.DLL!CreateProcessA():

```
77B82082 > 8BFF             MOV EDI,EDI
77B82084   55               PUSH EBP
77B82085   8BEC             MOV EBP,ESP
77B82087   6A 00            PUSH 0
77B82089   FF75 2C          PUSH DWORD PTR SS:[EBP+2C]
```

INTERCEPTION_SMART_SIDESTEP. As of Reader 10.1.0, this interception type is still not used. Chrome's source code suggests that this interception type is similar to INTERCEPTION_SIDESTEP.

INTERCEPTION_UNLOAD_MODULE. This is a special interception type for DLLs which should be restricted from being loaded on the sandbox 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 sandbox process. The current list of these DLLs is in "Appendix C: Evicted DLLs".

3.3.4. Inter-Process Communication (IPC)

The IPC infrastructure is used by Reader X to perform IPC calls (or "service calls") from the sandbox process to the broker process. Communication is performed using a shared memory and communication synchronization is done using events.

The IPC client hosted on the sandbox process is used as the interface when code running in the sandbox process needs to perform IPC calls to the broker process. An IPC call can be a forwarded API that needs to be serviced by the broker process, or an IPC call can also be a request by the sandbox process for the broker process to perform a particular action.

The IPC server hosted on the broker process receives the IPC calls and then forwards them to the appropriate dispatcher callbacks which will perform the actual servicing of the request. Dispatchers and dispatcher callbacks are discussed in section 3.3.5.

The IPC shared memory structure and other data structures relating to IPC messaging are discussed in the proceeding subsections. Since the IPC infrastructure of Reader X is based on Chrome's sandbox IPC infrastructure, the structure names and field names used in the listed data structures are taken from Chrome's sources.

3.3.4.1. IPC Shared Memory (IPCControl)

When initializing the sandbox process, the broker creates a chunk of shared memory and uses 2MB (0x200000) of the shared memory for the IPC infrastructure. The broker process duplicates a handle to the shared memory and transfers it to the sandbox process via WriteProcessMemory().

The structure of the IPC shared memory (IPCControl) is as follows:

Offset	Size/Type	Name	Description
0x0000	0x04/size_t	channels_count	Number of IPC channels.
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	15*0x14/ChannelControl	channels	15 ChannelControl structures for 15 IPC channels.
0x0134	15*0x20000/void*	(channel buffers)	15 IPC channel buffers for the 15 IPC channels. The size of each IPC channel buffer is 0x20000 bytes.

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

3.3.4.2. IPC Channels (ChannelControl)

To allow multiple IPC connections, the IPC shared memory is further divided into multiple IPC channels. Each IPC channel has its own corresponding IPC channel buffer (discussed in the next subsection) and synchronization mechanism. As of Reader 10.1.0, 15 IPC channels are available.

The format of an IPC channel (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 (discussed later).

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

3.3.4.3. IPC Channel Buffers (ActualCallParams)

Each IPC channel has a corresponding IPC channel buffer which contains the data passed between the IPC client and the IPC server when an IPC call is made.

When an IPC call is made, the caller (the sandbox process), sets the following information in the IPC channel buffer:

- IPC Tag – a unique identifier that specifies what service the caller is requesting. The IPC server uses this value to select the dispatcher callback routine that will service the request. An example would be the value 0x03 for a request on the broker process to invoke NtCreateFile() on behalf of the sandbox process.
- Parameters – these are the parameters for the IPC call. Example parameters are the object name and access mask used for the IPC call for NtCreateFile(). Parameters are serialized when stored in the IPC channel buffer because they need to be passed between processes.

The IPC server also stores the return values of the IPC call in the IPC channel buffer.

The format of an IPC channel buffer (CrossCallParams, 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.
0x0004	0x04 /uint32	is_in_out_	Contains an in/out parameter.
0x0008	0x34/CrossCallReturn	call_return	Return values filled out by the IPC server.
0x003C	0x04 /size_t	params_count_	Number of parameters.
0x0040	0x0C*(params_count+1)/ParamInfo	param_info_	Parameter information array.
0x????		parameters_	Actual parameter data.

(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 IPC channel buffer.

The format of a 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 Reader), REMOTEBUF_TYPE (8, specific to Reader)
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.

(Reference: http://src.chromium.org/viewvc/chrome/trunk/src/sandbox/src/crosscall_params.h?view=markup, http://src.chromium.org/viewvc/chrome/trunk/src/sandbox/src/internal_types.h?view=markup)

An interesting parameter type in Reader X that is not in Chrome's source code is REMOTEBUF_TYPE. REMOTEBUF_TYPE is a type that represents a buffer in a remote process. REMOTEBUF_TYPE has the following fields:

- Offset 0: pid (4 bytes) – PID 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.

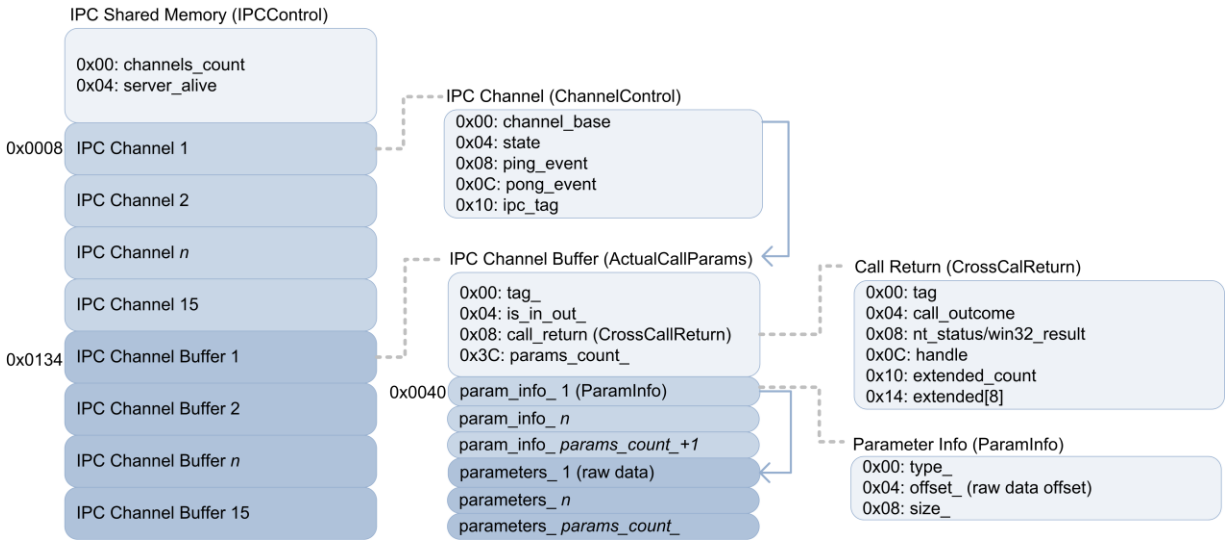
Return values from the IPC call are stored in the ActualCallParams.call_return field which is a CrossCallReturn structure. The format of the CrossCallReturn structure is as follows:

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 Chrome's sandbox_types.h).
0x0008	0x04/NTSTATUS,DWORD	nt_status/win32_result	Return value of an API call when executed on the broker 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.

(Reference: http://src.chromium.org/viewvc/chrome/trunk/src/sandbox/src/crosscall_params.h?view=markup, http://src.chromium.org/viewvc/chrome/trunk/src/sandbox/src/sandbox_types.h?view=markup)

3.3.4.4. IPC Shared Memory Illustration

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



Here are some important points about the illustration:

- There are 15 IPC channels available.
- Each IPC channel has a corresponding IPC channel buffer (pointed to by channel_base).
- The IPC channel buffer contains an IPC tag (tag_) which identifies the requested service the IPC call is for. The value of the IPC tag in the IPC channel buffer is copied to the ipc_tag field of the IPC channel.
- The IPC channel buffer contains a variable number of ParamInfo structures (param_info_) which contain information about the IPC call parameters.
- Each ParamInfo structure points to a serialized parameter data (via the offset_ field) which is also located in the IPC channel buffer.
- There is one extra ParamInfo structure and its offset_ field is used when getting the total size (used part) of the IPC channel buffer.
- The IPC channel buffer will also contain the IPC call return values (call_return) which is filled up by the IPC server.

3.3.4.5. IPC Client: Performing an IPC Call

A typical IPC call performed by the IPC client on the sandbox process involves the following steps:

1. A free IPC channel is selected from the IPC shared memory.
2. The corresponding IPC channel buffer of the selected IPC channel is initialized. This includes the IPC tag being set and the IPC call parameters serialized.
3. The IPC tag in the in the IPC channel is set to value of the IPC tag in the IPC channel buffer.
4. The IPC client notifies the IPC server that an IPC message is available by signaling the ping_event in the IPC channel.
5. The IPC client waits for the IPC server response by waiting for the pong_event in the IPC channel to be signaled.

3.3.4.6. IPC Server: Servicing an IPC Call

The IPC server on the broker process is notified that an IPC message is available in a particular IPC channel due to its ping_event being signaled. Once notified, the IPC server performs the following steps:

1. Verify that the corresponding IPC channel buffer of the IPC channel is properly formed.
2. Create a copy of the IPC channel buffer and perform some additional checks on the IPC channel buffer copy.
3. The IPC call parameters are deserialized from the IPC channel buffer copy.
4. The dispatcher callback routine is selected using the "IPC signature" (the IPC tag plus the types of all the IPC call parameters).
5. The dispatcher callback routine is called with the deserialized parameters passed.
6. The IPC server sets the call_return field in the original IPC channel buffer.
7. The IPC server notifies the IPC client that the IPC call is complete by signaling the pong_event in the IPC channel.

3.3.5. Dispatchers

Dispatchers are entities hosted in the broker process and their purpose is to service IPC calls from the sandbox process. A dispatcher can have multiple callback routines; these callback routines are the actual functions that execute the service requests. And as mentioned in the previous sections, services from the dispatchers are invoked by the sandbox process using the IPC mechanism.

3.3.5.1. Dispatcher Classes

Dispatcher classes group the dispatcher callback routines into functional groups. As of Reader 10.1.0, there are about 19 (1 of which is a dispatcher base class) dispatcher classes. Using information from Chrome's source code plus the C++ Run-time Type Information (RTTI) stored in the Reader binary, the names of the dispatcher classes can be recovered.

The table below lists the currently available dispatcher classes in Reader X.

(As of Reader 10.1.0)

Dispatcher Class Name	Purpose
Dispatcher	Base class of all dispatcher classes.
ExecProcessDispatcher	Handles the spawning of Reader executables. An example is the spawning of AdobeARM.exe when checking for updates.
FilesystemDispatcher	Handles file system services. Handles forwarded file-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 (more on this later). The IPC server uses PolicyBase to resolve the dispatcher callback routine that will service the IPC call.
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. Has interesting callback routines and is a large attack surface due to the number of its callback routines.
SandboxClipboardDispatcher	Handles clipboard services. Mostly handles forwarded clipboard-related USER32.DLL API calls.
SandboxCryptDispatcher	Handles cryptographic services. Mostly handles forwarded crypto-related ADVAPI32.DLL and CRYPT32.DLL API calls.

SandboxKerberosDispatcher	Handles Kerberos credential management services?
SandboxMAPIDispatcher	Handles MAPI services.
SandboxPreviewDispatcher	Handles file preview services?
SandboxPrintDispatcher	Handles printing services.
SandboxSelfhealDispatcher	Handles installation maintenance services. An example is triggering an installation repair action.
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.

Each dispatcher class has a virtual function named SetupService() (2nd virtual function) which performs dispatcher class-specific initialization such as invoking the Interception Manager to intercept specific APIs. The constructor of each dispatcher class is where the dispatcher callback table for each dispatcher class is initialized.

PolicyBase is a special type of a dispatcher class. It is the first dispatcher class instantiated by the broker process. It facilitates the initialization of the other dispatcher classes. The IPC server uses PolicyBase to identify which dispatcher callback routine will service the request. PolicyBase is also used by the other dispatcher classes as an interface to the policy engine; it is also extensively used in the initialization of the sandbox process.

Refer to "Appendix A: Dispatcher Classes" for list of the dispatcher classes with their corresponding virtual function table address and the address of their SetupService() function.

3.3.5.2. Dispatcher Callbacks

As previously discussed, dispatcher callbacks are the actual functions that execute the service requests. If the dispatcher callback has parameters, the IPC server passes the deserialized IPC call parameters (from the IPC channel buffer) to the dispatcher callback routine as arguments.

Note that the IPC server always passes an additional argument to the dispatcher callback routine, this argument is the first argument passed to the dispatcher callback routine and its type is IPCInfo. IPCInfo on the other hand, is a structure that contains information about the sandbox process, it also contains a CrossCallReturn structure (discussed in section 3.3.4.3) which the dispatcher callback routine store its return values. The contents of the CrossCallReturn structure is eventually copied by the IPC server to the IPC channel buffer.

Please refer to the function FilesystemDispatcher::NtCreateFile() in Chrome's source code (http://src.chromium.org/viewvc/chrome/trunk/src/sandbox/src/filesystem_dispatcher.cc?view=markup) for an example of a dispatcher callback routine.

As mentioned in the previous subsection, the dispatcher callback tables are initialized in the constructor of each dispatcher class. Information about each dispatcher callback is stored in an IPCCall structure which contains the following:

- Callback IPC tag
- Parameter information

- Callback routine address

More information about the IPCCall structure can be found in Chrome's crosscall_server.h header file (http://src.chromium.org/viewvc/chrome/trunk/src/sandbox/src/crosscall_server.h?view=markup).

3.3.6. 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.

There are three types of policies available in Reader X:

1. Hard-coded policies
2. Dynamic policies
3. Admin-configurable policies

3.3.6.1. Hard-coded Policies

Hard-coded policies are policies that are applied by default in the sandbox. These 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.

See the tables below for the description of the subsystems and semantics:

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.

The SUBSYS_MUTANT and SUBSYS_SECTION subsystems are unique to Reader X and are not used in Chromium.

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 to create 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_DENY	Deny all 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.

3.3.6.2. Dynamic Policies

Dynamic policies are policies that are applied in response to certain user interactions, such as saving a file. These are generated on the fly and require user confirmation to take effect. For instance, using the File -> Save As to save an opened PDF file as "c:\test.pdf" will result in a call to sandbox::PolicyBase::AddRule with the following parameters:

```
AddRule(SUBSYS_FILES, FILES_ALLOW_ANY, "c:\test.pdf")
```

3.3.6.3. Admin-configurable Policies

Admin-configurable policies are custom policies that can be added by an administrator through a configuration file. It allows the administrator to set whitelists that bypasses the restrictions imposed by the sandbox. The policy file is named ProtectedModeWhitelistConfig.txt and should reside in the Reader install directory (10). 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 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.
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.

3.3.7. Summary: Sandbox Mechanisms

In this section, we discussed in detail the mechanisms employed by the Reader X sandbox. First we described the mechanisms involved in setting up the sandbox such as the restrictions applied and the how the sandbox process is started. Then we went into detail about how the interception manager works and how the sandbox communicates with the broker process through IPC. Lastly, we discussed the different kinds of policies that can be applied to the sandbox and how they are configured. In the next sections, we will use the knowledge we gained in this section to discuss the sandbox's limitations and weaknesses.

4. SANDBOX SECURITY

After discussing the internals of the Reader X sandbox, we will now discuss its security aspects. Specifically, we will be looking at the current limitations and weaknesses of the Reader X sandbox and at the same time discuss their security implications. Also, we will be looking at the possible ways on how code running in the sandbox process can gain additional privileges or achieve code execution in a more privileged context, or in other words, how can a sandboxed code escape the Reader X sandbox.

4.1. Sandbox Limitations and Weaknesses

This section lists the current limitations and weaknesses of the Reader X sandbox. It answers the important question "what can malicious code do once it is running inside the Reader X sandbox?" Most of the items described in this section are already known to and noted by Adobe (11), and we can speculate that in most cases, the existence of these limitations/weaknesses is because of compatibility reasons or addressing them would require significant changes in Adobe Reader.

4.1.1. File System Read Access

One weakness of the Reader X sandbox is that it 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" in the Reader install directory 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.

4.1.2. Registry Read Access

Another weakness of the Reader X sandbox is that it 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.

4.1.3. Clipboard Read/Write Access

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 the job object which the sandbox process is assigned to, the SandboxClipboardDispatcher dispatcher class in the broker process also provides clipboard-related services which allows clipboard access in the context of the broker process. Thus, 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” (12; 13).

4.1.4. Network Access

A notable limitation of the Reader X sandbox is its inability to restrict network access. This limitation would allow malicious code running in the sandbox process to send stolen information to a remote server. And from an attacker’s perspective, 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. Moving forward, Adobe is investigating ways to restrict network access in the future (11).

4.1.5. Policy-Allowed Write Access to Certain Folders/Files

A weakness we found on our research 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.

One example of such permissive rule is the write access rule to "%APPDATA%\Adobe\Acrobat\10.0*". By leveraging the said rule, malicious code running in a sandbox process can create or modify certain JavaScript files located under the folder "%APPDATA%\Adobe\Acrobat\10.0\JavaScripts" to cause Reader X to execute an attacker-controlled JavaScript code on certain events. One example of such file is the "config.js" JavaScript file which is automatically executed every time a new instance of Reader X is spawned.

In addition, the ability to create a local file with a controllable file name can 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.

4.1.6. 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 copy of the PDF file containing an exploit code that initially compromised the sandbox process or dropping a combination an EXE file and an autorun.inf file.

4.1.7. Summary: Sandbox Limitations and Weaknesses

While code running in the Reader X sandbox is severely limited in terms of what it can do, it is still possible to carry out information theft attacks by leveraging the its 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.

Lastly, Adobe noted (14; 11) that they are aware and acknowledges that information leakage is possible and they plan to extend the sandbox to include restriction of read activities in future releases of Reader.

4.2. Sandbox Escape

After discussing the limitations and weaknesses of the Reader X sandbox, we will now take a look at ways how code running in the context of the sandbox process could gain additional privileges which would otherwise be limited by the sandbox. This section attempts to answers the question "how might malicious code escape the Reader X sandbox?"

As 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 (7). This means, that for example, code running in the sandbox process can use the NTDLL.DLL and KERNEL32.DLL base in the sandbox process when crafting a ROP sequence for the exploit to be used against the broker process.

4.2.1. 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 the sandbox 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” (15) by Tavis Ormandy and Julien Tinnes.

4.2.2. Named Object Squatting Attacks

Named object squatting is an attack that involves a malicious application creating a named object which 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. In the Hack In Paris 2011 presentation "Practical Sandboxing on the Windows Platform" (16), Tom Keetch was able to confirm that the Reader X sandbox is vulnerable to named object squatting.

4.2.3. Leveraging Write-Allowed Policy Rules

There are several resources (parts of the registry, parts of the file system, sections, etc.) the sandbox process is allowed write access to. If the broker process or another process running with a higher privilege trusts and uses the data stored in these writable resources, code running in the sandbox process might be able to control the behavior of these higher-privileged processes which in turn could lead to a sandbox escape.

Some example scenarios of leveraging the write-allowed policy rules for a sandbox escape are:

- If the sandboxed code can write data to a particular file or registry key that a higher-privileged application uses, it may be possible to exploit the higher-privileged application by crafting a malicious data designed to exploit a parsing vulnerability in the higher-privileged application and then storing it to the writable file or registry key.
- If the sandboxed code can write data to a particular file or registry key that is used by a higher-privileged application for storing configuration data, and if for example, the configuration data contains information such as executable file paths, library file paths, etc., execution of an attacker controlled code at a higher privilege level may be possible if the higher-privileged application fully trusts the contents of the configuration data.

There are numerous scenarios where the write-allowed policy rules can be abused. And as long as sandboxed code can control the behavior of higher-privileged applications, there is always a possibility of a sandbox escape.

4.2.4. Broker Process Attack Surface

Another potential sandbox escape is by attacking the broker process. Running with more privileges and a higher integrity level than the sandbox process, sandboxed code can leverage a vulnerability in the broker process in order to execute code at more privileged context. This section enumerates the attack surface in the broker process.

4.2.4.1. IPC Server

The first code in the broker process that touches untrusted data from the IPC shared memory is the IPC server. Specifically, this is the code that verifies and deserializes the IPC call parameters from the IPC channel buffer.

An attack on the IPC server will involve malicious code running in the sandbox process creating a malformed IPC channel buffer content and then emulating an IPC call in order for the IPC server to process the contents of the IPC channel buffer. From an auditing perspective, the following are the interesting IPC server routines (using the function names from Chrome's source code):

- SharedMemIPCServer::InvokeCallback() - code invoked in the IPC server when an IPC call is received and when the dispatcher callback routine is about to be called. It invokes the two functions described below.
- CrossCallParamsEx::CreateFromBuffer() – code used to verify if the contents of the IPC channel buffer is properly formed and it creates a copy of the IPC channel buffer for further processing.
- GetArgs() – code that deserializes the IPC call parameters from the IPC channel buffer to their actual types. Deserialization is done using a copy of the IPC channel buffer.

Refer to "Appendix B: Interesting Entry points" for the address of the above functions in Reader 10.1.0.

Since Adobe used Chrome's IPC server code for Reader X, we can assume that the source code for the above functions had already been extensively audited by several parties. However, we can expect that Adobe will add new functionalities in the Reader X IPC server - such as adding new IPC call parameter types or possibly extending the IPC messaging data structures to accommodate new features. The code that will handle those new functionalities will be interesting because it is new code which only a few had taken a look at.

4.2.4.2. Dispatcher Callbacks

The large attack surface in the broker process is due to the number dispatcher callbacks that perform potentially security-sensitive operations which inputs are directly coming from a potentially malicious code running in the sandbox process. Using knowledge of the IPC messaging data structures plus the number and types of parameters each dispatcher callback routine accepts, an attacker can craft a properly formed malicious IPC message which can pass through the checks performed by the IPC server and can reach the target dispatcher callback routine. Refer to section 3.3.5.2 for information about dispatcher callbacks.

We can expect that Adobe will continuously add new dispatcher callback routines to accommodate new features/services that need to be exposed by the broker process to the sandbox process.

4.2.4.3. Policy Engine

Another attack surface in the broker process is the policy engine. Acting as a "gatekeeper" in the broker process, it decides which potentially security-sensitive actions are allowed to run in the context of the broker process. Malicious code running in the sandbox process can leverage a vulnerability in the policy engine in order evade policy checks and thereby write to sensitive parts of the file system or registry, which in turn could lead to elevation of privileges if the said part of the registry or file system is being used by the operating system, the

broker process or an application that runs with a higher privilege. Finding vulnerabilities in the policy engine involves understanding how the policy engine performs policy evaluation using the policy rules and then looking for ways to influence the policy evaluation results.

4.2.5. Summary: Sandbox Escape

In this section, we had listed possible ways to escape the Reader X sandbox. From attacking other applications to attacking the broker process, sandboxed code has multiple options to gain code execution in a more privileged context. Of course, the list is not complete as there may be new sandbox escape techniques that will be discovered.

With more and more critical applications being sandboxed, an attacker would now need to use a second vulnerability to install persistent malware in a target system. That being said, local elevation of privilege (EoP) vulnerabilities will become more important. This is interesting because they have received little attention on the endpoint of late. Lastly, the attack surface of the broker component of sandboxes will be continually scrutinized for security vulnerabilities.

5. CONCLUSION

Being able to integrate a sandbox into a large application such as Adobe Reader is an engineering feat because it is a complex balancing act between adding security while not breaking existing functionality. We feel that Adobe made the right decision on basing their sandbox design on well-known techniques such as the “Practical Windows Sandboxing” technique and not re-inventing the wheel by basing their sandbox code on the open source Google Chrome sandbox code and thereby taking the lessons learned when the Chrome sandbox was built with it.

We hope that you, the reader, now have a better understanding of the Reader X sandbox and that we had shed light on how its internal mechanisms work, and we were able to convey its currently limitations and weaknesses. But more importantly, and being one of the major goal of our paper, is that we hope that we had created an awareness that the impact of specialized malicious code running in the Reader X sandbox is still substantial, and thus, we cannot afford to be complacent - we still (and always) need to be cautious about the files we open even if the application we are using have a sandboxing capability.

6. BIBLIOGRAPHY

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

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

3. —. Practical Windows Sandboxing – Part 3. [Online] http://blogs.msdn.com/b/david_leblanc/archive/2007/07/31/practical-windows-sandboxing-part-3.aspx.

4. **Randolph, Kyle, et al.** Inside Adobe Reader Protected Mode – Part 2 – The Sandbox Process. *Adobe Secure Software Engineering Team (ASSET) Blog.* [Online] http://blogs.adobe.com/asset/2010/10/inside-adobe-reader-protected-mode-%E2%80%93-part-2-%E2%80%93-the-sandbox-process.html.

5. **Microsoft.** Restricted Tokens. *MSDN.* [Online] http://msdn.microsoft.com/en-us/library/aa379316%28v=vs.85%29.aspx.

6. —. What is the Windows Integrity Mechanism? *MSDN.* [Online] http://msdn.microsoft.com/en-us/library/bb625957.aspx.

7. **Russinovich, Mark, Solomon, David and Ionescu, Alex.** *Windows® Internals: Including Windows Server 2008 and Windows Vista, Fifth Edition.* s.l. : Microsoft Press, 2009. 0735625301.

8. **Microsoft.** Job Objects. *MSDN.* [Online] http://msdn.microsoft.com/en-us/library/ms684161%28VS.85%29.aspx.

9. —. User Objects. *MSDN.* [Online] http://msdn.microsoft.com/en-us/library/ms725486%28v=vs.85%29.aspx.

10. **Randolph, Kyle, et al.** Inside Adobe Reader Protected Mode – Part 3 – Broker Process, Policies, and Inter-Process Communication. *Adobe Secure Software Engineering Team (ASSET) Blog.* [Online] http://blogs.adobe.com/asset/2010/11/inside-adobe-reader-protected-mode-part-3-broker-process-policies-and-inter-process-communication.html.

11. —. Inside Adobe Reader Protected Mode – Part 1 – Design. *Adobe Secure Software Engineering Team (ASSET) Blog.* [Online] http://blogs.adobe.com/asset/2010/10/inside-adobe-reader-protected-mode-part-1-design.html.

12. **Keetch, Tom.** Practical Sandboxing on the Windows Platform (White Paper). *Black Hat Europe 2011.* [Online] https://media.blackhat.com/bh-eu-11/Tom_Keetch/BlackHat_EU_2011_Keetch_Sandboxes-WP.pdf.

13. —. Practical Sandboxing on the Windows Platform (Slides). *Black Hat Europe 2011.* [Online] https://media.blackhat.com/bh-eu-11/Tom_Keetch/BlackHat_EU_2011_Keetch_Sandboxes-Slides.pdf.

14. **Arkin, Brad.** Introducing Adobe Reader Protected Mode. *Adobe Secure Software Engineering Team (ASSET) Blog.* [Online] http://blogs.adobe.com/asset/2010/07/introducing-adobe-reader-protected-mode.html.

15. **Ormandy, Tavis and Tinnes, Julien.** There's a party at Ring0, and you're invited. *Black Hat USA 2010.* [Online] http://www.cr0.org/paper/to-jt-party-at-ring0.pdf.

16. **Keetch, Tom.** Practical Sandboxing on the Windows Platform. *Hack In Paris 2011.* [Online] http://www.slideshare.net/tkeetch/hack-in-paris-2011-assessing-practical-sandboxes.

17. **Johnson, Richard.** A Castle Made of Sand: Adobe Reader X Sandbox. *CanSecWest Vancouver 2011.* [Online] http://rjohnson.uninformed.org/Presentations/A%20Castle%20Made%20of%20Sand%20-%20final.pdf.

18. **Ridley, Stephen.** Escaping the Sandbox. *Black Hat Abu Dhabi 2010.* [Online] https://media.blackhat.com/bh-ad-10/Ridley/BlackHat-AD-2010-Ridley-Escaping-The-Sandbox-slides.pdf.

7. APPENDIX A: DISPATCHER CLASSES

(As of Reader 10.1.0)

Dispatcher Class Name	Virtual Function Table Address	SetupService() Address
Dispatcher	0x004E6328	0x0048E7D7
ExecProcessDispatcher	0x004EB098	0x0044C360
FilesystemDispatcher	0x004D0BEC	0x0042C850
MutantDispatcher	0x004EC438	0x0046DD40
NamedPipeDispatcher	0x004D0918	0x0042C660
PolicyBase	0x004CFA50	0x004214B0
RegistryDispatcher	0x004D023C	0x00428320
SandboxBrokerServerDispatcher	0x004E6D98	0x0044C360
SandboxClipboardDispatcher	0x004EBB2C	0x0044C360
SandboxCryptDispatcher	0x004EBAF8	0x0044C360
SandboxKerberosDispatcher	0x004EB1B4	0x0044C360
SandboxMAPIDispatcher	0x004EC040	0x0044C360
SandboxPreviewDispatcher	0x004E92E0	0x0044C360
SandboxPrintDispatcher	0x004E9E50	0x0044C360
SandboxSelfhealDispatcher	0x004E9E90	0x0044C360
SandboxWininetDispatcher	0x004EA128	0x0044C360
SectionDispatcher	0x004EC2E8	0x0046D390
SyncDispatcher	0x004D0420	0x00428F90
ThreadProcessDispatcher	0x004D0844	0x0042AD00
Total: 19		

8. APPENDIX B: INTERESTING ENTRY POINTS

(As of Reader 10.1.0)

Function Name	Address	Purpose
SharedMemIPCServer::InvokeCallback()	0x004232A0	Code invoked in the IPC server when an IPC call is received and when the dispatcher callback routine is about to be called.
CrossCallParamsEx::CreateFromBuffer()	0x00427C80	Code used to verify if the contents of the IPC channel buffer is properly formed and it creates a copy of the IPC channel buffer for further processing.
GetArgs()	0x00422EA0	Code that deserializes the IPC call parameters from the IPC channel buffer to their actual types. Deserialization is done using a copy of the IPC channel buffer.

9. APPENDIX C: EVICTED DLLS

(As of Reader 10.1.0)

DLL Name
adialhk.dll
acpiz.dll
avgrsstx.dll

btkeyind.dll
cmcsyshk.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
pctavhook.dll
pctgmhk.dll
prntrack.dll
radhslib.dll
radprlib.dll
rlhook.dll
r3hook.dll
sahook.dll
sbrige.dll
sc2hook.dll
sguard.dll
smum32.dll
smumhook.dll
ssldivx.dll

syncor11.dll
systools.dll
tfwah.dll
wblind.dll
wbhelp.dll
winstylerthemehelper.dll