

Cutting Edge Microsoft Browser Security – From People Who Owned It

Chuanda Ding (@FlowerCode_)

Zhipeng Huo (@R3dF09)

Wei Wei (@Danny__Wei)



腾讯安全玄武实验室
TENCENT SECURITY XUANWU LAB

Abstract

We have seen too much focus on finding Win32k bugs. Is the sandbox itself too secure to escape?

Microsoft Edge, the new default browser for Windows 10, is heavily sandboxed.

In fact, it is probably the only browser with its main process running inside a sandbox. Microsoft even goes to great length to design and implement platform security features exclusively for Microsoft Edge.

In this paper, we will take a deep dive into the Microsoft Edge security architecture. This includes sandbox initialization, browser broker implementation, inter-process communication, and renderer security isolation.

We will present two logical sandbox escape bug chain consists of three bugs for Microsoft Edge.

One of which we have used in Pwn2Own, and the other two are completely new. They are entirely different from memory corruption bugs, as all we have done is abusing normal features implemented in the browser and operating system.

Table of Contents

Abstract	2
1. Introduction.....	4
1.1. Microsoft Edge	4
1.2. Universal Windows Platform.....	4
1.2.1. AppContainer.....	5
1.2.2. Child AppContainer.....	7
1.2.3. The Brokers.....	7
1.3. Microsoft Edge Architecture	8
2. Process Startup and Privilege Separation.....	10
2.1. Manager Process Startup	10
2.2. Content Process Startup	12
3. Inter-Process Communication	16
3.1. RPC	16
3.2. COM.....	17
3.2.1. Overview.....	17
3.2.2. COM Security	20
3.3. LCIE IPC.....	21
3.3.1. Overview.....	21
3.3.2. LCIE IPC Message Security	24
4. Vulnerabilities.....	25
4.1. Sandbox Escape Chain 1.....	25
4.1.1. Browser Broker Bug.....	25
4.2. Sandbox Escape Chain 2.....	31
4.2.1. Flash Broker Bug.....	31
4.2.2. SOP Bypass.....	38
4.2.3. Exploit Chain	41
4.2.4. Patches	42
5. Conclusion	43
Acknowledgments.....	43
References.....	43

1. Introduction

1.1. Microsoft Edge

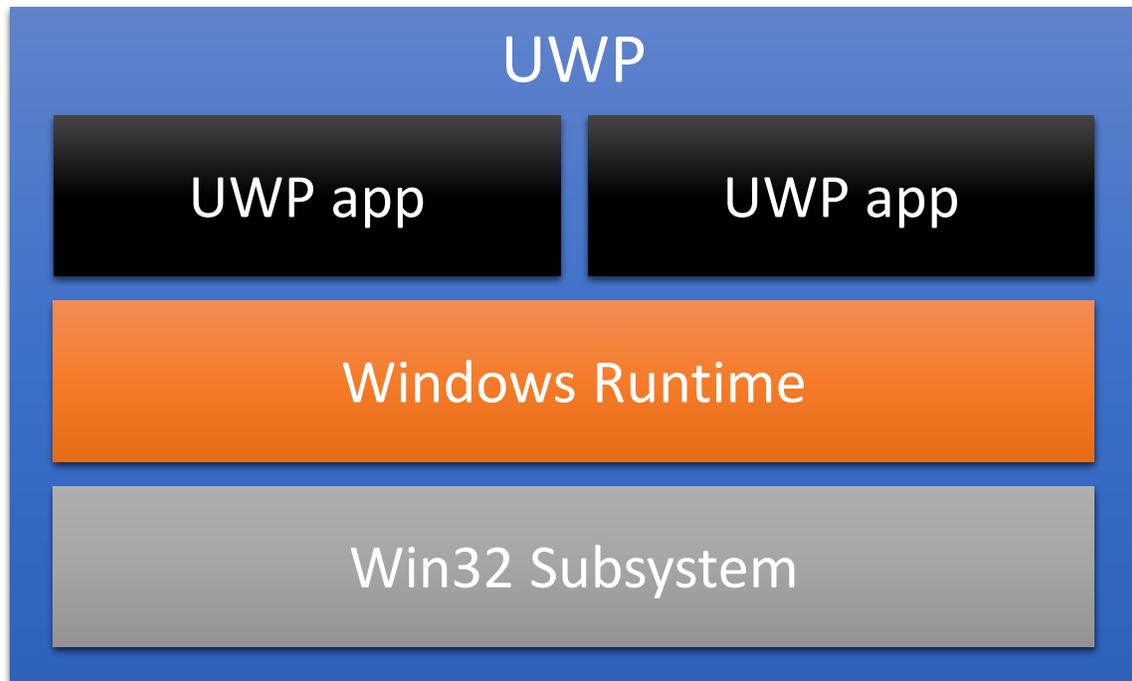
Microsoft Edge is a new web browser developed by Microsoft. It is the default web browser on Windows 10, replacing Internet Explorer. According to Microsoft, it is a fast and secure browser designed for Windows 10 and “the faster way to get things done on the web”. The largest change in Microsoft Edge security is that the new browser is a Universal Windows Platform app. This fundamentally changes the process model, so that both the manager process, and the assorted content processes (renderers), run within AppContainer sandboxes^[1]. Making browser runs inside a sandbox will prevent arbitrary code execution vulnerabilities from affecting the underlying OS. Microsoft Edge has been a target of Pwn2Own contest starting from 2016.

1.2. Universal Windows Platform

UWP apps are built with Windows Runtime APIs (WinRT API), which was introduced in Windows 8. WinRT is based on an enhanced version of COM. Language projections are provided for C++, .NET languages, and JavaScript. These projections make it easy to access WinRT types, methods, properties, and events from developers’ familiar environments.^[2]

“WinRT” stands for “Windows Runtime”. However, this is not a runtime in the same way that .NET Runtime or Oracle’s Java Runtime Environment is a runtime. WinRT is still layered on top of the Windows subsystem DLLs. It has a set of libraries offering a range of services through APIs.^[3]

Following diagram shows the relationship between UWP app, WinRT, and Win32.

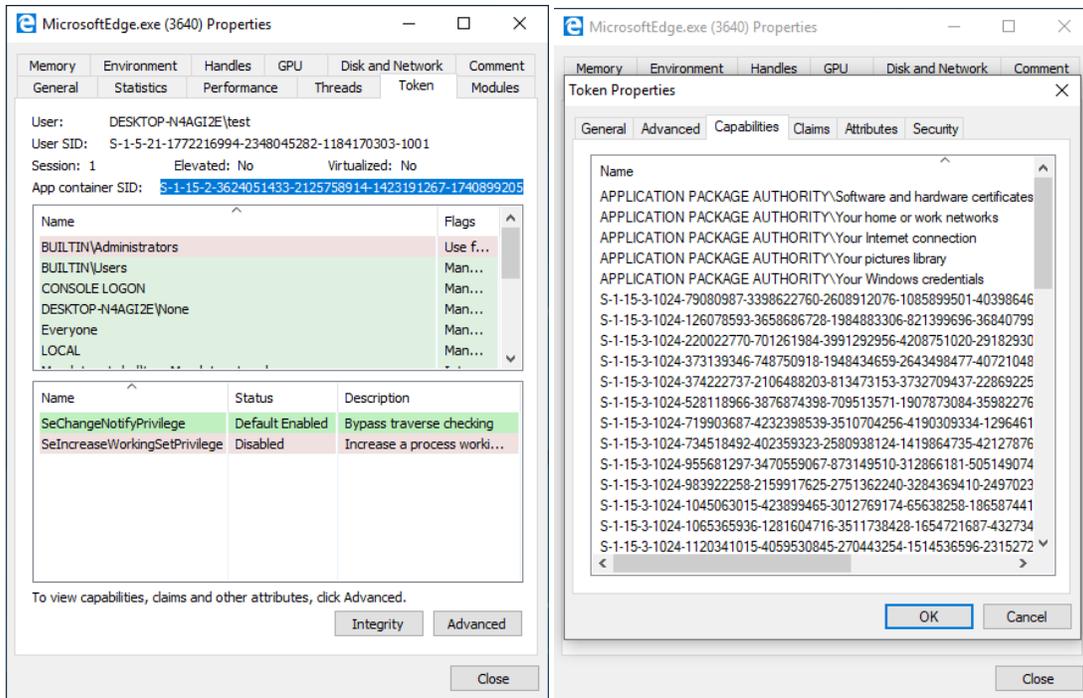


Universal Windows Platform

1.2.1. AppContainer

From the security perspective, UWP apps run in a new type of sandbox called AppContainer. Isolation is the primary goal of AppContainer. It introduces many isolation technologies, such as securable object restrictions, object namespace isolation, global atom table restrictions, enhanced UIPI, and network isolation. All these isolation techniques use the AppContainer's LowBox token for access checking.

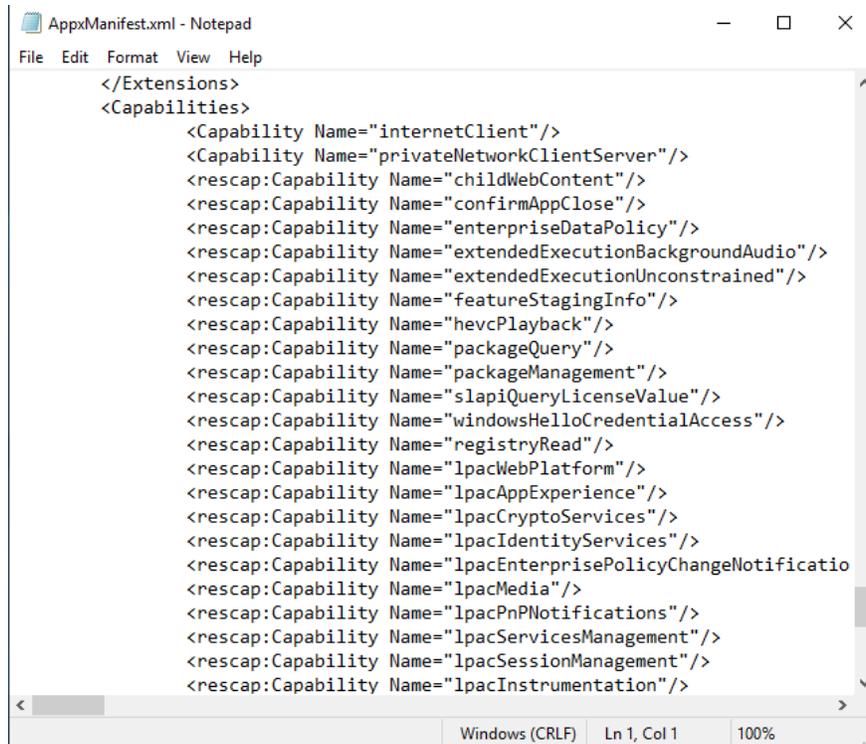
The LowBox token was also introduced in Windows 8. It has three key members: Integrity level, AppContainer SID and Capability SIDs. All UWP apps run at low integrity level with a special SID (begin with S-1-15-2) and a set of Capability SIDs.



Microsoft Edge AppContainer SID and Capability SIDs

Windows use these members to determine if an UWP app has access to resources in the same user account. Unlike the IE's Protected Mode, which only uses integrity level to limits write and execute access to resources that belongs to same user account, AppContainer can also be used for limiting read access.

Developers can also configure UWP apps' **AppxManifest.xml** file to restrict their access more granularly. For example, the access to internet, camera, clipboard, shared user certificates, etc. These restrictions are implemented with Capability SIDs. User security is protected with AppContainer sandbox.

A screenshot of a Notepad window titled 'AppxManifest.xml - Notepad'. The window displays XML code for the 'Capabilities' section of an AppX manifest. The code lists various capabilities, including 'internetClient', 'privateNetworkClientServer', and several 'rescap:Capability' entries with names like 'childWebContent', 'confirmAppClose', 'enterpriseDataPolicy', etc. The status bar at the bottom indicates 'Windows (CRLF)', 'Ln 1, Col 1', and '100%' zoom.

```
</Extensions>
<Capabilities>
  <Capability Name="internetClient"/>
  <Capability Name="privateNetworkClientServer"/>
  <rescap:Capability Name="childWebContent"/>
  <rescap:Capability Name="confirmAppClose"/>
  <rescap:Capability Name="enterpriseDataPolicy"/>
  <rescap:Capability Name="extendedExecutionBackgroundAudio"/>
  <rescap:Capability Name="extendedExecutionUnconstrained"/>
  <rescap:Capability Name="featureStagingInfo"/>
  <rescap:Capability Name="hevcPlayback"/>
  <rescap:Capability Name="packageQuery"/>
  <rescap:Capability Name="packageManagement"/>
  <rescap:Capability Name="slapiQueryLicenseValue"/>
  <rescap:Capability Name="windowsHelloCredentialAccess"/>
  <rescap:Capability Name="registryRead"/>
  <rescap:Capability Name="lpacWebPlatform"/>
  <rescap:Capability Name="lpacAppExperience"/>
  <rescap:Capability Name="lpacCryptoServices"/>
  <rescap:Capability Name="lpacIdentityServices"/>
  <rescap:Capability Name="lpacEnterprisePolicyChangeNotificatio
  <rescap:Capability Name="lpacMedia"/>
  <rescap:Capability Name="lpacPnPNotifications"/>
  <rescap:Capability Name="lpacServicesManagement"/>
  <rescap:Capability Name="lpacSessionManagement"/>
  <rescap:Capability Name="lpacInstrumentation"/>
```

MicrosoftEdge's Capabilities defined in AppxManifest.xml

1.2.2. Child AppContainer

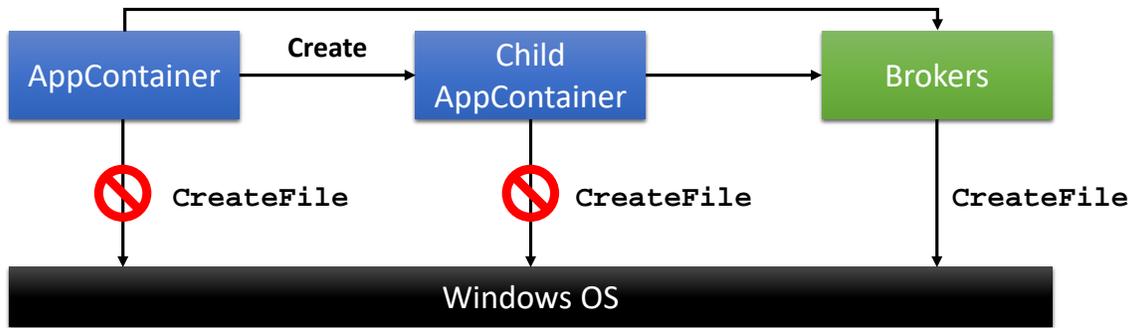
One AppContainer is enough for most UWP apps. This is true for all other pre-installed apps, such as *Calculator*, *Maps* and *OneNote*. However, an UWP app can also create another type of AppContainer called child AppContainer, if it wants to create its own nested AppContainer to further lock down the app.

Microsoft Edge needs to run several types of renderers with different privileges. It creates child AppContainers with different capabilities for several types of renderers.

A child AppContainer has four more RIDs in addition to AppContainer SID's eight RIDs to uniquely identify it. Chapter 2.1.2 will detail the creation process of child AppContainer.

1.2.3. The Brokers

As we mentioned earlier, UWP app runs in AppContainer and built with the WinRT API. Windows Runtime is responsible for the resource operations of UWP app. Many WinRT APIs runs in a broker called Runtime Broker, since many resource operations require permissions higher than AppContainer, such as reading and writing user files, accessing the clipboard, etc.



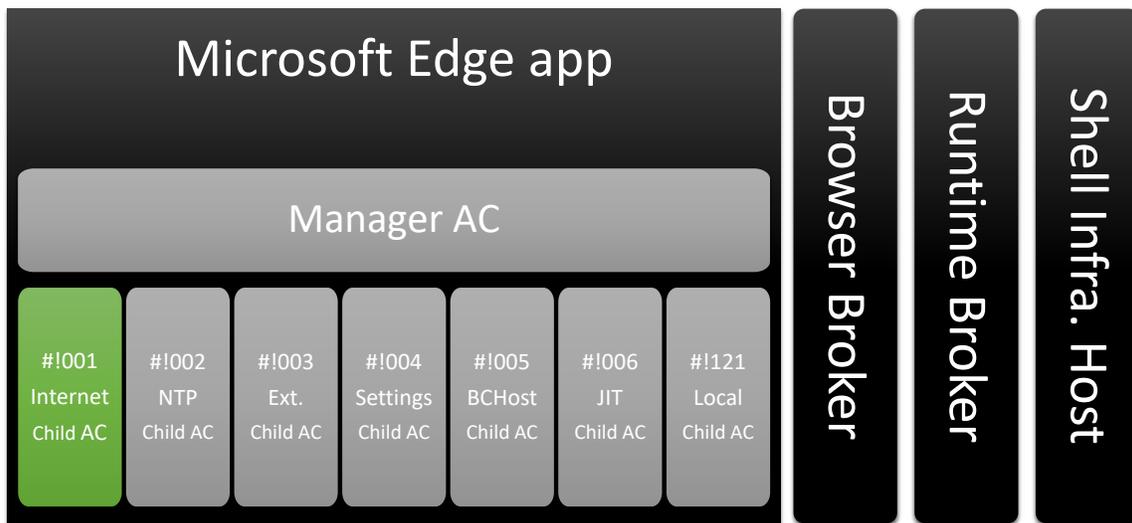
Brokers

Browser also needs a broker for privileged operations, such as changing permission of downloaded files or access other services.

There are already many articles detailing UWP, AppContainer, Runtime Broker etc. You can learn more about them from referenced articles^{[2][3][4]}.

1.3. Microsoft Edge Architecture

The following diagram illustrates the architecture of *Microsoft Edge*.



Microsoft Edge Architecture

MicrosoftEdge.exe is the manager process, responsible for managing renderer processes and providing basic functions. Unlike other browsers, the manager process is a UWP app that runs in an AppContainer sandbox.

There are several types of renderers in *Microsoft Edge*. They are also called content processes, with process name **MicrosoftEdgeCP.exe**.

Renderer types

Moniker	Type
001	Internet zone renderer
002	New tab page renderer
003	Extensions page renderer
004	Settings page renderer
005	Flash Player enabled page renderer
006	Just-In-Time energy renderer
121	Local zone & Intranet zone renderer

Browser Broker, Runtime Broker and Shell Experience Host all runs with normal user privilege.

2. Process Startup and Privilege Separation

In this section we will talk about the startup of *Microsoft Edge* app. We are not going to cover every detail. What we really care about is privileges assigned to each Edge process, especially the Internet AC process.

2.1. Manager Process Startup

Manager process is the first started process during the activation of *Microsoft Edge*. It is responsible for creating child processes, also known as content process or renderer.

The following is startup sequence of manager process.

1. Activating Edge

There are many ways to activate an UWP app. The most common way is by double clicking shortcut. Shortcut is a lnk file that is parsed and handled by the shell program.

Activating UWP apps is different from starting traditional Win32 applications. Instead of directly creating a new process, **explorer.exe** sends an activation request to **sihost.exe**.

2. sihost.exe

“sihost” stands for “Shell Infrastructure Host” and runs with medium integrity level. **sihost.exe** is a part of Windows shell environment. The activation request from **explorer.exe** is processed by `activationmanager!Execution::ActivationManagerShim::ActivateApplicationForContractByAppId`

The activation request passed into the function contains:

```
AppId: Microsoft.MicrosoftEdge_8wekyb3d8bbwe!MicrosoftEdge
Contract: Windows.Launch
```

Just like out-of-process COM activation, the activation requests will be sent to RPCSS service. And there are many other functions to handle different activation requests:

```
activationmanager!Execution::ActivationManagerShim::ActivateApplicationFor*
```

3. RPCSS

RPCSS service is the Service Control Manager (SCM) for COM, DCOM, and the new WinRT technology. It handles object activations requests and object exporter resolutions. **RPCSS** service is divided into several processes on newer OS's. It is an important part of RPC, COM, DCOM, and WinRT. For example, when we want to launch an out-of-process COM, the client will communicate with **RPCSS** to request an activation. It will also help connect client and server.

UWP app is based on WinRT, and WinRT is based on COM. The implementation of WinRT is similar with COM. The real activation of UWP also happens in the **RPCSS** service.

a. Get activation information

After **RPCSS** gets the application name, it would find the application activation information.

The needed information is stored in this directory:

```
%ProgramData%\Microsoft\Windows\AppRepository\Packages
```

The system gets the directory through function

```
kernelbase!GetSystemMetaDataPathForPackage
```

RPCSS then gets full name of the package. For Edge, the package full name is

```
Microsoft.MicrosoftEdge_42.17074.1000.0_neutral__8wekyb3d8bbwe
```

There is a **ActivationStore.dat** in the directory. This file is a hive file that could be loaded by **NtLoadKeyEx**. It will be loaded dynamically at runtime to get the activation information. For example, the target executable path for activation. The registry hierarchy looks like this:

```
\REGISTRY\A\{XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX}
```

```
\ActivatableClassId
```

Name	Type	Data
(Default)	REG_SZ	(value not set)
ActivationType	REG_DWORD	0x00000001 (1)
Server	REG_SZ	MicrosoftEdge.AppXdnhjhccw3zf0j06tkg3jtqr00qdm0khc.mca

```
\REGISTRY\A\{XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX}
```

```
\Server
```

Name	Type	Data
(Default)	REG_SZ	(value not set)
ActivatableCla...	REG_MULTI...	MicrosoftEdge.MicrosoftEdge.AppXr0a78g9862rkpb1ydx2mbk2s3x6120x0.mca MicrosoftEdge...
AppUserModelId	REG_SZ	Microsoft.MicrosoftEdge_8wekyb3d8bbwe!MicrosoftEdge
ExePath	REG_EXPAN...	C:\Windows\SystemApps\Microsoft.MicrosoftEdge_8wekyb3d8bbwe!MicrosoftEdge.exe
IdentityType	REG_DWORD	0x00000002 (2)
Instancing	REG_DWORD	0x00000000 (0)
Permissions	REG_BINARY	01 00 14 80 ac 00 00 00 b8 00 00 00 14 00 00 00 30 00 00 00 02 00 1c 00 01 00 00 00 11 00 14 00 04

As shown in above screenshot, for **MicrosoftEdge**, the **ActivationType** is 1, which means out-of-process activation. Executable path of the server is also available in the hive by querying server's full name. This is similar with local server COM activation. All other activation information is under the **Server** key.

b. Creating MicrosoftEdge.exe

The program path for **MicrosoftEdge** is:

"C:\Windows\SystemApps\Microsoft.MicrosoftEdge_8wekyb3d8bbwe\MicrosoftEdge.exe"

But there are some differences with the common process creation. RPCSS will create an AppContainer process.

How to create AppContainer process?

- 1) Get Package SID from Package Family Name

DeriveAppContainerSidFromAppContainerName

- 2) Get Capability SIDs

Capability SIDs of an UWP app is stored in registry.

Capability SIDs comes from **AppXManifest.xml**, current package capability, and an additional **cellularData** capability.

- 3) Create LowBox Token

nt!NtCreateLowBoxToken

- 4) Create **MicrosoftEdge.exe**

Finally, a manager process named **MicrosoftEdge.exe** is started by RPCSS service.

2.2. Content Process Startup

Content process or renderer is a type of child AppContainer process of Edge manager process.

1. MicrosoftEdge.exe

Edge manager process is responsible for creating child AppContainer process.

MicrosoftEdge.exe is an AppContainer process that has limited privileges. To create child processes, it calls Runtime Broker through COM.

2. Runtime Broker

Runtime Broker creates a LowBox token from child AppContainer package SID. The SID is calculated from the token of **MicrosoftEdge.exe** and a restricted name via **DeriveRestrictedAppContainerSidFromAppContainerSidAndRestrictedName**.

MicrosoftEdge.exe package SID is:

S-1-15-2-3624051433-2125758914-1423191267-1740899205-1073925389-3782572162-737981194

The value is the SHA-256 of "Microsoft.MicrosoftEdge_8wekyb3d8bbwe"

SHA-256 of "001" is:

S-1-15-2-1912002900-2594761559-4142726862-4256926629-1688279915-2739229046-3928706915

The child AppContainer SID combines two SID values. The SID of "Microsoft.MicrosoftEdge_8wekyb3d8bbwe/001" is:

S-1-15-2-3624051433-2125758914-1423191267-1740899205-1073925389-3782572162-737981194-4256926629-1688279915-2739229046-3928706915

Next, where to get capability SIDs of child AppContainer process?

These capability SIDs are hardcoded in Edge, it's in `edgeIso!GetRACEenumerationFlags`.

Runtime Broker get the capabilities from **MicrosoftEdge.exe** through COM callback.

The capabilities are as below:

<code>internetClient</code>
<code>sharedUserCertificates</code>
<code>location</code>
<code>microphone</code>
<code>webcam</code>
<code>registryRead</code>
<code>lpacWebPlatform</code>
<code>lpacCom</code>
<code>lpacAppExperience</code>
<code>lpacCryptoServices</code>
<code>lpacIdentityServices</code>
<code>lpacInstrumentation</code>
<code>lpacEnterprisePolicyChangeNotifications</code>
<code>lpacMedia</code>
<code>lpacPnPNotifications</code>
<code>lpacServicesManagement</code>
<code>lpacSessionManagement</code>
<code>lpacPrinting</code>
<code>lpacPayments</code>
<code>lpacClipboard</code>
<code>childWebContent</code>

For restricted name higher than 071 there are two more capabilities:

`privateNetworkClientServer`

`enterpriseAuthentication`

With package SID and capability SIDs, a LoxBox token could be created with `nt!NtCreateLoxboxToken`.

The Runtime Broker would pass the activation requests and the LoxBox token to `sihost.exe`.

3. `sihost.exe`

The process gets the token from Runtime Broker and registers RAC activation token with RPCSS through `combase!CoRegisterRacActivationToken`.

After that, it sends an activation request to RPCSS service.

4. RPCSS

RPCSS then performs the following operations:

- 1) Lookup RAC token
 Get RAC token that registered previously
- 2) Get WinRT runtime class
 Get the activation information
- 3) Create content process

Now a content process is started.

3. Inter-Process Communication

IPC components are most vulnerable to privilege escalation bugs.

In this chapter, we will talk about inter-process communication mechanisms used by *Microsoft Edge*. There are three types of IPC mechanisms: RPC, COM and LCIE IPC. These IPCs provide a large attack surfaces for sandbox.

3.1. RPC

RPC (Remote Procedure Call) is an inter-process communication mechanism. It allows client and server to communicate over several protocol sequences, such as ALPC ports, named pipes, Winsock etc.

The server's interface (identified by an UUID) will be bound on an endpoint with specified protocol. Then a client can access the interface with the identity.

RPC server can use security descriptor to control access permission of an endpoint. It can also use interface security callback function to check permissions of a client.

By default, there are many RPC servers in Windows. Some of them are accessible in a sandbox.

In *Microsoft Edge*, the JavaScript JIT engine runs in a separate renderer. Other processes use RPC to communicate with JIT renderer. For example, internet renderer sends JavaScript code to JIT renderer via RPC for compilation. JIT renderer will reply with generated machine code.

JIT renderer registers an RPC server:

```
status = RpcServerRegisterIf3(  
    ServerIChakraJIT_v0_0_s_ifspec,  
    NULL,  
    NULL,  
    RPC_IF_AUTOLISTEN,  
    RPC_C_LISTEN_MAX_CALLS_DEFAULT,  
    (ULONG)-1,  
    NULL,  
    securityDescriptor);
```

You can find the complete source code in ChakraCore repository:
`ChakraCore/lib/JITServer`

3.2. COM

3.2.1. Overview

COM comes from OLE technology. After years of evolution, COM has become the foundation of *Windows*. COM is widely used in *Windows* shell components and service components, because it provides strong reusability, scalability, and isolation. For example, *Microsoft Office* uses COM/OLE to implement compound document. *Windows* also uses COM to implement *ActiveX* control. Many local and remote services use COM/DCOM to communicate with each other.

COM defines a binary interoperability standard for creating reusable software libraries that interact at runtime. A reusable interface implementation is called a component, a component object, or a COM class object (Identified by a CLSID). A component implements one or more COM interfaces. And a COM interface (Identified by an IID) is a collection of member functions. All communication among COM components occurs through interfaces, and all services offered by a component are exposed through its interface. A caller can access only the interface member functions. Internal state is unavailable to a caller unless it is exposed in the interface.^[5]

The Client / Server Model

A COM class implements several COM interfaces. The implementation consists of binaries that run when a caller interacts with an instance of the COM class. COM enables using a class in different applications, including applications written without knowledge of a class. On a *Windows* platform, classes exist either in a dynamic-linked library (DLL) or in another application (EXE).

On client/server's host system, COM maintains a registration database of all the CLSIDs for the COM objects installed on the system. The registration database is a mapping between each CLSID and the location of the DLL or EXE that houses the corresponding class. COM queries this database whenever a caller wants to create an instance of a COM class. The caller needs to know only the CLSID to request a new instance of the class.

COM uses client/server model for object interaction. The client is the caller that requests a COM object from the system, and the server is the module that houses COM objects that provides services to clients.^[5]

In-Process COM Server and Out-of-Process COM Server

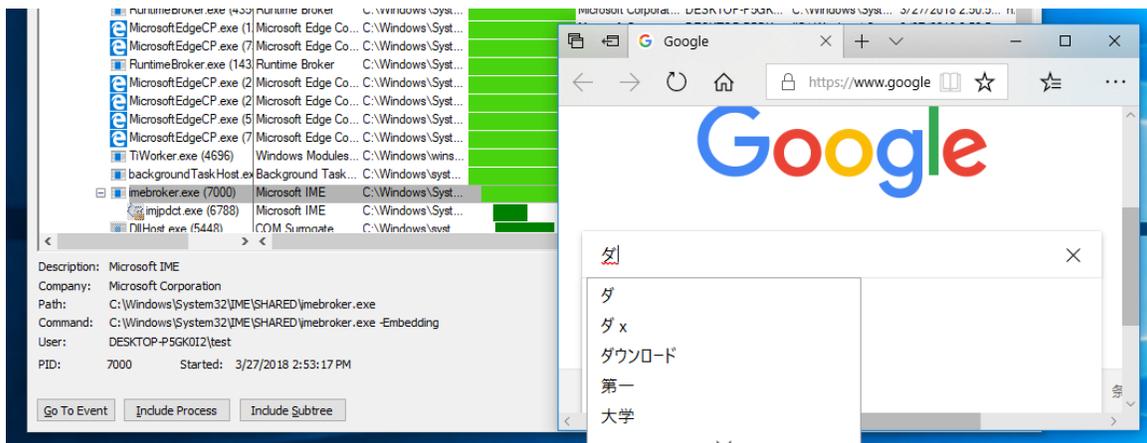
There are two types of COM servers, in-process COM server and out-of-process COM server. In-process COM servers are implemented in a dynamic linked library (DLL). Out-of-process COM

servers are implemented in an executable file (EXE). Some out-of-process COM servers can reside on a remote computer or have different permissions than their client. This type of COM server is also called DCOM server.

We can call `CoCreateInstance` with `CLSCTX_INPROC_SERVER` to create an in-process COM server and call `CoCreateInstance` with `CLSCTX_LOCAL_SERVER` to create an out-of-process COM server. The client communicates with out-of-process COM server through LPC/RPC.

From the security perspective, we are more interested in out-of-process COM, since they run in separate processes and their permissions may be higher than AppContainer. They opened up a large attack surface for sandboxed processes.

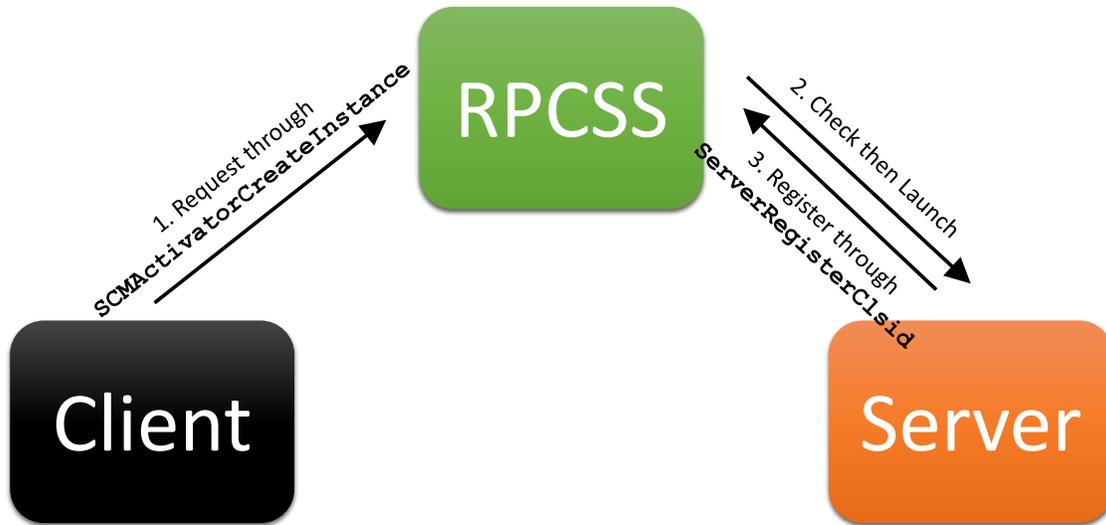
For example, we found *Microsoft Edge* uses a COM server **ImeBroker** to create dictionaries, learn words etc. **ImeBroker** runs at medium integrity level and had a sandbox escape bug in the past.



ImeBroker operating on dictionaries

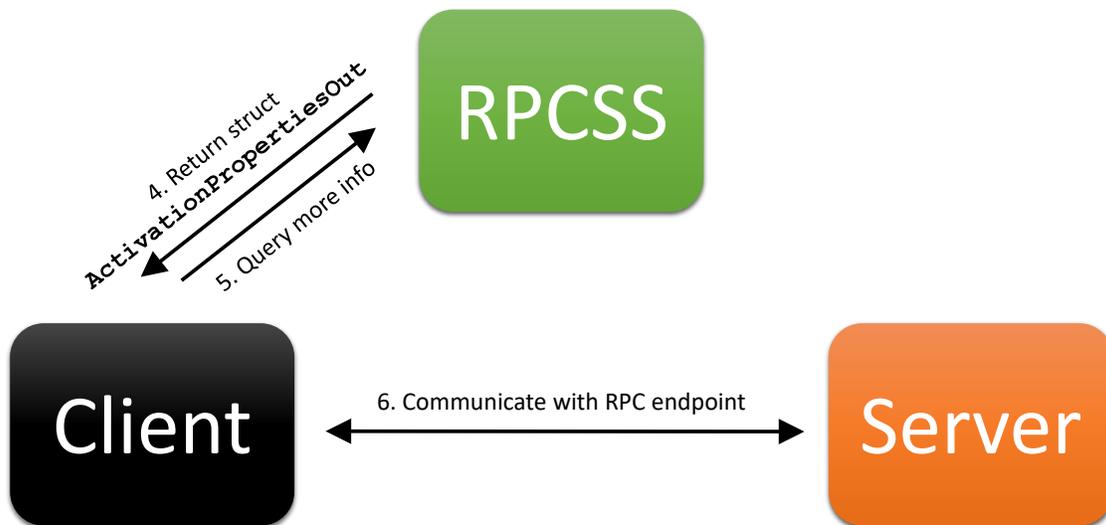
Out-of-Process COM Activation

To launch or activate an out-of-process COM server, the COM runtime in the client will send a launch or activation request to system activator that runs in RPCSS service, by calling `RPCSS!SCMAActivatorCreateInstance` through RPC.



Out-of-process COM Activation

The launch and activation permission is checked in `RPCSS!CClassData::LaunchOrActivationAllowed`, which is a child function of `RPCSS!SCMActivatorCreateInstance`. If client has launch/activation permission, RPCSS will launch/activate associated out-of-process COM server with appropriate user account. When an out-of-process COM server starts, it will register its class and interface information to RPCSS service through `RPCSS!ServerRegisterClsid`.



Out-of-process COM Activation

Then RPCSS service will return an `ActivationPropertiesOut` structure back to the client. The structure is an `OBJREF` that contains information about out-of-process COM server. The client can query more information about the server, such as RPC endpoint. Then client could establish an RPC connection to the server. James Forshaw has detailed this process^[6].

3.2.2. COM Security

COM security heavily depends on Windows and RPC security mechanisms. COM security relies on authentication (the process of verifying a caller's identity) and authorization (the process of determining whether a caller is authorized to do what it is asking to do).

There are two main types of security in COM: activation security and call security. Activation security determines whether a client can launch a server at all. After a server is launched, it can use call security to control client access to server objects.^[7]

Security Settings

System-wide

System-wide security settings control the default launch and access permission and call-level security capabilities for COM servers that do not have process-wide security settings or call **CoInitializeSecurity** explicitly. These settings associated with the key `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Ole`.

The following are security related settings:

Key	Description	Remarks
DefaultLaunchPermission	Define default launch ACL for the computer	
DefaultAccessPermission	Define default access permission list for the computer	By default, this value has no entries in it. Only the server principal and system are allowed to call the server.
MachineLaunchRestriction	Principals not given permissions here cannot obtain them even if the permissions are granted by the DefaultLaunchPermission registry value or by the CoInitializeSecurity function.	
MachineAccessRestriction	Principals not given permissions here cannot obtain them even if the permissions are granted by the DefaultAccessPermission registry value or by the CoInitializeSecurity function.	
LegacyAuthenticationLevel	Sets the default authentication level for applications that do not call CoInitializeSecurity	the default authentication level established by the system is RPC_C_AUTHN_CONNECT
LegacyImpersonationLevel	Sets the default level of impersonation for applications that do not call CoInitializeSecurity	the default impersonation level established by the system is RPC_C_IMP_LEVEL_IDENTIFY

Process-wide

COM server can use process-wide security settings to supply their own security values. The process-wide security settings in the registry associated with the key `HKEY_LOCAL_MACHINE\SOFTWARE\Classes\AppID\{AppID_GUID}`. The registry value **LaunchPermission**, **AccessPermission**, **AuthenticationLevel** etc. can be used for corresponding security settings.

It is worth noting that COM Server can call **CoInitializeSecurity** explicitly to override default permission. Otherwise, the COM runtime will implicitly call **CoInitializeSecurity** with the process-wide security settings or the computer-wide security settings in the registry.

Besides, if the **AuthenticationLevel** is none, the **AccessPermission** and **DefaultAccessPermission** values are ignored for that application. And if the **AuthenticationLevel** is not present and the **LegacyAuthenticationLevel** is none, the **AccessPermission** and **DefaultAccessPermission** values are ignored for that application.

Security Check

As mentioned earlier, the RPCSS service is responsible for launching/activating the COM server. RPCSS service will get the client's identity to check if it has access to launch/activate the server. This is achieved by calling `RPCSS!CClassData::LaunchOrActivationAllowed`.

Launch means create a new instance of the COM server. Activation means create a new object on an existing server. Usually, both launch and activation permission use **LaunchPermission**.

For the access permission, it is checked by COM runtime at server side. RPCSS will call the interface security callback function (`combase!ORPCInterfaceSecCallback`) registered by the COM server at startup during access check. The callback function in turn will call `combase!CheckAccess` to check client's access permission.

3.3. LCIE IPC

3.3.1. Overview

LCIE stands for Loosely-Coupled IE^[8]. It is a collection of internal architecture changes to *Internet Explorer* that improve reliability, performance, and scalability of the browser. LCIE isolates tabs (renderers) from the UI frame (manager).

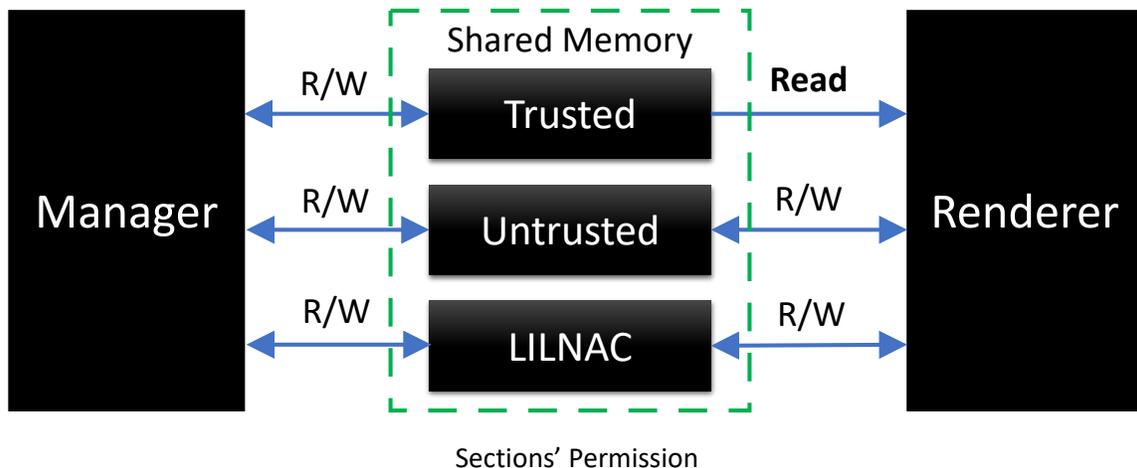
LCIE IPC is an IPC designed for LCIE based on shared memory, also known as Shared Memory IPC^[9]. It is used for sharing data and exchanging inter-process messages.

Window message is an important inter-process communication mechanism. In newer Windows OS's, window messaging is restricted across security boundaries.

LCIE IPC is a simulation of window messaging, based on shared memory. LCIE uses sections to share memory. During the initialization of **MicrosoftEdge.exe**, it creates a named file mapping through **CreateFileMapping**. There are three types of sections: Trusted, LILNAC, Untrusted:

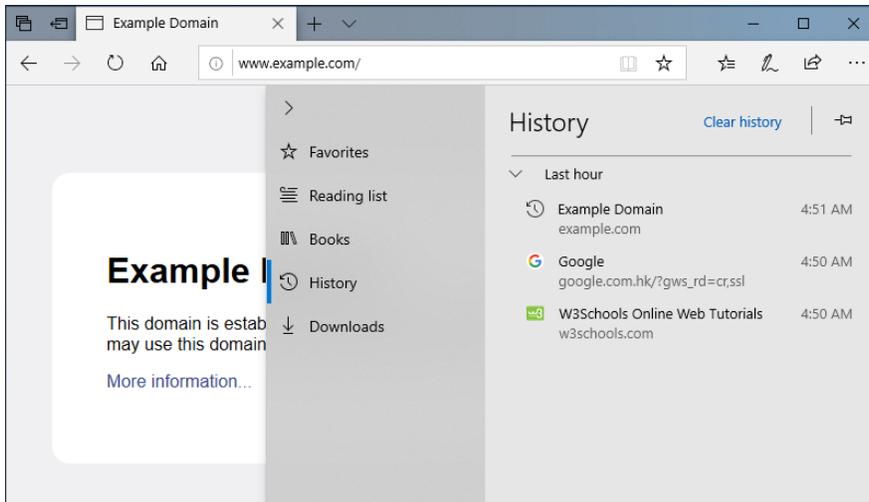
```
IsoSpaceV2_Scope_Trusted  
IsoSpaceV2_Scope_LILNAC  
IsoSpaceV2_Scope_Untrusted
```

The renderer process and browser broker would open the sections and write to or read from the shared memory through **OpenFileMapping**.



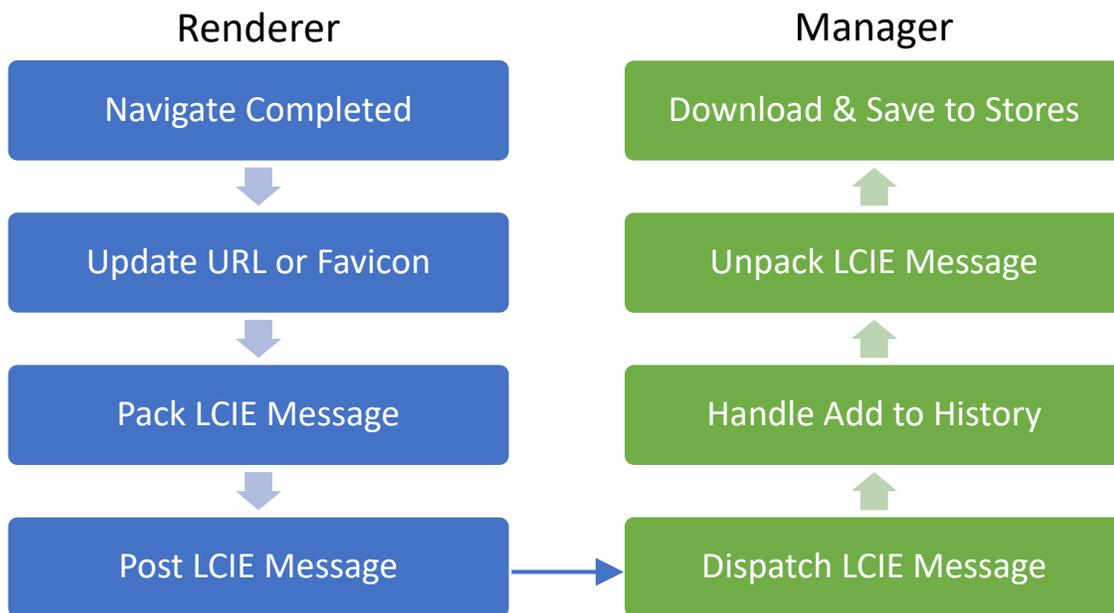
LCIE IPC uses event to send message across process boundaries. It uses **SetEvent** to notify target process to process new messages. The target process would go over all the sections to process the messages that transferred to it.

A trivial use case would be renderer adding URLs and favicons to manager's history through LCIE IPC.



Manager's history

The renderer will send URL of website to manager when navigation is completed. It will pack website URL as a LCIE message by calling **IsoAllocMessageBuffer** and **LCIEPackString**, then send the message to manager by calling **LCIEPostMessage** with message ID 0xC59.



Add URL/Favicon to History

In manager, **LCIEAuthorityManagerWinProc** will be called when a message is received. It uses message ID to dispatch and handle messages. Manager will get website URL by calling **IsoGetMessageBufferAddress**, **IsoGetVariableArtifactSize** and **LCIEUnpackString**, then download its favicon and add them to history.

3.3.2. LCIE IPC Message Security

There are many types of LCIE Message:

```
ISO_MSG_WM_MSG
ISO_MSG_WM_MSG_UNTRUSTED
ISO_MSG_WM_MSG_AVAILABLE
ISO_MSG_WM_MSG_AVAILABLE_UNTRUSTED
ISO_MSG_WM_MSG_HUNG_CHECK
```

Message sent by manager process should be trusted. Messages sent by renderer should be carefully checked.

Trusted messages will be written into **IsoSpaceV2_Scope_Trusted**. As shown before, renderer process has no permission to write messages into the trusted scope.

In manager and broker, some functions would check the LCIE message type. If the message is untrusted, the process will not handle it. This will prevent untrusted process from calling sensitive functions.

4. Vulnerabilities

4.1. Sandbox Escape Chain 1

4.1.1. Browser Broker Bug

Browser Broker

Browser Broker is an important part of *Microsoft Edge* app. It is a local server COM object which can be launched through out-of-process COM activation. Here is some information about this COM object:

```
CLSID: {0002DF02-0000-0000-C000-000000000046}
```

```
AppID: {DD9C53BC-8441-4B94-BD0E-36E6E02A6D61}
```

Name:	BrowserBroker Class
CLSID:	0002DF02-0000-0000-C000-000000000046
Server Type:	LocalServer32
Server:	C:\Windows\system32\browser_broker.exe
CmdLine:	C:\Windows\system32\browser_broker.exe
TreatAs:	N/A
Threading Model:	Both

When activating Edge, Browser Broker will be launched immediately with process name **browser_broker.exe**. Browser Broker is critical for Edge to function, so **MicrosoftEdge.exe** will always check if it is still alive and will launch a new instance if it is dead.

The broker process runs at medium integrity level, with very few security mitigations enabled. Browser Broker could communicate with *Microsoft Edge* processes running in AppContainer. It helps AppContainer processes perform privileged operations, such as **LaunchIE**, **OpenFolder** and more.

1. Manager AC

When **MicrosoftEdge.exe** starts, the **iertutil!AttachBrowserElevationBroker** will launch a Browser Broker server and get its interface.

Browser Broker could be launched with this method:

```
CoCreateInstance(  
    CLSID_BrowserBroker,  
    nullptr,  
    CLSCTX_LOCAL_SERVER,  
    IID_PPV_ARGS(&browser_broker)  
)
```

When Edge manager process creates a child process, the **BrowserBroker** interface is transferred to that child process through LCIE inter-process communication. The transfer process is as follows:

- a) Allocate shared memory

```
CIsoScope::AllocSharedMemoryPerFlags
```

- b) Get the **BrowserBroker** interface

```
iertutil!GetBrowserBrokerInterface
```

- c) Marshal interface

```
edgeIso!LCIEMarshalInterfaceIntoBufferOtherProcess  
combase!CoMarshalInterface
```

Now there will be an **OBJREF** of **BrowserBroker** interface stored in a block of shared memory that can be read by that child process.

2. OBJREF

OBJREF is the name of the structure of marshalled interface in COM. A COM interface could be marshalled into **OBJREF** and passed to another context, where it is unmarshalled back to a COM interface.

This is an example of **BrowserBroker** **OBJREF**:

```
4d 45 4f 57 01 00 00 00-f6 e7 79 71 e0 4f b3 48
```

```

8b 6a bb 41 3b f6 ea 0d-00 04 00 00 01 00 00 00
f0 35 ad 74 ad 0a d3 2d-c4 98 33 8d 14 f1 c3 27
02 a4 00 00 e0 24 00 00-98 cf 60 cf 7f a2 21 0a

```

This can be parsed as:

574f454d	Signature	'MEOW'
00000001	Flag	OBJREF_STANDARD
7179e7f6 48b34fe0 41bb6a8b 0deaf63b	IID	IBrowserBrokerFactory
00000400	Flags	
00000001	cPublicRefs	
74ad35f0 2dd30aad	OXID	0x2dd30aad`74ad35f0
8d3398c4 27c3f114	OID	0x27c3f114`8d3398c4
0000a402 000024e0 cf60cf98 0a21a27f	IPID	

3. Child Process

When child process starts, how can it get **BrowserBroker** interface?

a) Get shared memory

```
edgeIso!IsoGetShareMemoryAddress
```

b) Unmarshal Interface

```
edgeIso!LCIEUnmarshalInterfaceFromBuffer
```

```
combase!CoUnmarshalInterface
```

After the child process gets a **BrowserBroker** interface, it can call interface methods. But things are not that easy. Most of the methods have access check to limit operations a child process can perform.

Access check

As we have known methods that browser broker provided are useful for privilege escalation. If we can bypass the access check, we may get higher privileges.

How does browser broker check permissions?

Most of the methods start with a call to function **BrokerAuthenticateAttachedCallerGetPIC**.

Because of the check, only a few non-privileged methods can be called in Internet Child AC.

Behind this check function is `edgeIso!IsoGetTokenIsoIntegrity`. This function returns an integer representation of child AppContainer's restricted name.

From symbols we could know the return value is an enum **IsoIntegrity**, we got some interesting values from `urlmon.dll`.

```
IsoIntegrity_PIC_MRAC = 1
IsoIntegrity_PIC_Dynamic_Low = 7
IsoIntegrity_PIC_Dynamic_High = 119
IsoIntegrity_PIC_Intranet_AC = 121
IsoIntegrity_PIC_Trusted_AC = 122
```

Dive into the access check function:

```
BrokerAuthenticateAttachedCallerGetPIC(arg_0, &arg_1);
```

arg_0 has different values in different methods. For example:

```
CBrowserBrokerInstance::LaunchIE = 1
CBrowserBrokerInstance::CreateBrokerObject = 2
CBrowserBrokerInstance::AddCredential = 5
```

The differences between 1 and 2 and 5:

```
1: Only trusted AC can access
2: Everyone can access
5: All children AC except 002 can access
```

Therefore, in internet child AC, we cannot call the **LaunchIE** method. In fact, only trusted AC can access all privileged methods.

Trusted AC

What is the trusted AC? Which caller process could be in a trusted AC? It is quite easy to guess that Microsoft Edge Manager Process **MicrosoftEdge.exe** is in trusted AC. But how did Microsoft implement it?

After a deep research, we find an AC is trusted when the caller's Package SID equals a SID passed in through **RequestBroker** method when **MicrosoftEdge.exe** launches the browser broker. It is always the Package SID of **MicrosoftEdge.exe**.

RequestBroker

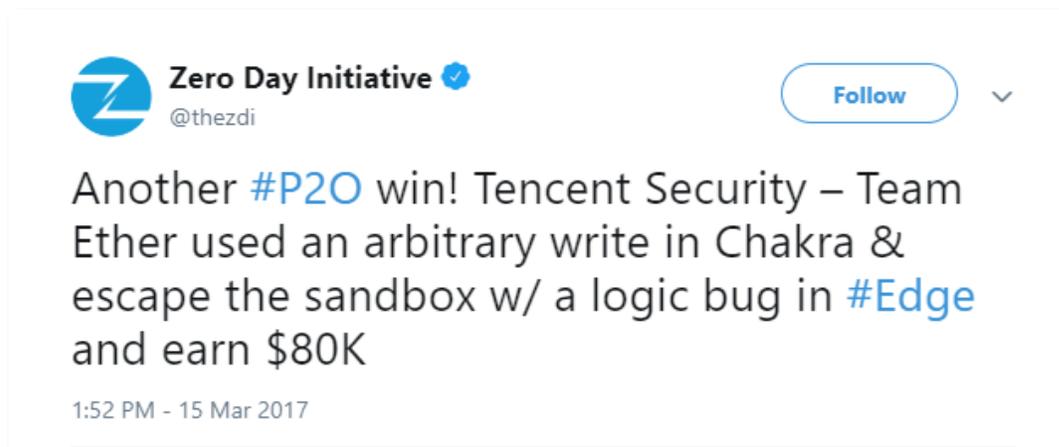
RequestBroker is a browser broker method used to register context information when the browser broker interface is being created.

RequestBroker is designed to be called only once.

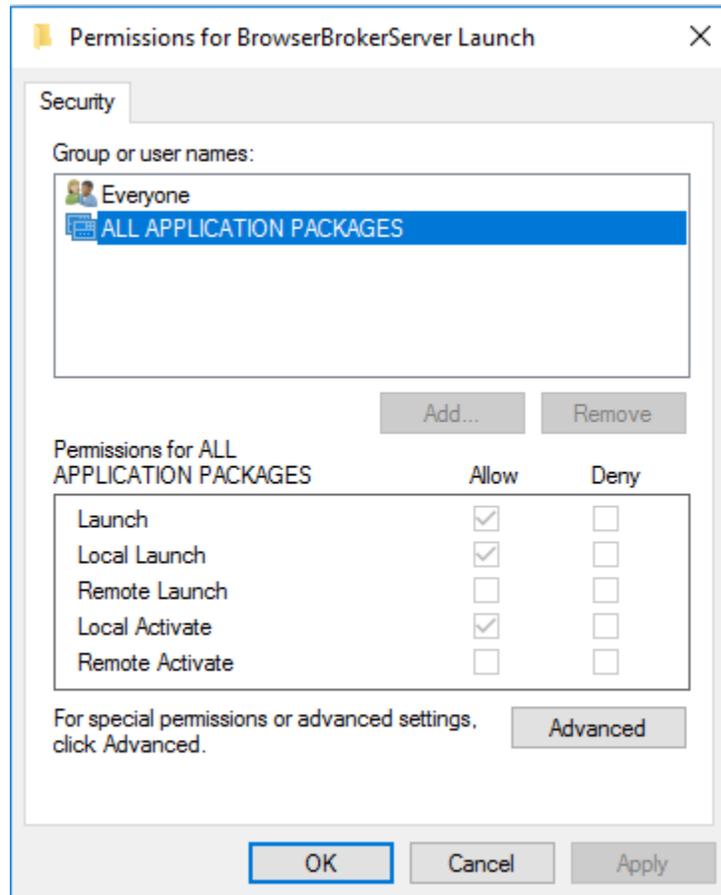
```
if (!dwProcessId || InterlockedCompareExchange(variable,
dwProcessId, 0))
    return 0x80070005;
```

CVE-2017-0233

We have used this vulnerability to win the Pwn2Own 2017 Edge Category. This is the only logical vulnerability used to bypass Edge sandbox during the contest.



Let us inspect browser broker Launch Permission and Access Permission on Windows 10 14393.



We could know that **ALL APPLICATION PACKAGES** have Launch and Local Activate permissions.

ALL APPLICATION PACKAGES

ALL APPLICATION PACKAGES is just a SID, S-1-15-2-1. It represents all applications running in an app package context. According to Microsoft, "To allow all AppContainers to access a resource, add the **ALL APPLICATION PACKAGES** SID to the ACL for that resource. This acts like a wildcard." [10]

It seems that all AppContainers can launch the browser broker.

Now the sandbox bypass:

1. Launch a new browser broker in content process
2. **RequestBroker** to register context information. It is the first call to **RequestBroker**, so we can register Content Process Package SID with browser broker.
3. Now content process AC is trusted
4. Call arbitrary broker methods

There are many dangerous methods that can be called in a trusted AC. For example, we could launch *Internet Explorer* with controlled arguments through **LaunchIE** method.

But we found something more dangerous.

CBrowserBrokerInstance::WriteClassesOfCategory

This method starts with

```
BrokerAuthenticateAttachedCallerGetPIC(1, &v8);
```

The access check type is 1, which means it can only be called in a trusted AC.

After the check, this function calls

LoadTheSinglePossibleSPFrameDllForThisProcess

Inside it is:

```
wscpy_s(&Dst, 0x104ui64, AppDir);  
wscat_s(&Dst, 0x104ui64, L"\\eModel.dll");  
v4 = LoadLibraryExW(&Dst, 0i64, 0x1010u);
```

It will load an **eModel.dll** from Edge application directory. Interestingly, initialization of application directory global variable is also done through **RequestBroker** during initialization.

We can now load a custom **eModel.dll** into **browser_broker.exe**.

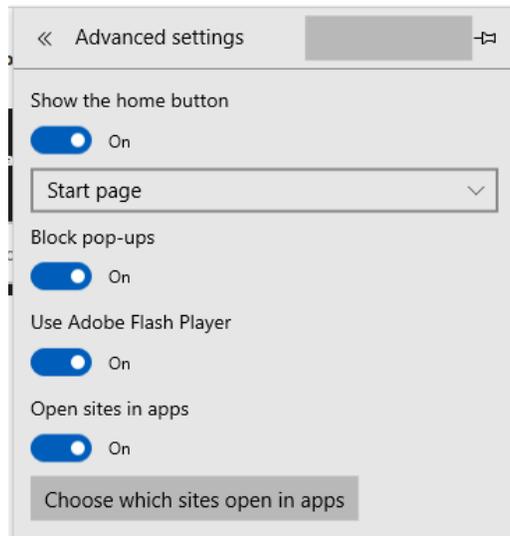
4.2. Sandbox Escape Chain 2

4.2.1. Flash Broker Bug

Flash

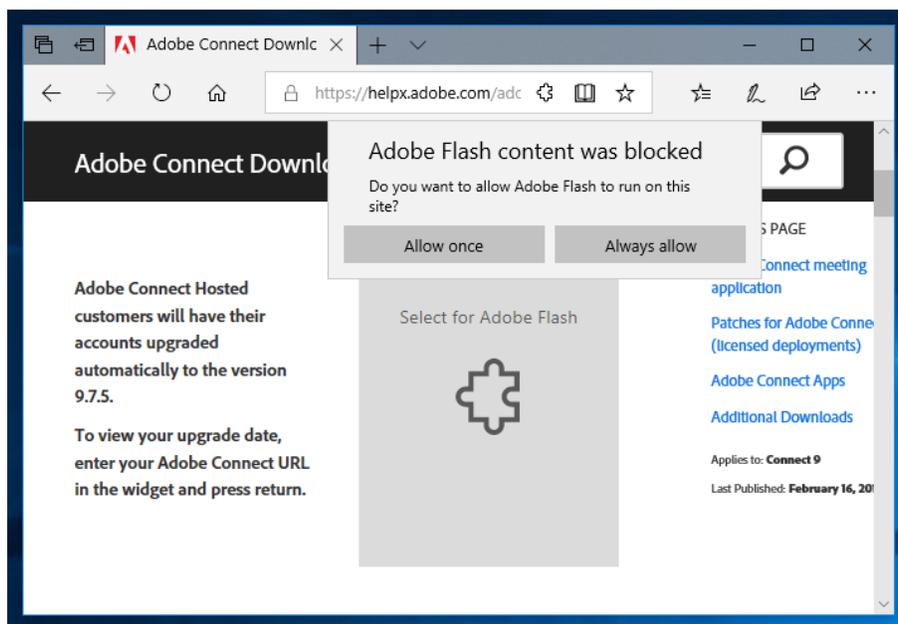
Adobe Flash used to lead the way on the web for rich content, gaming, animations, and media of all kinds, and inspired many of the current web standards powering HTML5^[11]. Last year, *Adobe* announced that *Adobe Flash* will no longer be supported after 2020. However, there are still many websites and applications using *Flash* technology.

Adobe Flash Player is pre-installed starting from *Windows 8*. As shown below, *Adobe Flash Player* is integrated into *Microsoft Edge* and enabled by default.



Adobe Flash Player enabled by default

To phase out *Adobe Flash Player* from browsers, *Microsoft*, *Google*, *Mozilla*, and *Apple* all have begun limiting the auto-run of *Flash*. They all implemented a Click-to-Run for *Flash* and planned to disable *Flash* by default in the future.



Click-to-Run for Flash

Microsoft Edge runs *Adobe Flash Player* in a special renderer called **BCHost**. As we mentioned in chapter 1.3, **BCHost** renderer also runs in a child *AppContainer* sandbox. The Internet renderer cannot access *Adobe Flash Player*. This further hardens the *Microsoft Edge* sandbox, as attackers cannot exploit *Adobe Flash Player* vulnerabilities in Internet renderer.

Flash Broker

Same as *Microsoft Edge*, *Adobe Flash Player* also needs its own broker to perform privileged operations. Flash Broker is also the manager of *Flash*-based add-ins, such as *Adobe Connect Add-in*, *Microsoft Outlook Add-in* and more. As a result, *Flash* broker becomes another attack surface of *Microsoft Edge* sandbox.

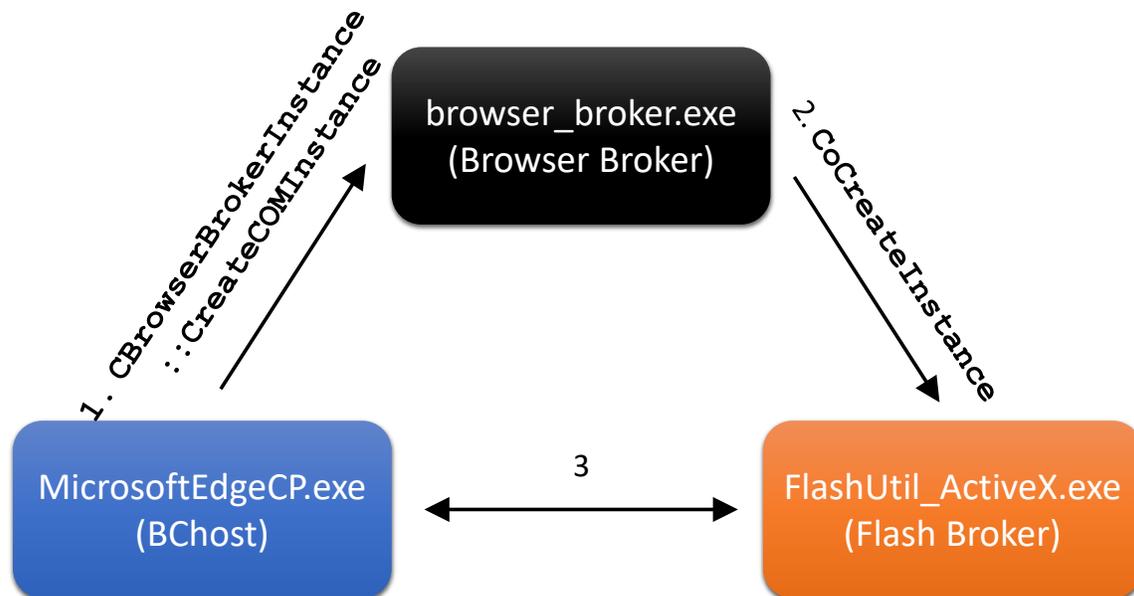
Flash Broker Permissions

In fact, Microsoft have considered isolating Flash broker when designing *Microsoft Edge*. That is why it runs *Adobe Flash Player* in a special renderer, and access to Flash broker is also strictly restricted.

No renderers have launch and activate permission to Flash broker. Instead, launch requests are relay to the browser broker. The Browser broker is responsible for launching the Flash broker and passing its interface back to renderer. Only **BCHost** and local renderers have access permission to Flash broker interface.

Flash Broker Activation

Following diagram shows activation process of the Flash broker. **CoCreateInstance** was shimmed and relayed to Browser broker's **CBrowserBrokerInstance::CreateCOMInstance**. Browser broker launches the Flash broker and passes the interface back to the renderer.



Activation Process

Flash Broker Features

Currently, Flash broker exports 6 interfaces with 124 methods. Its interfaces and functions are as follows.

IFlashBroker1: file, LCD accessor, register profile, add-in operations

IFlashBroker2: register profile operations

IFlashBroker3: popup and GDI device operations

IFlashBroker4: utility functionalities

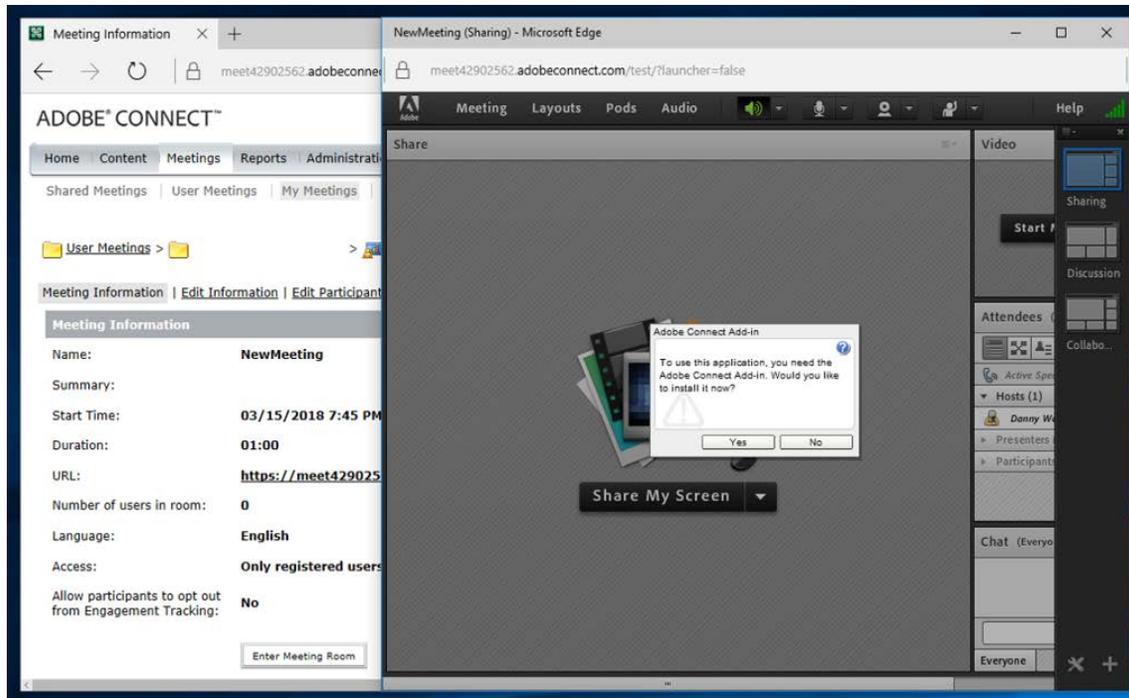
IFlashBroker5: utility functionalities

IFlashBroker6: file operations, add-in operations

Flash broker supplies a lot of functions such as popup, file operations, add-in operations etc. The most important thing is that it runs at Medium Integrity Level. In the last few years, there have been multiple vulnerabilities found in Flash broker.

Flash Broker Bug

We found some interesting behavior when investigating these interfaces and functions of the Flash broker. As following screenshot shows, *Flash Player* prompts user to install an Add-in when user enters a meeting room created by *Adobe Connect*. *Adobe Connect* is a software used for presentations, web conferencing, and user desktop sharing.



Adobe Flash Player prompts user to install an add-in

If user clicks the Yes button, Flash broker will download and then launch the add-in with medium integrity level.

Functions with name starts with "BrokerLM" in **IFlashBroker6** are used for downloading and launching add-ins. They are called in this order:

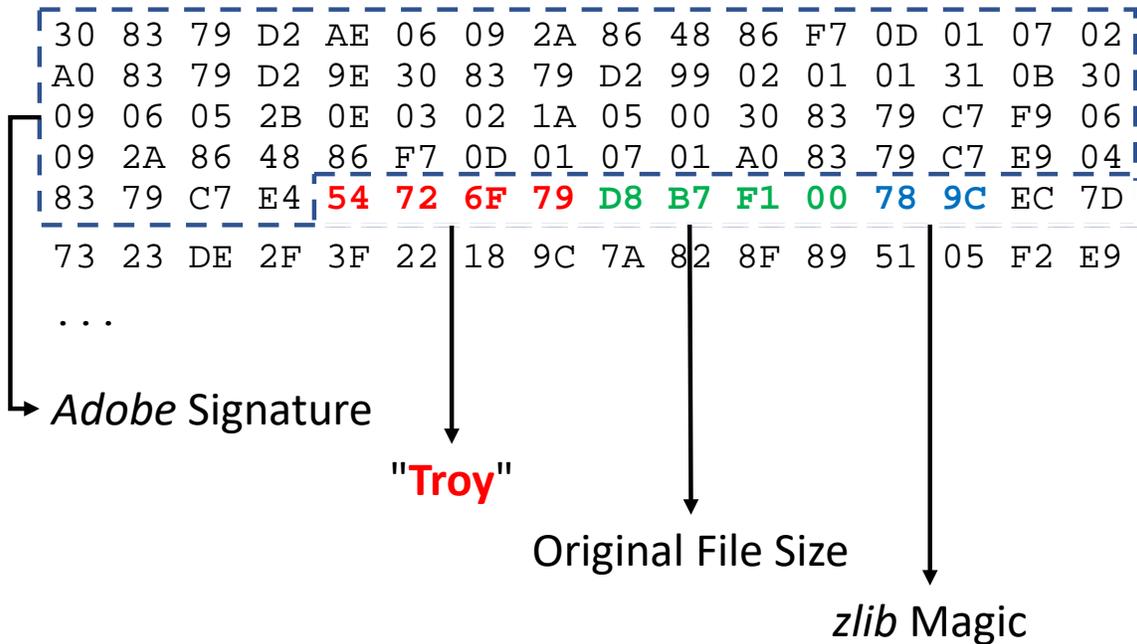
```
IFlashBroker6::BrokerLMOpenDownload(p_url, p_lmCookie)
IFlashBroker6::BrokerLMUpdateDownload(p_lmCookie, p_pos, p_len)
IFlashBroker6::BrokerLMCloseDownload(p_lmCookie, p_applicationName, p_resultCode)
IFlashBroker6::BrokerLMLaunch(p_applicationName, p_applicationParams)
```

For each add-in, two paired files (dot z file & dot s file) should be downloaded and their signatures must be valid.

Dot Z file

First, the file with the extension Z is downloaded. The compressed data block must start with a magic string 'Troy'. This catches our strong interest. Looks like the developer is making a reference to the familiar story about Troy when writing this piece of code.

Before the magic string is a digital signature block. It will be verified with a hardcoded certificate chain. Following the magic string is the uncompressed file size.

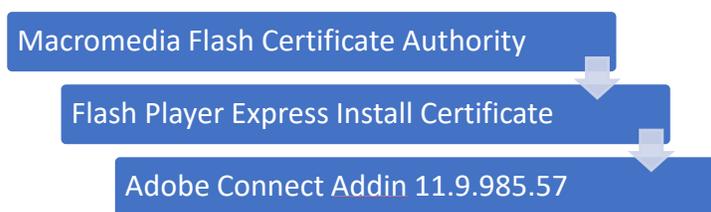


Dot Z file header

Dot S file

Second, the file with the extension S is downloaded. It is also signed by *Adobe*. It holds a SHA-256 digest for the uncompressed add-in and is used for verifying add-in at a later stage. All digital signatures of downloaded files are verified with a hardcoded certificate chain.

The certificate chain of *Adobe Connect* Add-in is as follows:



Flash Add-in Download and Launch

The entire process of downloading and launching an add-in is as follows.

1. Download and verify the dot z file
 - 1) Verify the URL is in "macromedia.com" domain
 - 2) Download file

- 3) Verify the digital signature of the dot z file
 - 4) Decompress the dot z file
2. Download the dot s file
 - 1) Verify the URL is in "macromedia.com" domain
 - 2) Download file
 3. Launch add-in
 - 1) Verify the digital signature of dot s file
 - 2) Verify add-in with the SHA-256 digest contained in dot s file
 - 3) Run add-in with controllable arguments

The add-in can only be run when the signatures of downloaded files are verified to be valid and the SHA-256 of add-in matches the digest. It is worth pointing out that Flash broker can only download files from "macromedia.com".

Good news is we can run add-in with controllable arguments.

CVE-2018-12828

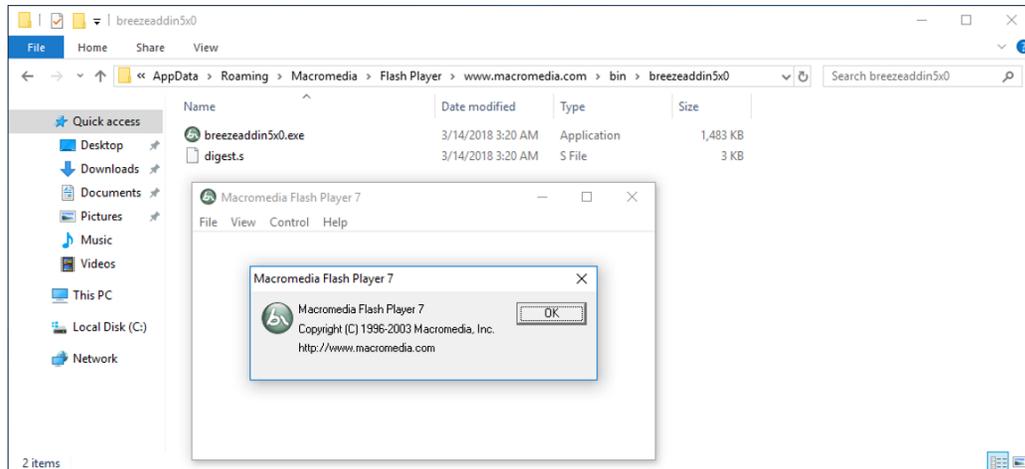
The attack surface of *Microsoft Edge* sandbox is extended because of the controllable arguments even though we can only download files from "macromedia.com".

These add-ins may contain vulnerabilities or become unmaintained. So, what we need is to find as many add-ins as possible. If we can find bugs in these add-ins, we may break the sandbox.

After investigation, we found several add-ins and digest files on the Macromedia web server. Many of them still have valid signatures. Most important thing is that some of them are vulnerable.

All these vulnerable add-ins built with an ancient Adobe Flash Player and can open an SWF file via command line arguments. It turns out that we can use some known vulnerabilities of Flash to escape Edge sandbox.

Macromedia Breeze is a case. *Breeze* is a web communications application created by *Macromedia*. It has a built-in *Adobe Flash Player* released in the year 2003.



Macromedia Breeze Add-in

4.2.2. SOP Bypass

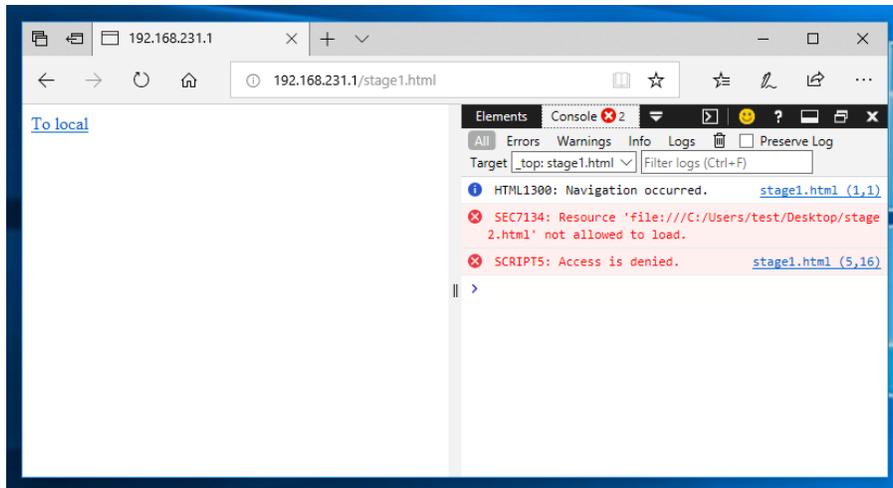
For now, we assume that we already have code execution by exploiting an RCE bug. And we can escape sandbox from **BCHost** renderer or local zone renderer with the Flash broker bug.

To build a complete exploit chain, we have two options. Either find a way to run JavaScript in **BCHost** renderer or cross origin from internet to local.

However, as we mentioned earlier, getting the **BCHost** renderer still needs user confirmation. We cannot control the confirmation button within renderer. That is controlled by the manager. Finding a UXSS bug in the local zone is also not easy. Let us think about what we can do with code execution in the internet zone.

Navigation

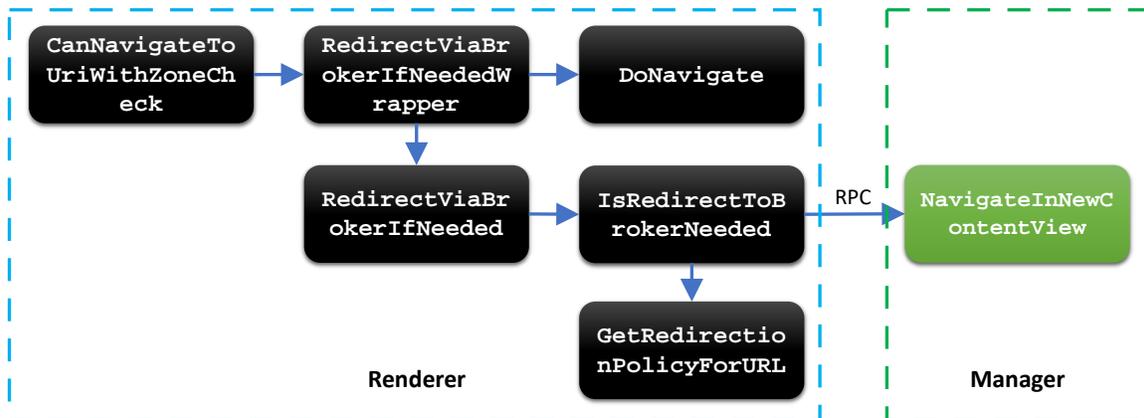
As the following screenshot shows, access is denied when navigating from the internet to local. Same origin policy forbids this kind of behavior to prevent cross site information leakage.



Navigate from Internet to Local Zone Renderer

Let us dig into navigation internals. A navigation will happen when we visit a website through address bar or click a hyperlink on a web page.

During navigation, `edgehtml!CDoc::FollowHyperlink2` will be called. Renderer and manager will do a series of checks for the target URL to determine whether to navigate in current renderer or a new renderer.



Behind a navigation

`FollowHyperlink2` will first call `CanNavigateToUriWithZoneCheck` to check if the target URL satisfies the zone policy. If not, it will deny access. Otherwise, it will call `RedirectViaBrokerIfNeededWrapper` to determine if the target URL needs to be opened in a new renderer or the current renderer.

`RedirectViaBrokerIfNeededWrapper` will check the target URL with redirection policy by calling `GetRedirectionPolicyForURL`. If the check passes, the navigation request will be sent to the manager via RPC.

For example, it will call **GetPICForPrivilegedInternalPage** to check if the target URL points to a legally privileged zone renderer (about:config, res://edgehtml.dll/flags.htm, res://edgehtml.dll/compat.htm etc.). The PIC corresponding to a privileged zone renderer is 4.

GetPICFromZoneForUrl checks and retrieves the PIC of the local zone renderer.

To reduce number of RPC calls, the renderer calls the same function (**GetRedirectionPolicyForURL**) in process.

SOP Bypass

CVE-2018-8358

There are two security issues in the above navigation procedure.

First, the zone check for navigation is completely inside the renderer process. We can navigate to a local page when host URL's protocol is file. It can be bypassed with crafted data or request manager directly.

Second, manager retrieves PIC from zone ID, which in turn is calculated from target URL (`urlmon!IEGetZoneIUri`). However, there is no additional check on file URL, and the retrieved PIC is 121 when target URL is a local HTML file.

By exploiting these issues, we can render a local HTML file dropped by internet renderer in a local zone renderer.

Exploit the SOP Bypass

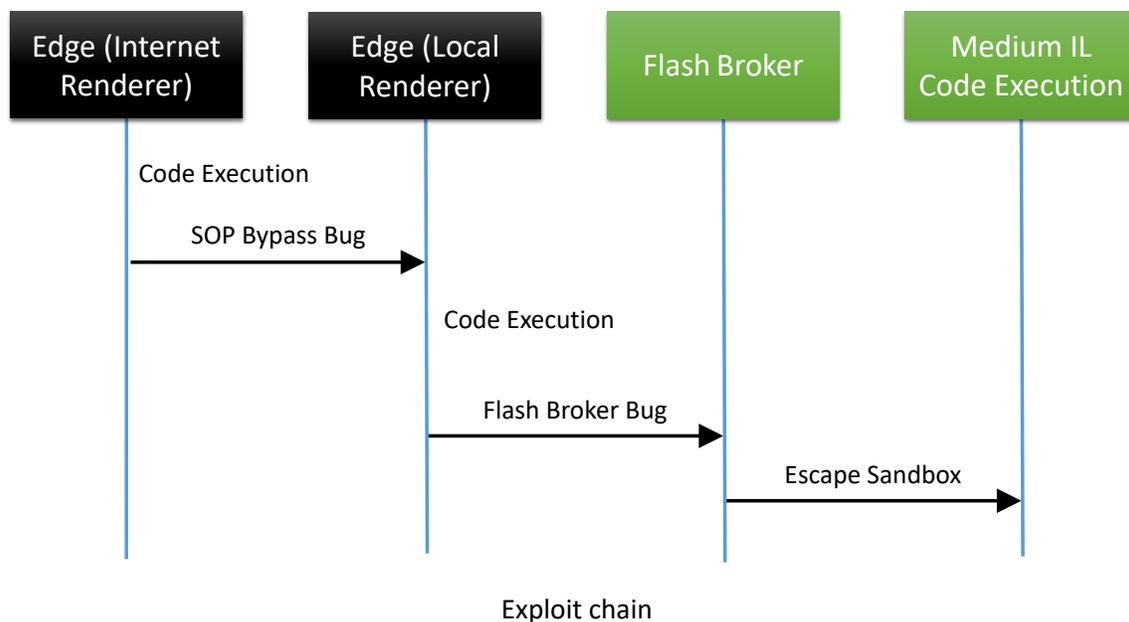
To exploit the SOP bypass, we first write a local HTML file into a temporary folder that can be accessed by local zone renderer.

Then we create an **AnchorElement** with that file as source. We can locate the **HostUrl** via the **AnchorElement** and modify it to begin with "file:/// " protocol in internet renderer.

Finally, we trigger a navigation via an **onclick** event. It bypasses the zone check and navigate to the local HTML file in a local zone renderer.

4.2.3. Exploit Chain

Let us put it all together to build a complete exploit chain. The following diagram shows the entire sandbox escape process.



First, we get code execution through RCE bugs in internet zone renderer. Then we exploit the SOP bypass to navigate to a local zone renderer. Next, we exploit the RCE bugs again. Finally, we use the Flash broker bug to escape the *Microsoft Edge* sandbox.

4.2.4. Patches

For the *Flash* broker bug, *Adobe* removed those vulnerable add-ins from macromedia.com.

For the SOP bypass bug, *Microsoft* added a file integrity level check in `IEGetZoneIUri`. *Microsoft Edge* can only navigate to an internet zone renderer if target file has a low integrity level label.

5. Conclusion

In this paper, we have presented

1. Two logical sandbox escape bug chain consists of three bugs for Microsoft Edge.
2. The internals of Microsoft Edge security architecture and inter-process communication.
3. How to abuse legitimate features to form logical bug chains, and logical sandbox escape on Windows.

Acknowledgments

We would like to thank Alex Ionescu (@aionescu) and James Forshaw (@tiraniddo) for their excellent talks, papers, and projects on Windows security.

We also thank Yang Yu (@tombkeeper) for supporting our research.

References

- [1]. <https://blogs.windows.com/msedgedev/2015/05/11/microsoft-edge-building-a-safer-browser/>
- [2]. Windows Internals Part 1. 7th Edition
- [3]. <https://arstechnica.com/features/2012/10/windows-8-and-winrt-everything-old-is-new-again/5/>
- [4]. <https://github.com/tyranid/WindowsRuntimeSecurityDemos/blob/master/The%20Inner%20Workings%20of%20the%20Windows%20Runtime.pdf>
- [5]. <https://docs.microsoft.com/en-us/windows/desktop/com/com-technical-overview>

- [6]. https://www.troopers.de/downloads/troopers17/TR17_Demystifying_%20COM.pdf
- [7]. <https://docs.microsoft.com/en-us/windows/desktop/com/security-in-com>
- [8]. <https://blogs.msdn.microsoft.com/ie/2008/03/11/ie8-and-loosely-coupled-ie-lcie/>
- [9]. WP-Asia-14-Yason-Diving-Into-IE10s-Enhanced-Protected-Mode-Sandbox
- [10]. <https://docs.microsoft.com/en-us/windows/desktop/SecAuthZ/implementing-an-appcontainer>
- [11]. <https://blogs.windows.com/msedgedev/2017/07/25/flash-on-windows-timeline/>