DIVING INTO IE 10'S ENHANCED PROTECTED MODE SANDBOX

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

(v3)

ABSTRACT

With the release of Internet Explorer 10 in Windows 8, an improved version of IE's Protected Mode sandbox, called Enhanced Protected Mode (EPM), was introduced. With the use of the new AppContainer process isolation mechanism introduced in Windows 8, EPM aims to further limit the impact of a successful IE compromise by limiting both read and write access and limiting the capabilities of the sandboxed IE process.

As with other new security features integrated in widely-deployed software, it is just prudent to look at how EPM works internally and also evaluate its effectiveness. This presentation aims to provide both by delving deep into the internals and assessing the security of IE 10's Enhanced Protected Mode sandbox.

The first part of this presentation will focus on the inner workings of the EPM sandbox where topics such as the sandbox restrictions in place, the inter-process communication mechanism in use, the services exposed by the higher-privileged broker process, and more are discussed. The second part of this presentation will cover the security aspect of the EPM sandbox where its limitations are assessed and potential avenues for sandbox escape are discussed.

Finally, in the end of the presentation, an EPM sandbox escape exploit will be demonstrated. The details of the underlying vulnerability, including the thought process that went through in discovering it will also be discussed.

IBM Security Systems | © 2014 IBM Corporation

CONTENTS

Contents 2

1. Introduction 3

2. Sandbox Internals 4

2.1. Architecture 4
2.2. Sandbox Restrictions 5
2.2.1. Securable Object Restrictions 6
2.2.2. Object Namespace Isolation 7
2.2.3. Global Atom Table Restrictions 7
2.2.4. User Interface Privilege Isolation (UIPI) Enhancements 8
2.2.5. Network Isolation 8
2.2.6. Analysis: Unapplied Restriction/Isolation Mechanisms 9
2.3. IE Shims (Compatibility Layer) 9
2.4. Elevation Policies 10
2.5. Inter-Process Communication 10
2.5.1. Shared Memory IPC 11
2.5.2. COM IPC 14
2.6. Services 14
2.6.1. User Broker Object Services 14
2.6.2. Known Broker Objects Services 16
2.6.3. Broker Components Message Handlers 17
2.7. Summary: Sandbox Internals 18

3. Sandbox Security 19

3.1. Sandbox Limitations/Weaknesses 19
3.1.1. File System Access 19
3.1.2. Registry Access 20
3.1.3. Clipboard Access 20
3.1.4. Screen Scraping And Screen Capture 21
3.1.5. Network Access 21
3.2. Sandbox Escape 21
3.2.1. Local Elevation of Privilege (EOP) Vulnerabilities 21
3.2.2. Policy/Permission Vulnerabilities 21
3.2.3. Policy Check Vulnerabilities 22
3.2.4. Service Vulnerabilities 23
3.3. Summary: Sandbox Security 23

4. Conclusion 24

5. Bibliography 25

1. INTRODUCTION

One of the goals of Protected Mode since its introduction in IE7 is to prevent an attack from modifying data and to prevent the installation of persistent malicious code in the system [1]. However, because Protected Mode does not restrict read access to most resources and it does not restrict network access [2, 3], attacks can still read and exfiltrate sensitive information or files from the system. With the release of IE10 in Windows 8, an improved version of Protected Mode called Enhanced Protected Mode (EPM) was introduced with one of the objectives being to protect personal information and corporate assets [4] by further limiting the capabilities of the sandboxed IE.

New security features in widely-deployed software such as the EPM in IE always deserve a second look from unbiased observers because important questions such as "how does it work?", "what can malicious code still do or access once it is running inside the EPM sandbox?" and "what are the ways to bypass it?" needs to be answered candidly. Answering these important questions is the goal of this paper.

The first part of this paper (Sandbox Internals) takes a look at how the EPM sandbox works by discussing what sandbox restrictions are in place and the different mechanisms that make up the EPM sandbox. The second part of this paper (Sandbox Security) is where the EPM sandbox limitations/weaknesses and potential avenues for EPM sandbox escape are discussed. To sum up the main points discussed, a summary section is provided in the end of each part.

This paper is based on IE10 update KB2817183 (April 2013) running in Enhanced Protected Mode on Windows 8 (x64). With the exception of a few noted differences, the information in this paper still mostly applies to IE10 and IE11 patch KB2909921 (February 2014), the latest IE patch when this paper was updated.

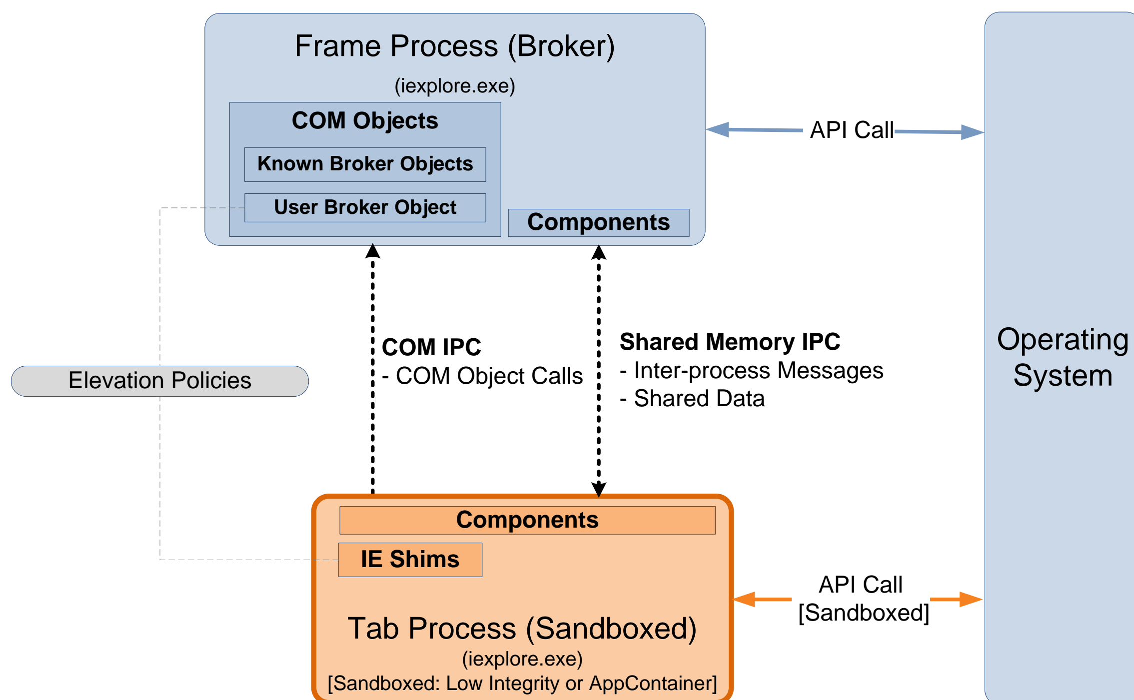
2. SANDBOX INTERNALS

Understanding how a piece of software works is the first step in figuring out its limitations and weaknesses. In this first part of the paper we'll take a look at the internals of the EPM sandbox. We'll start by looking at the high-level architecture which shows how the different sandbox mechanisms work together and then discuss in detail how each mechanism works.

2.1. ARCHITECTURE

IE10 follows the Loosely-Coupled IE (LCIE) process model introduced in IE8 [5]. In the LCIE process model, the *frame process* hosting the browser window (also called the UI frame) is separate from the *tab processes* which host the browser tabs. The frame process is responsible for the lifetime of the tab processes. The tab processes, on the other hand, are responsible for parsing and rendering content and also host the browser extensions such as Browser Helper Objects (BHO) and toolbars.

To limit the impact of a successful IE compromise, the tab process which processes potentially untrusted data is sandboxed. IE10 supports two options for sandboxing the tab process – Protected Mode which runs the tab process at Low Integrity, and Enhanced Protected Mode which runs the tab process inside a more restrictive *AppContainer* (see section 2.2):



To facilitate sharing of data and inter-process message exchange between components in the tab process and the frame process, a shared memory IPC mechanism (2.5.1) is used. Additionally, a separate COM IPC (2.5.2) mechanism is used by the sandboxed process for invoking services exposed by COM objects in the frame process.

A shim mechanism (2.3) in the tab process provides a compatibility layer for browser extensions to run on a low-privilege environment, it is also used for forwarding API calls to the broker in order to support certain functionalities, furthermore, it is also used to apply elevation policies (2.4) if an operation will result in the launching of a process or a COM server.

The shim mechanism uses the services exposed by the User Broker COM Object (2.6.1) to launch an elevated process/COM server if dictated by the elevation policies. Finally, several Known Broker COM Objects (2.6.2) in the frame process also provide additional services that are used by the shim mechanism and other components in the tab process.

For uniformity, the rest of this paper will use the term "*broker process*" for the higher-privileged frame process and the term "*sandboxed process*" for the lower-privileged tab process.

2.2. SANDBOX RESTRICTIONS

The sandboxed process is mainly restricted via the *AppContainer* [6, 7, 8, 9] process isolation feature first introduced in Windows 8. In this new process isolation mechanism, a process runs in an AppContainer which limits its read/write access, and also limits what the process can do.

In a very high level, an AppContainer has the following properties:

- *Name*: A unique name of the AppContainer
- *SID* (security identifier): AppContainer SID – SID generated from the AppContainer name using *userenv!DeriveAppContainerSidFromAppContainerName()*
- *Capabilities*: Security capabilities [10] of the AppContainer - it is a list of what processes running in the AppContainer can do or access

In the case of IE's AppContainer, the AppContainer name is generated using the format "*windows_ie_ac_%03i*". And by default, the capabilities assigned to IE's AppContainer are as follows (see Figure 1):

- *internetExplorer* (S-1-15-3-4096)
- *internetClient* (S-1-15-3-1)
- *sharedUserCertificates* (S-1-15-3-9)
- *location* (S-1-15-3-3215430884-1339816292-89257616-1145831019)
- *microphone* (S-1-15-3-787448254-1207972858-3558633622-1059886964)
- *webcam* (S-1-15-3-3845273463-1331427702-1186551195-1148109977)

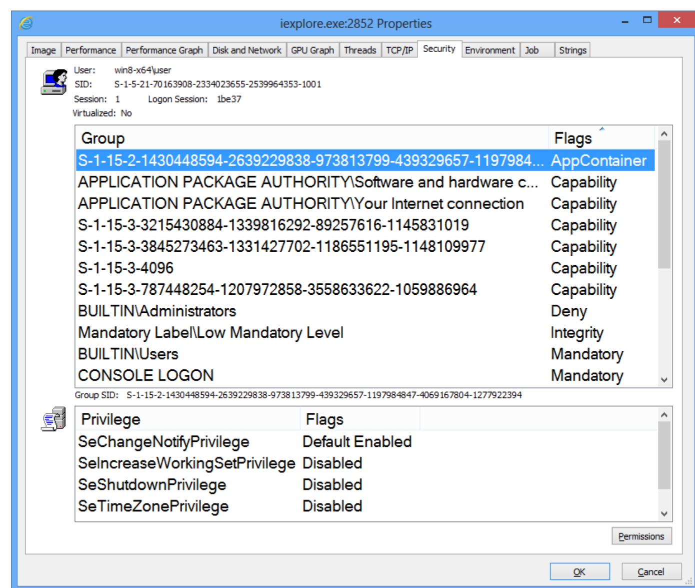


Figure 1 – IE EPM AppContainer and Capabilities

If private network access (2.2.5) is turned on, IE uses a separate AppContainer with the following additional capabilities assigned in order to allow access to private network resources:

- *privateNetworkClientServer* (S-1-15-3-3)
- *enterpriseAuthentication* (S-1-15-3-8)

The broker process calls *iertutil!MICSecurityAware_CreateProcess(),* an internal IE function responsible for spawning the sandboxed process in IE's AppContainer. The actual spawning of the sandboxed process is done via a

call to *kernel32!CreateProcesW()* in which the passed *lpStartupInfo.lpAttributeList* has a *SECURITY_CAPABILITIES* [11] attribute to signify that the process will be running in an AppContainer.

At a low level, processes running in an AppContainer are assigned a *Lowbox* token which, among other things, has the following information set:

- *Token flags – TOKEN_LOWBOX* (0x4000) is set
- *Integrity* – Low Integrity
- *Package* – AppContainer SID
- *Capabilities* –Array of Capability SIDs
- *Lowbox Number Entry* – A structure which links the token with an *AppContainer number* (also called *Lowbox Number or Lowbox ID)*, a unique per-session value that identifies an AppContainer and is used by the system in various AppContainer isolation/restriction schemes.

With the additional information stored in the token of an AppContainer process, additional restrictions and isolation schemes can be enforced by the system. The next subsections will discuss some of these restrictions and isolation schemes in details.

2.2.1. SECURABLE OBJECT RESTRICTIONS

One of the important restrictions afforded by AppContainer is that for an AppContainer process to access securable objects (such as files and registry keys), the securable object would need to have an additional access control entry for any of the following:

- The AppContainer
- "*ALL APPLICATION PACKAGES*" (S-1-15-2-1)
- A capability that matches one of the AppContainer's capabilities

What this means, for example, is that personal user files such as those stored in the user's Documents, Pictures and Videos folder (i.e. *C:\Users\<UserName>\Documents,Pictures,Videos*) will not be accessible to the sandboxed process because they do not have an access control entry for any of the above.

There are AppContainer-specific locations which are available for an AppContainer process for data storage. These locations are as follows:

- File System:
  - *%UserProfile%\AppData\Local\Packages\<AppContainer Name>\AC*
- Registry:
  - *HKCU\Software\Classes\Local Settings\Software\Microsoft\Windows\CurrentVersion\AppContainer\Storage\<AppContainer Name>*

AppContainer processes are able to access the above locations because they have an access control entry for the AppContainer

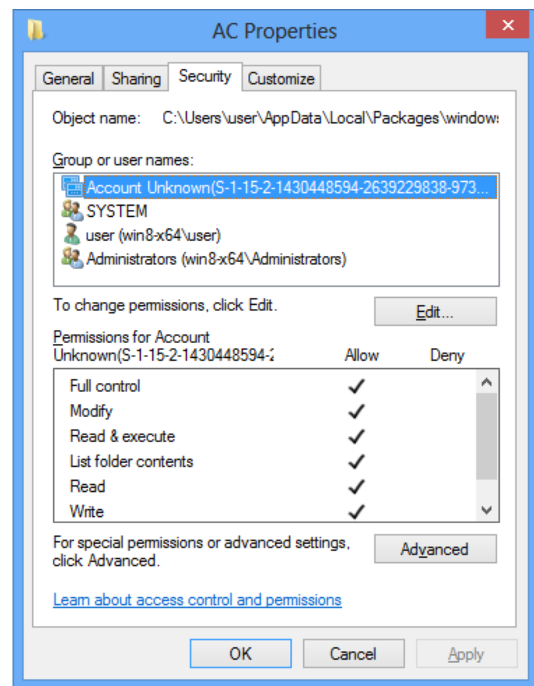


Figure 2 - AppContainer ACE in an AppContainer-specific location

(see Figure 2).

The sandboxed IE process can also access browser-related data located outside the previously described AppContainer locations because they are stored in locations that have an access control entry for the *internetExplorer* capability (see Figure 3).

Examples of these locations are:

- File System:
    - *%UserProfile%\AppData\Local\Microsoft\Feeds* (Read access)
    - *%UserProfile%\Favorites* (Read/Write access)
- Registry:
    - *HKCU\Software\Microsoft\Feeds* (Read access)
    - A few subkeys of *HKCU\Software\Microsoft\Internet Explorer* (Read or Read/Write access)

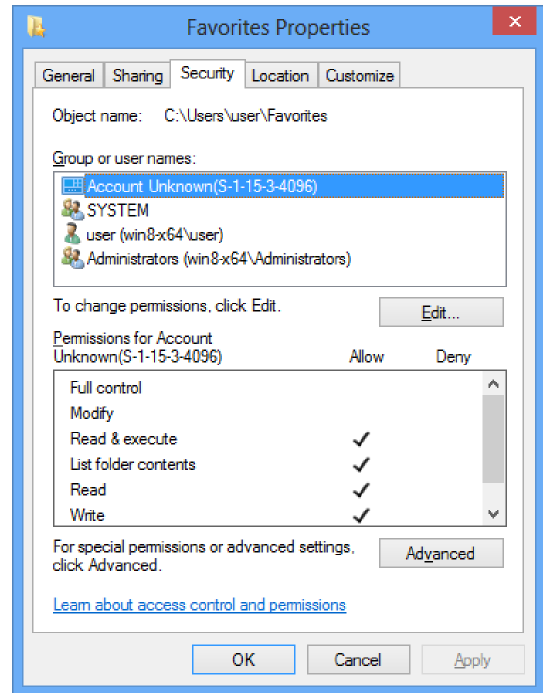


Figure 3 - *internetExplorer* capability ACE in a browser-related location

2.2.2. OBJECT NAMESPACE ISOLATION

An isolation scheme provided by AppContainer is that processes running in an AppContainer will have their own object namespace. With this isolation feature, named objects that will be created by the AppContainer process will be inserted in the following AppContainer-specific object directory (see Figure 4):

- *\Sessions\<Session>\AppContainerNamedObjects\<AppContainer SID>*

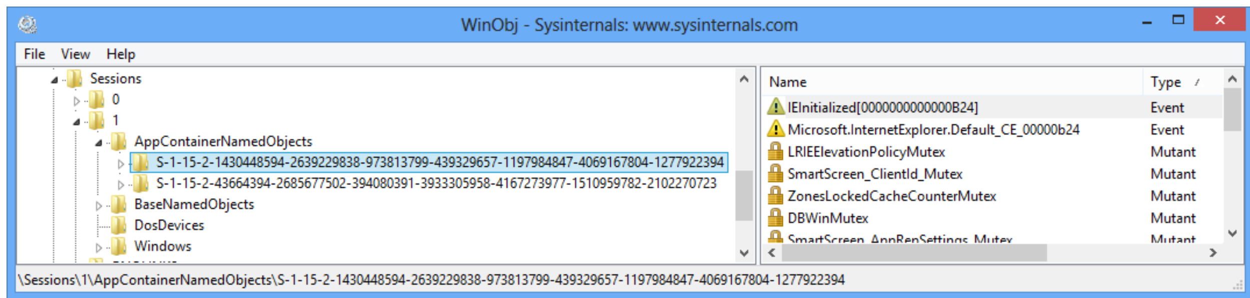


Figure 4 - AppContainer-specific object directory

Object namespace isolation prevents *named object squatting* [12], a privilege escalation technique that relies on multiple processes with different privileges sharing the same object namespace.

2.2.3. GLOBAL ATOM TABLE RESTRICTIONS

AppContainer processes are also restricted from querying and deleting global atoms. Querying and deletion is limited to global atoms that are created by processes running in the same AppContainer or existing global atoms that are referenced (which can occur via a call to *kernel32!GlobalAddAtom()*) by processes running in the same

AppContainer. The query restriction is lifted if the *ATOM_FLAG_GLOBAL* flag is set in the atom (more details on this is in the reference mentioned below).

The query restriction prevents leaking of potentially sensitive information stored in global atoms while the delete restriction prevents an AppContainer process from freeing atoms that are referenced by more privileged processes which can be an opportunity for a sandbox escape.

Internally, in Windows 8, atoms in the global atom table are represented by an updated *nt!_RTL_ATOM_TABLE_ENTRY* structure which additionally keeps track of which AppContainers are referencing the atom. The kernel uses the previously mentioned AppContainer number (termed as Lowbox ID in the related atom table entry structures) to identify which AppContainers are referencing a particular atom.

More information regarding the atom table changes in Windows 8, including privilege escalation attacks relating to atoms can be found in Tarjei Mandt's presentation *"Smashing the Atom: Extraordinary String Based Attacks"* [13].

2.2.4. User Interface Privilege Isolation (UIPI) Enhancements

Initially introduced in Windows Vista, User Interface Privilege Isolation (UIPI) [14, 15] prevents a lower integrity process from sending write-type windows messages or installing hooks in a higher-integrity process. It mitigates *shatter* attacks [16], which is another privilege escalation technique that relies on the ability of processes with different privileges to exchange window messages.

In Windows 8, an additional check was added to restrict the sending of write-type messages across AppContainers. The additional check involves comparing the AppContainer number of the processes if their integrity level is equal. This, for example, prevents the sandboxed IE process from sending write-type messages to another Windows Store App or to a process which also runs at low integrity but is not running in an AppContainer (AppContainer number 0 is given to non-AppContainer processes).

2.2.5. Network Isolation

Another restriction provided by AppContainer is network isolation [17]. With network isolation, the AppContainer needs to have specific capabilities in order for the AppContainer process to connect to Internet and public network endpoints (*internetClient)*, connect to and receive connections from Internet and public network endpoints (*internetClientServer*), and connect to and receive connections from private (trusted intranet) network endpoints (*privateNetworkClientServer)*. There are certain checks [17] that are used by the system to determine whether an endpoint is classified as part of the Internet/public network or the private network.

By default, the sandboxed IE process only has the *internetClient* capability which means that it can only connect to Internet and public network endpoints. Connections to private network endpoints such as those that are part of trusted home and corporate intranets are blocked. Additionally, receiving connections from Internet, public network and private network endpoints are also blocked.

There are instances where a user needs access to a site hosted on a private network endpoint but the site is categorized in a security zone in which EPM will be enabled. For these cases, in IE10, the user is given an option to enable "private network access", which if enabled, will result in the sandboxed process to run in a separate IE AppContainer that includes the *privateNetworkClientServer* and *enterpriseAuthentication* capability. In IE11, private network access is automatically enabled when visiting private network sites.

2.2.6. ANALYSIS: UNAPPLIED RESTRICTION/ISOLATION MECHANISMS

Compared to other sandbox implementations, such as the Google Chrome sandbox [18], there are still well-known sandbox restrictions or isolation mechanisms that IE EPM does not apply.

The first unapplied restriction is job object [19] restrictions. Using job object restrictions, a sandboxed process can be restricted from performing several types of operations, such as preventing access to the clipboard, spawning additional processes, and preventing the use of USER handles (handles to user interface objects) owned by process not associated with same job, etc. Though the sandboxed process is associated with a job, the job object associated to it doesn't have any strict restrictions in place.

Also, EPM does not implement window station and desktop isolation [20], therefore, the sandboxed process can still access the same clipboard used by other processes associated to the same windows station, and can still send messages to windows owned by other processes on the same desktop.

Finally, EPM does not use a restricted token [21] which can further restrict its access to securable resources. Interestingly, Windows 8 allows the use of restricted tokens in AppContainer processes, giving developers of sandboxed applications the ability to combine the two restriction/isolation schemes.

Without these restrictions or isolation mechanisms applied, some attacks, mostly resulting to disclosure of some types of potentially sensitive or personal information can still be performed by the sandboxed process. Details regarding these attacks are discussed in the Sandbox Limitations/Weaknesses section (3.1).

2.3. IE SHIMS (COMPATIBILITY LAYER)

For compatibility with existing binary extensions [22], IE includes a shim mechanism that redirects certain API calls so that they will work on a low-privilege environment. An example of this is altering a file's path to point to a writable location such as *"%UserProfile%\AppData\Local\Microsoft\Windows\Temporary Internet* Files*Virtualized\\<original path>"* before calling the actual API.

There are also certain instances in which an API needs to be executed in the context of the broker in order to support a particular functionality. An example is forwarding *kernel32!CreateFileW()* calls to the *CShdocvwBroker* Known Broker Object (2.6.2) in order to allow the loading of local files that passes the Mark-of-the-Web (MOTW) check [23] in EPM.

Additionally, the shim mechanism allows IE to apply elevation policies (2.4) to API calls that would potentially result in the launching of a process or a COM server. These APIs are as follows:

- *kernel32.dll!CreateProcessA*
- *kernel32.dll!CreateProcessW*
- *kernel32.dll!WinExec*
- *ole32.dll!CoCreateInstance*
- *ole32.dll!CoCreateInstanceEx*
- *ole32.dll*!*CoGetClassObject*

When the above APIs are called in the sandboxed process, IE Shims first checks if the API call needs to be forwarded to the broker process by checking the elevation policies. If the API call needs to be forwarded to the broker process, via the COM IPC (2.5.2), the call will be forwarded to the User Broker Object (2.6.1) in the broker process. The User Broker Object will in turn re-check the API call against the elevation policies and then execute the API call if allowed by the elevation policies.

The shim mechanism is provided by the *ieshims.dll* module which sets up a callback that is called every time a DLL is loaded via the *ntdll!LdrRegisterDllNotification()* API. When the DLL load callback is executed, the DLL's entry point (via *LDR_DATA_TABLE_ENTRY.EntryPoint*) is updated to execute *ieshims!CShimBindings::s_DllMainHook()* which in turn performs the API hooking and then transfers controls to the DLL's original entry point. API hooking is done via import address table patching, for dynamically resolved API addresses, *kernel32!GetProcAddress()* is hooked to return a shim address.

2.4. ELEVATION POLICIES

Elevation policies determine how a process or a COM server will be launched and at what privilege level. As discussed in the previous section, the IE Shims mechanism uses the elevation policies to determine if certain launch-type API calls will be executed in the sandboxed process or will be forwarded to the User Broker Object in the broker process.

Elevation policies are stored in registry keys with the following format:

- *HKLM\Software\Microsoft\Internet Explorer\Low Rights\ElevationPolicy\<GUID>*

With each registry key having the following values:

- *AppName* – For executables: Application executable name
- *AppPath* – For executables: Application path
- *CLSID* – For COM servers: COM class CLSID
- *Policy* – Policy value. Based on Microsoft's documentation [1], can be any of the following values:

Policy	Description
0	Protected Mode prevents the process from launching.
1	Protected Mode silently launches the broker as a low integrity process.
2	Protected Mode prompts the user for permission to launch the process. If permission is granted, the process is launched as a medium integrity process.
3	Protected Mode silently launches the broker as a medium integrity process.

Note that if EPM is enabled and if the policy is 1 (low integrity), the process will actually be launched in the sandboxed process' AppContainer. If there is no policy for a particular executable or a COM class, the default policy used is dictated by the following registry value:

- *HKLM\Software\Microsoft\Internet Explorer\Low Rights::DefaultElevationPolicy*

Internally, in IE Shims, the elevation policy checks are done via *ieshims!CProcessElevationPolicy::GetElevationPolicy()*, and in the User Broker Object, the elevation policy checks are done via *ieframe!CProcessElevationPolicy::GetElevationPolicy()* and *ieframe!CProcessElevationPolicy::GetElevationPolicyEx()*.

2.5. INTER-PROCESS COMMUNICATION

Inter-Process Communication (IPC) is the mechanism used by the broker and sandboxed process for sharing data, exchanging messages and invoking services. There are two types of IPC mechanism used by IE - shared memory IPC and COM IPC. Each IPC mechanism is used for a particular purpose and details of each are discussed in the sections below.

2.5.1. SHARED MEMORY IPC

The shared memory IPC is used by the broker and sandboxed process for sharing data and exchanging inter-process messages between their components. An example use of the shared memory IPC is when the *CShellBrowser2* component in the sandboxed process wanted to send a message to the *CBrowserFrame* component in the broker process.

BROKER INITIALIZATION

Upon broker process startup, three shared memory sections (internally called *Spaces*) are created in a private namespace. The name of these shared memory sections are as follows:

- *IsoScope_<broker_pid_hex>\IsoSpaceV2_ScopeTrusted*
- *IsoScope_<broker_pid_hex>\IsoSpaceV2_ScopeLILNAC*
- *IsoScope_<broker_pid_hex>\IsoSpaceV2_ScopeUntrusted*

The broker process also creates messaging events so that it can be notified if a message destined to it is available in the shared memory sections. The messaging events are created in a private namespace with the format "*IsoScope_<broker_pid_hex>\iso_sm_e_<broker_pid_hex>_<broker_iso_process_hex>_<iso_integrity_hex>*".
Where *broker_iso_process_hex* is the ID of the *IsoProcess Artifact* (*Artifacts* are further discussed below) that contains the broker process information and *iso_integrity_hex* is a value pertaining to a trust level (the value 1 is used if the event can be accessed by an untrusted AppContainer-sandboxed IE process).

SANDBOXED INITIALIZATION

Next, when the sandboxed process starts, the sandboxed process opens the shared memory sections previously created by the broker process with the following access rights:

- *IsoScope_<broker_pid_hex>\IsoSpaceV2_ScopeTrusted* (Read)
- *IsoScope_<broker_pid_hex>\IsoSpaceV2_ScopeLILNAC* (Read/Write)
- *IsoScope_<broker_pid_hex>\IsoSpaceV2_ScopeUntrusted* (Read/Write)

Similar to the broker process, the sandboxed process creates a messaging event so that it can be notified if a message destined to it is available in the shared memory sections. The messaging event is created in a private namespace with the format "*IsoScope_<broker_pid_hex>\ iso_sm_e_<broker_pid_hex>_<sandboxed_iso_process_hex>_<iso_integrity_hex>*".

ARTIFACTS

Data communicated or shared between processes are called *Artifacts*. There are different types of *Artifacts* and the following are the identified types of *Artifacts* so far (including their unique ID and purpose):

- *SharedMemory* (0x0B) – Shared data
- *IsoMessage* (0x0D) – IPC message
- *IsoProcess* (0x06) – Process information
- *IsoThread* (0x08) - Thread information

The *Artifacts* are stored in the shared memory via several layer of structures, these structures are described in the next sections.

SPACE STRUCTURE

The top level structure of each shared memory is called a *Space*. A *Space* contains multiple *Containers* with each *Container* designed to hold a specific type of *Artifact*.

Offset	Size	Description
00	dw	?
02	dw	?
04	dd	Unique Space Index
08	dd	?
0C	dd	Creator process ID
10	dd	?
14	dd	?
18	dd	?
1C	dd	?
20	dd	?
24	dd	?
28	Container[…]	Array of Container

CONTAINER STRUCTURE

A *Container* is a block of memory that holds an array of *ContainerEntry* which in turn contains the actual *Artifacts*. The following is the structure of a *Container*, it tracks which *ContainerEntry* is available for use:

Offset	Size	Description
00	dw	Next Available ContainerEntry Index
02	dw	?
04	dd	?
08	dd	?
0C	dw	?
0E	dw	Maximum Number of ContainerEntry
10	ContainerEntry […]	Array of ContainerEntry

There are instances where a *ContainerEntry* is needed but there is none left available in the *Container*. In these cases, an "expansion" *Container* is created in a separate shared memory section which is named as follows:

- *<original shared memory section name>_<unk_hex>:<unk_hex>_<unk_hex>*

CONTAINERENTRY STRUCTURE

Each *ContainerEntry* acts as a header for the *Artifact* it holds, wherein it tracks the type and the reference count of the *Artifact*. The following is the structure of a *ContainerEntry*:

Offset	Size	Description
00	db	?
01	db	Artifact Type
02	dw	?
04	dd	Reference Count
08	Varies	Artifact

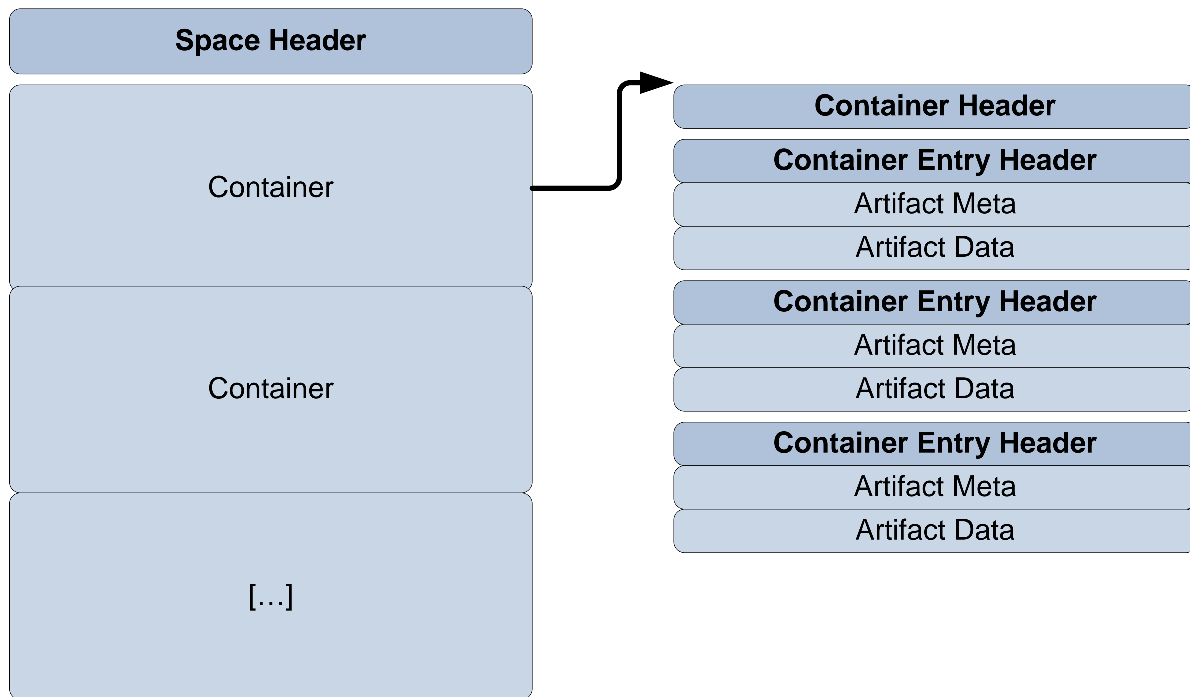
ARTIFACT (ISOARTIFACT) STRUCTURE

Finally, each *Artifact* has a 12 byte metadata described below. The actual *Artifact* data is stored starting at offset 0xC of the *Artifact* structure.

Offset	Size	Description
00	db	Artifact Type
01	db	?
02	dw	?
04	dd	Unique Artifact ID
08	dd	Creator Process IsoProcess Artifact ID or Destination Process IsoProcess Artifact ID (if Artifact type is IsoMessage)
0C	Varies	(Artifact Data)

SHARED MEMORY ("SPACE") STRUCTURE ILLUSTRATION

The illustration below shows how the shared memory (*Space*) structure is laid out:



SENDING A MESSAGE

Sending a message involves a source process allocating an *IsoMessage Artifact* in the shared memory. The destination process is then notified that an *IsoMessage* is available in the shared memory by setting the messaging event.

RECEIVING A MESSAGE

When the destination process is notified via the messaging event, the function *iertutil!CIsoSpaceMsg::MessageReceivedCallback()* is invoked. *iertutil!CIsoSpaceMsg::MessageReceivedCallback()* then looks for an *IsoMessage Artifact* in the *Spaces*. All *IsoMessage Artifacts* destined to the process are dispatched to the appropriate handlers by posting a message to a thread's message queue.

2.5.2. COM IPC

COM IPC is the second IPC mechanism used by IE. It is used by the sandboxed process to call the services exposed by COM objects in the broker process. One example of a COM object in the broker process is the User Broker Object (2.6.1) which handles the launch-type APIs forwarded by IE Shims (2.3).

Initialization/Bootstrapping

To bootstrap the COM IPC, the broker process first marshals the *IEUserBroker* interface of the User Broker Object. The marshaled interface is stored in a structure called *IsoCreationProcessData* which, thru the shared memory IPC mechanism (2.5.1), is shared with the sandboxed process by storing it in a *SharedMemory Artifact*. The *Artifact* ID of the *SharedMemory Artifact* is passed by the broker process to the sandboxed process via the "*CREADAT*" command line switch.

When the sandboxed process loads, it uses the "*CREADAT"* switch to find the *SharedMemory Artifact* and retrieve the *IsoCreationProcessData* structure. The *IEUserBroker* interface is then unmarshalled from the *IsoCreationProcessData* structure and will become ready for use in the sandboxed process.

Once the IEUserBroker interface is unmarshalled, code running the sandboxed process can use the helper function *iertutil!CoCreateUserBroker()* and similarly named functions in *iertutil.dll* to retrieve a pointer to the *IEUserBroker* interface.

Instantiation of Broker COM Objects

The sandboxed process uses the *IEUserBoker->BrokerCreateKnownObject()* method to instantiate known/allowed broker COM objects (also known as "*Known Broker Objects*") in the broker process. These Known Broker Objects (2.6.2) are listed in the Services section.

2.6. Services

In this paper, the term service refers to any functionality exposed by the broker process that is reachable or callable from the sandboxed process. These services are typically code that needs to be executed in a higher-privileged level or code that needs to be executed in the context of the frame/broker process. An example service is *IEUserBroker->WinExec()*, a functionality exposed by the User Broker Object that is used by IE Shims to launch an elevated process.

2.6.1. User Broker Object Services

The sandboxed IE process uses the User Broker Object to perform privileged operations such as launching elevated processes/COM servers and instantiating known/allowed COM broker objects. As mentioned in the COM IPC section, the sandboxed process uses the *iertutil!CoCreateUserBroker()* and similarly named functions to retrieve an *IEUserBroker* interface pointer.

The COM class that implements the User Broker Object is the *ieframe!CIEUserBrokerObject* class. The following are the services exposed by the User Broker Object including the available interfaces and their corresponding methods:

Interface*	Method	Notes
IID_IEUserBroker **{1AC7516E-E6BB-4A69-B63F-E841904DC5A6}**	Initialize()	Returns the PID of the broker process.
	CreateProcessW()	Invoke *CreateProcessW()* in the

		context of the broker process as dictated by the elevation policies. Handles forwarded *CreateProcessW()* calls from IE Shims.
	WinExec()	Invoke *WinExec()* in the context of the broker process as dictated by the elevation policies. Handles forwarded *WinExec()* calls from IE Shims.
	BrokerCreateKnownObject()	Instantiate a "Known Broker Object" (2.6.2).
	BrokerCoCreateInstance()	Invoke *CoCreateInstance()* in the context of the broker process as dictated by the elevation policies. Handles forwarded *CoCreateInstance()* calls from IE Shims.
	BrokerCoCreateInstanceEx()	Invoke *CoCreateInstanceEx()* in the context of the broker process as dictated by the elevation policies. Handles forwarded *CoCreateInstanceEx()* calls from IE Shims.
	BrokerCoGetClassObject()	Invoke *CoGetClassObject()* in the context of the broker process as dictated by the elevation policies. Handles forwarded *CoGetClassObject()* calls from IE Shims.
IID_IERegHelperBroker {41DC24D8-6B81-41C4-832C-FE172CB3A582}	DoDelSingleValue()	Delete known/allowed registry key value.
	DoDelIndexedValue()	Delete known/allowed registry key value.
	DoSetSingleValue()	Set data of a known/allowed registry key value.
	DoSetIndexedValue()	Set data of a known/allowed registry key value.
	DoCreateKey()	Create a known/allowed registry key.

IID_IEAxInstallBrokerBroker **{B2103BDB-B79E-4474-8424-4363161118D5}**	BrokerGetAxInstallBroker()	Instantiate *"Internet Explorer Add-on Installer"* COM object which causes the launching of the *"%ProgramFiles%\Internet Explorer\ieinstal.exe"* COM server.

*Interface ID may change in newer IE updates as new interface methods are added or removed. Refer to *ieframe!CIEUserBrokerObject::QueryInterface()* for an updated list of interface IDs.

2.6.2. Known Broker Objects Services

As previously mentioned, the *IEUserBroker->CreateKnownBrokerObject()* service allows the sandboxed process to instantiate known/allowed COM objects in the broker process. These known/allowed COM objects provide additional services to the sandboxed process and are listed below:

COM Class and CLSID(s)	Interface*	Methods/Description/Notes (if any)
ieframe!CShdocvwBroker **CLSID_ShdocvwBroker** **{9C7A1728-B694-427A-94A2-A1B2C60F0360}** **CLSID_CShdocvwBrokerNoFrameAffinity** **{20CCEF1E-0185-41A5-A933-509C43B54F98}**	**IID_IShdocvwBroker** {A9968B49-EAF5-4B73-AA93-A25042FCD67A} In IE11: {FED6B29E-13A0-48FA-8835-093F6F419388}	Interface has large number of services called by various parts of the sandboxed code. An example service is the service that handles the forwarded *kernel32!CreateFileW()* API by IE Shims and displaying the Internet Options dialog box.
	IID_IEBrokerAttach {7673B35E-907A-449D-A49F-E5CE47F0B0B2}	Method: *AttachIEFrameToBroker()*
	IID_IHTMLWindow ServicesBroker {806D58DF-EE78-4630-9475-F9B337A2DFCB}	Methods: *Blur()*, *Focus()*, *BlurBrowser()*, *FocusBrowser()*
msfeeds!CLoriBroker **CLSID_FeedsLoriBroker** **{A7C922A0-A197-4AE4-8FCD-2236BB4CF515}**	**IID_ILoriEvents** {07AB1E58-91A0-450F-B4A5-A4C775E67359}	*"Microsoft Feeds LoRi Broker"*
msfeeds!CArbiterBroker **CLSID_FeedsArbiterLoRiBroker** **{34E6ABFE-E9F4-4DDF-895A-7350E198F26E}**	**IID_IArbiterBroker** {EDB9EF13-045C-4C0A-808E-3294C59703B4}	*"Microsoft Feeds Arbiter LoRi Broker"*
ieframe!CProtectedModeAPI	**IID_IProtectedModeAPI** {3853EAB3-ADB3-4BE8-9C96-	Handles the following Protected Mode API

CLSID_CProtectedModeAPI **{ED72F0D2-B701-4C53-ADC3-F2FB59946DD8}** **CLSID_CProtectedModeAPINoFrameAffinity** **{8C537469-1EA9-4C85-9947-7E418500CDD4}**	C883B98E76AD}	functions [24] *IEShowSaveFileDialog()*, *IESaveFile()*, *IERegCreateKeyEx()*, and *IERegSetValueEx()*
	IID_IEBrokerAttach {7673B35E-907A-449D-A49F-E5CE47F0B0B2}	Method: *AttachIEFrameToBroker()*
iertutil!SettingStore::CSettingsBroker **CLSID_CSettingsBroker** **{C6CC0D21-895D-49CC-98F1-D208CD71E047}**	**IID_ISettingsBroker** {EA77E0BC-3CE7-4CC1-928F-6F50A0CE5487}	*"Internet Explorer Settings Broker"*
ieframe!CRecoveryStore **CLSID_IERecoveryStore** **{10BCEB99-FAAC-4080-B2FA-D07CD671EEF2}**	**IID_IRecoveryStore** {A0CEE1E8-9548-425D-83E3-A47E69F39D30} In IE11: {5C333B75-A015-4564-8BF3-998CCC4FF9F5}	*"IE Recovery Store"*. This particular interface has a number of services.
	IID_ICookieJarRecoveryData {8FD276BB-5A9D-4FC0-B0BD-90FA7CF1283D}	Methods: *SetCookie()*, *SetCookiesFromBlob()*, *DeleteCookie()*, *EnumCookies()*, *ClearCookies()*
	IID_ICredentialRecoveryData {239D58CC-793C-4B64-8320-B51380087C0B}	Methods: *SetCredential()*, *SetCredentialFromBlob()*, *EnumCredentials()*, *ClearCredentials()*
wininet!WininetBroker **CLSID_WininetBroker** **{C39EE728-D419-4BD4-A3EF-EDA059DBD935}**	**IID_IWininetBroker** {B06B0CE5-689B-4AFD-B326-0A08A1A647AF}	*"WinInetBroker Class"*

*Interface ID may change in newer IE updates as new interface methods are added or removed. Refer to the corresponding *QueryInterface()* method of the listed COM classes for an updated list of interface IDs.

2.6.3. BROKER COMPONENTS MESSAGE HANDLERS

Finally, another set of broker functionality that are reachable or callable from the sandboxed process are the message handlers of broker components. These message handlers are invoked when inter-process messages via the shared memory IPC are received. Examples of broker component message handlers are:

- *ieframe!CBrowserFrame::_Handle*()*
- *ieframe!CDownloadManager::HandleDownloadMessage()*

Typically, message handlers will directly or indirectly invoke *iertutil!IsoGetMessageBufferAddress()* to retrieve the *IsoMessage Artifact* and then parse/use it according to an expected format.

2.7. SUMMARY: SANDBOX INTERNALS

The first part of this paper described in details the different mechanisms that make up the EPM sandbox. First discussed is the overall sandbox architecture which described in a high level each sandbox mechanism and how they are interconnected.

A new process isolation mechanism introduced in Windows 8, called AppContainer, is the main mechanism used by EPM to isolate and limit the privileges and capabilities of the sandboxed process.

Through API hooking, the IE Shims (Compatibility Layer) mechanism enables binary extensions to work on a low-privileged environment by redirecting operations to writable locations, it is also used for forwarding API calls to the broker in order to support certain functionalities, furthermore, it is also the mechanism used to apply elevation policies to operations that would result in the launching of a process or a COM server.

For inter-process communication, the sandboxed and the broker process use two types of IPC mechanism to communicate – COM IPC which is used by the sandboxed process to perform calls to COM objects in the broker process, and shared memory IPC which is used by components of the sandboxed process and the broker process to exchange inter-process messages and to share data.

Finally, the broker processes exposes several services which are callable from the sandboxed process via the IPC mechanisms. The services are typically code that needs to run in a higher-privilege level or code that needs to run in the context of the frame/broker process.

3. SANDBOX SECURITY

After discussing how the EPM sandbox works, in this second part of the paper, we'll take a look at the security aspect of the EPM sandbox. This part is divided into two sections – the first section discusses the limitations or weaknesses of the EPM sandbox, and the second section discusses the different ways how code running in the sandboxed process can gain additional privileges or achieve code execution in a more privileged context.

3.1. SANDBOX LIMITATIONS/WEAKNESSES

This section lists the limitations or weaknesses of the EPM sandbox. It answers the important question "what can malicious code still do or access once it is running inside the sandboxed process?" The most likely reason for the existence of some of these limitations/weaknesses is because of compatibility reasons or that addressing them would require significant development effort. Nonetheless, the items discussed here are current limitations/weaknesses, future patches or EPM improvements may address some, if not all of them.

3.1.1. FILE SYSTEM ACCESS

In a default Windows 8 install, the sandboxed process can still list and read most files from the following folders (including their subfolders) because of the read access control entry for *ALL APPLICATION PACKAGES*:

- *%ProgramFiles% (C:\Program Files)*
- *%ProgramFiles(x86)% (C:\Program Files (x86))*
- *%SystemRoot% (C:\Windows)*

The reason for the read access control entry for *ALL APPLICATION PACKAGES* in most system/common files and folders is for compatibility with AppContainer-sandboxed applications (such as IE running in EPM and Windows Store Apps), because they need access to system/common resources in order to properly launch or operate [25].

The consequence of the read access to system/common files and folders is that the sandboxed code can list installed applications in the system - information which can be used for future attacks. Additionally, configuration files or license keys used by 3rd party application (both of which may contain sensitive information) can be stolen if they are stored in the above folders.

Additionally, EPM uses the following AppContainer-specific folders to store EPM cache files and cookies:

- *%UserProfile%\AppData\Local\Packages\<AppContainer Name>\AC\InetCache*
- *%UserProfile%\AppData\Local\Packages\<AppContainer Name>\AC\InetCookies*

Since the AppContainer process has full access to files on the above folders, the sandboxed process will be able to read potentially sensitive information, especially from cookies which may contain authentication information to websites. Note that EPM uses a separate AppContainer when browsing private network sites (i.e. when private network access is turned on), therefore, cookies and cache files for private network sites will be stored in a different AppContainer-specific location. Also, cookies and cache files for sites visited with EPM turned off are also stored in a different location [26].

Finally, the sandboxed code can take advantage of its read/write access to files on the *%UserProfile\Favorites* folder (due to the access control entry for the *internetExplorer* capability), whereby, it can potentially modify all shortcut files so that they will point to an attacker-controlled site.

3.1.2. REGISTRY ACCESS

Most of the subkeys in the following top-level registry keys can still be read by the sandboxed process because of the read access control entry for *ALL APPLICATION PACKAGES*:

- *HKEY_CLASSES_ROOT*
- *HKEY_LOCAL_MACHINE*
- *HKEY_CURRENT_CONFIG*

Similar to system/common files and folders, the reason for the read access control entry for *ALL APPLICATION PACKAGES* in most system/common registry keys is for compatibility with AppContainer-sandboxed applications.

The consequence of the read access to system/common registry keys is that the sandboxed process will be able to retrieve system and general application configuration/data which may contain sensitive information or may be used in future attacks. Example of such registry keys are:

- *HKLM\Software\Microsoft\Internet Explorer\Low Rights\ElevationPolicy* (IE Elevation Policies)
- *HKLM\Software\Microsoft\Windows NT\CurrentVersion* (Registered Owner, Registered Organization, etc.)

Furthermore, there are also several user-specific configuration/information contained in the subkeys of *HKEY_CURRENT_USER* which are still readable from the sandboxed process. Read access to these subkeys is due to the read access control entry for *ALL APPLICATION PACKAGES* or the *internetExplorer* capability. Examples of these registry keys are:

- Readable via the *ALL APPLICATION PACKAGES* access control entry:
  - *HKCU\Software\Microsoft\Windows\CurrentVersion\Explorer\RunMRU* (Run command MRU)
  - *HKCU\Software\Microsoft\Windows\CurrentVersion\Explorer\RecentDocs* (Recent Items)

- Readable via the *internetExplorer* capability or the ALL APPLICATION PACKAGES access control entry:
  - *HKCU\Software\Microsoft\Internet Explorer\TypedURLs* (Typed URLs in IE)

Access to user-specific locations in the registry (*HKCU*) and the file system (*%UserProfile%*) could potentially be further locked down by EPM using a restricted token. However, it would mean that access to resources that the EPM-sandboxed process normally has direct access to (such as AppContainer-specific locations and those have an access control entry for the *internetExplorer* capability) would need to be brokered.

3.1.3. CLIPBOARD ACCESS

Because clipboard read/write restrictions are not set in the job object associated to the sandboxed process, the sandboxed process can still read from and write to the clipboard. Additionally, as discussed in the Sandbox Restrictions section, EPM does not support window station isolation, therefore, the sandboxed process shares the clipboard with other processes that are associated to the same window station.

A caveat, however, is that the AppContainer process should be the process that is currently actively receiving keyboard input, otherwise, *user32!OpenClipboard()* will fail with an access denied error. One way for this to happen is to coerce the user to press a key while a window or control owned by the AppContainer process is focused.

Nonetheless, because the sandboxed process still has access to the clipboard, the sandboxed process will be able to read potentially sensitive information from the clipboard (such as passwords – if the user uses an insecure

password manager that does not regularly clear the clipboard). Moreover, clipboard write access can lead to arbitrary command execution if malicious clipboard content is inadvertently pasted to a command prompt. Also, if a higher-integrity application fully trusts and uses the malicious clipboard content, its behavior can be controlled, potentially leading to a sandbox escape - these are described by Tom Keetch in the paper *"Practical Sandboxing on the Windows Platform"* [12].

3.1.4. Screen Scraping And Screen Capture

Another information disclosure attack that can still be performed by the sandboxed process is *screen scraping* [27]. Because EPM does not support desktop isolation and the UILIMIT_HANDLE restriction is not set in the job object associated to the sandboxed process, the sandboxed process can still send messages that are not blocked by UIPI to windows owned by other processes. By sending allowed query messages such as *WM_GETTEXT*, the sandboxed process will be able to capture information from windows or controls of other applications.

Additionally, another interesting information disclosure attack that is possible is performing a screen capture. The resulting screen capture data may also contain potentially sensitive information or information that can be used in future attacks, such as what applications the user usually uses (via pinned programs in the taskbar) or what security software is running (via the tray icons), etc.

3.1.5. Network Access

Finally, because of the *internetClient* capability, the sandboxed process can still connect to Internet and public network endpoints. This allows malicious code running in the sandboxed process to communicate and send stolen information to a remote attacker. Additionally, this capability can be leveraged by a remote attacker in order to use the affected system to connect or attack other Internet/public network endpoints, thereby concealing the real source of the connection or the attack.

3.2. Sandbox Escape

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

3.2.1. Local Elevation of Privilege (EOP) Vulnerabilities

A common escape vector among sandbox implementations is to elevate privileges by exploiting Local Elevation of Privilege (EoP) vulnerabilities. Exploiting local EoP vulnerabilities, especially those that can result in arbitrary code execution in kernel mode are an ideal way to bypass all the restrictions set on the sandboxed code. With multiple kernel attack vectors [28] such as system calls, parsers and device objects which are available to a sandboxed code, we can expect that local EoP vulnerabilities will become more valuable as more and more widely-deployed applications are being sandboxed.

A recent example of using a kernel mode vulnerability to escape a browser sandbox is CVE-2013-1300 [29], a vulnerability in Win32k that Jon Butler and Nils discovered and used to escape the Google Chrome sandbox in Pwn2Own 2013.

3.2.2. Policy/Permission Vulnerabilities

Policy/permission vulnerabilities involve permissive write-allowed sandbox policies or resource permissions (such as permissive ACLs) that can be leveraged by the sandboxed process to control the behavior of a higher-privileged

process. These also include elevation policies that could result in the execution of arbitrary code in a more privileged context.

An example policy vulnerability in IE is CVE-2013-3186 [30], a vulnerability discovered by Fermin Serna which involves an elevation policy that allows the execution of *msdt.exe* in medium integrity without prompt. The issue is that there are certain arguments that can be passed to *msdt.exe* that would result in the execution of arbitrary scripts at medium integrity.

3.2.3. POLICY CHECK VULNERABILITIES

Because the elevation policy checks in IE determine how executables will run (e.g. with prompt or without prompt) and at what privilege level, weaknesses, particularly logic vulnerabilities that lead to an incorrect policy check result is a vector for sandbox escape. Typically, these policy check logic vulnerabilities are easier to exploit than memory corruption vulnerabilities since exploiting them will just involve evading the policy checks via specially formed resource names, execution of the privileged operation such as spawning a given executable or writing to a privileged resource will be done by the broker using its own code.

In the course of my research for this paper, I discovered a policy check vulnerability in IE (CVE-2013-4015) [31] which allows execution of any executable in medium integrity without prompt. Specifically, the vulnerability exists in *ieframe!GetSanitizedParametersFromNonQuotedCmdLine(),* a helper function used (directly and indirectly) by the User Broker Object to parse a non-quoted command line into its application name and argument components, and the result of which are eventually used in an elevation policy check.

By using a whitespace other than the space character to delimit an application name and its arguments, *ieframe!GetSanitizedParametersFromNonQuotedCmdLine()* will be misled to extract an incorrect application name.

Consider the following command line:

```
C:\Windows\System32\cmd.exe\t\..\notepad.exe /c calc.exe
```

In the above case, *ieframe!GetSanitizedParametersFromNonQuotedCmdLine()* will treat the string "*C:\Windows\System32\cmd.exe\t\..\notepad.exe*" (normalized to "*C:\Windows\System32\notepad.exe*") as the application name, and the string "*/c calc.exe*" as the argument. Because of a default elevation policy, the executable "*C:\Windows\System32\notepad.exe*" is allowed to execute in medium integrity without prompt, thus, the application name will pass the elevation policy check and the command line is passed to the *kernel32!WinExec()* API by the User Broker Object.

However, when the command line is executed by *kernel32!WinExec()*, instead of spawning Notepad, the actual application that will be executed will be "*C:\Windows\System32\cmd.exe*" with the argument "*\..\notepad.exe /c calc.exe*". We reported this vulnerability to Microsoft and it was patched last July 2013 (MS13-055) [32].

Another example of a policy check vulnerability in another sandbox implementation is CVE-2011-1353 [33], a vulnerability that Paul Sabanal and I discovered (and also independently discovered by Zhenhua Liu) in the policy engine of the Adobe Reader X sandbox [34]. The vulnerability is due to lack of canonicalization of a resource name when evaluating registry deny-access policies. By using a registry resource name that contains additional backslashes, it is possible to bypass the registry-deny policy check, thereby, allowing a sandboxed code to write to a critical registry key.

3.2.4. SERVICE VULNERABILITIES

The services exposed by higher-privileged processes make up a large part of the attack surface in a sandbox implementation, including IE. To support features or functionality that needs to be executed in a higher privilege level, the broker process or another process running with a higher privilege level would need to implement those capabilities and expose them to the sandboxed process as services. From a developer's perspective, writing the code of these services requires additional precautions since they run in the context of a higher-privileged process and uses untrusted data as input.

A notable example of a service vulnerability in another sandbox implementation is CVE-2013-0641 [35], a Reader sandbox vulnerability that was leveraged by the first in-the-wild Reader sandbox escape exploit. The vulnerability is in the service that handles the execution of *user32!GetClipboardFormatNameW()* in the context of the broker process. Because of an incorrect output buffer size being passed to the API, a buffer overflow can be triggered in the broker process by the sandboxed process. By overwriting a virtual function table pointer, the exploit was able to control the execution flow of the broker process, eventually leading to the execution of an attacker-controlled code in the higher-privileged broker process.

Two more examples of a service vulnerability are CVE-2012-0724 and CVE-2012-0725 [33], two vulnerabilities that Paul Sabanal and I discovered (and also independently discovered by Fermin Serna) in the Chrome Flash sandbox. The vulnerabilities were due to services in a broker process fully trusting a pointer they received from the sandboxed process. Because the attacker-controlled pointer is dereferenced for a function call, execution flow of the broker process can be controlled and arbitrary code execution in the higher-privileged broker process can be achieved.

As discussed and listed in the Services section (2.6), there are numerous services exposed to the IE sandboxed process. From an attacker's perspective, these exposed services are potential opportunities for sandbox escape and therefore, we can expect attackers will be (or currently are) auditing these services for vulnerabilities. We can anticipate that in the future, additional services would be exposed to the sandboxed process in order to accommodate new browser features and functionality.

3.3. SUMMARY: SANDBOX SECURITY

In terms of limitations or weakness, there are still some of types of potentially sensitive or personal information that can be accessed from the file system and the registry because of how the access control list of certain files, folders and registry keys are setup. EPM browser cookies and cache files stored in the AppContainer-specific folder which also contains sensitive information can also be accessed. Additionally, EPM currently does not protect against screen scraping, screen capture and clipboard access. Finally, because the sandboxed process can still connect to Internet and pubic network endpoints, stolen information can be sent to a remote attacker.

In terms of sandbox escape, there are multiple options for an attacker to gain code execution in a more privileged context. Ranging from generic attacks such as exploiting local elevation of privilege vulnerabilities and leveraging write-allowed resource permissions, to taking advantage of the sandbox mechanisms, such as evading policy checks, taking advantage of sandbox policy issues, and attacking the services exposed by the broker process.

4. CONCLUSION

Enhanced Protected Mode is a welcome incremental step done by Microsoft in improving IE's sandbox. Its use of the new AppContainer process isolation mechanism will certainly help in preventing theft of personal user files and corporate assets from the network. However, as discussed in this paper, there are still some limitations or weaknesses in the EPM sandbox that can lead to the exfiltration of some other types of potentially sensitive or personal information and there are still some improvements that can be done to prevent the attacks described in this paper.

Also, the new AppContainer process isolation mechanism is a great addition to Windows 8 as it gives vendors an additional option in developing sandboxed applications. For security researchers, AppContainer is an interesting feature to further look at as there are certainly more isolation/restriction schemes that it provides than what are listed in this paper.

Finally, I hope this paper had helped you, the reader, understand the inner workings of IE10's Enhanced Protected Mode sandbox.

5. BIBLIOGRAPHY

[1] M. Silbey and P. Brundrett, "MSDN: Understanding and Working in Protected Mode Internet Explorer," [Online]. Available: http://msdn.microsoft.com/en-us/library/bb250462(v=vs.85).aspx.

[2] T. Keetch, "Escaping from Protected Mode Internet Explorer," [Online]. Available: http://archive.hack.lu/2010/Keetch-Escaping-from-Protected-Mode-Internet-Explorer-slides.ppt.

[3] J. Drake, P. Mehta, C. Miller, S. Moyer, R. Smith and C. Valasek, "Browser Security Comparison - A Quantitative Approach," [Online]. Available: http://files.accuvant.com/web/files/AccuvantBrowserSecCompar_FINAL.pdf.

[4] A. Zeigler, "Enhanced Protected Mode," [Online]. Available: http://blogs.msdn.com/b/ie/archive/2012/03/14/enhanced-protected-mode.aspx.

[5] A. Zeigler, "IE8 and Loosely-Coupled IE (LCIE)," [Online]. Available: http://blogs.msdn.com/b/ie/archive/2008/03/11/ie8-and-loosely-coupled-ie-lcie.aspx.

[6] Ollie, "Windows 8 App Container Security Notes - Part 1," [Online]. Available: http://recxltd.blogspot.com/2012/03/windows-8-app-container-security-notes.html.

[7] A. Ionescu, "Windows 8 Security and ARM," [Online]. Available: https://ruxconbreakpoint.com/assets/Uploads/bpx/alex-breakpoint2012.pdf.

[8] A. Allievi, "Securing Microsoft Windows 8: AppContainers," [Online]. Available: http://news.saferbytes.it/analisi/2013/07/securing-microsoft-windows-8-appcontainers/.

[9] S. Renaud and K. Szkudlapski, "Windows RunTime," [Online]. Available: http://www.quarkslab.com/dl/2012-HITB-WinRT.pdf.

[10] Microsoft, "MSDN: App capability declarations (Windows Store apps)," [Online]. Available: http://msdn.microsoft.com/en-us/library/windows/apps/hh464936.aspx.

[11] Microsoft, "MSDN: SECURITY_CAPABILITIES Structure," [Online]. Available: http://msdn.microsoft.com/en-us/library/windows/desktop/hh448531(v=vs.85).aspx.

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

[13] T. Mandt, "Smashing the Atom: Extraordinary String Based Attacks," [Online]. Available: www.azimuthsecurity.com/resources/recon2012_mandt.pptx.

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

[15] E. Barbosa, "Windows Vista UIPI (User Interface Privilege Isolation)," [Online]. Available: http://www.coseinc.com/en/index.php?rt=download&act=publication&file=Vista_UIPI.ppt.pdf.

[16] Wikipedia, "Shatter attack," [Online]. Available: http://en.wikipedia.org/wiki/Shatter_attack.

[17] Microsoft, "MSDN: How to set network capabilities," [Online]. Available: http://msdn.microsoft.com/en-us/library/windows/apps/hh770532.aspx.

[18] The Chromium Authors, "The Chromium Projects - Sandbox," [Online]. Available: http://www.chromium.org/developers/design-documents/sandbox.

[19] 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.

[20] 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.

[21] 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.

[22] E. Lawrence, "Brain Dump: Shims, Detours, and other "magic"," [Online]. Available: http://blogs.msdn.com/b/ieinternals/archive/2012/07/31/internet-explorer-compatibility-detours-shims-virtualization-for-toolbars-activex-bhos-and-other-native-extensions.aspx.

[23] E. Lawrence, "Enhanced Protected Mode and Local Files," [Online]. Available: http://blogs.msdn.com/b/ieinternals/archive/2012/06/20/loading-local-files-in-enhanced-protected-mode-in-internet-explorer-10.aspx.

[24] Microsoft, "MSDN: Protected Mode Windows Internet Explorer Reference," [Online]. Available: http://msdn.microsoft.com/en-us/library/ms537312(v=vs.85).aspx.

[25] Microsoft, "Win8: App: Modern: Apps fail to start if default registry or file permissions modified," [Online]. Available: http://support.microsoft.com/kb/2798317.

[26] E. Lawrence, "Understanding Enhanced Protected Mode," [Online]. Available: http://blogs.msdn.com/b/ieinternals/archive/2012/03/23/understanding-ie10-enhanced-protected-mode-network-security-addons-cookies-metro-desktop.aspx.

[27] Wikipedia, "Data scraping - Screen scraping," [Online]. Available: http://en.wikipedia.org/wiki/Data_scraping#Screen_scraping.

[28] 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.

[29] J. Butler and Nils, "MWR Labs Pwn2Own 2013 Write-up – Kernel Exploit," [Online]. Available: https://labs.mwrinfosecurity.com/blog/2013/09/06/mwr-labs-pwn2own-2013-write-up---kernel-exploit/.

[30] F. Serna, "CVE-2013-3186 - The case of a one click sandbox escape on IE," [Online]. Available: http://zhodiac.hispahack.com/index.php?section=blog&month=8&year=2013.

[31] IBM Corporation, "Microsoft Internet Explorer Sandbox Bypass," [Online]. Available: http://www.iss.net/threats/473.html.

[32] Microsoft, "Microsoft Security Bulletin MS13-055 - Critical," [Online]. Available: https://technet.microsoft.com/en-us/security/bulletin/ms13-055.

[33] P. Sabanal and M. V. Yason, "Digging Deep Into The Flash Sandboxes," [Online]. Available: https://media.blackhat.com/bh-us-12/Briefings/Sabanal/BH_US_12_Sabanal_Digging_Deep_WP.pdf.

[34] 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.

[35] M. V. Yason, "A Buffer Overflow and Two Sandbox Escapes," [Online]. Available: http://securityintelligence.com/a-buffer-overflow-and-two-sandbox-escapes/.