UNDERSTANDING THE MICROSOFT OFFICE 2013 PROTECTED-VIEW SANDBOX

Yong Chuan, Koh (@yongchuank)

2015/07/09

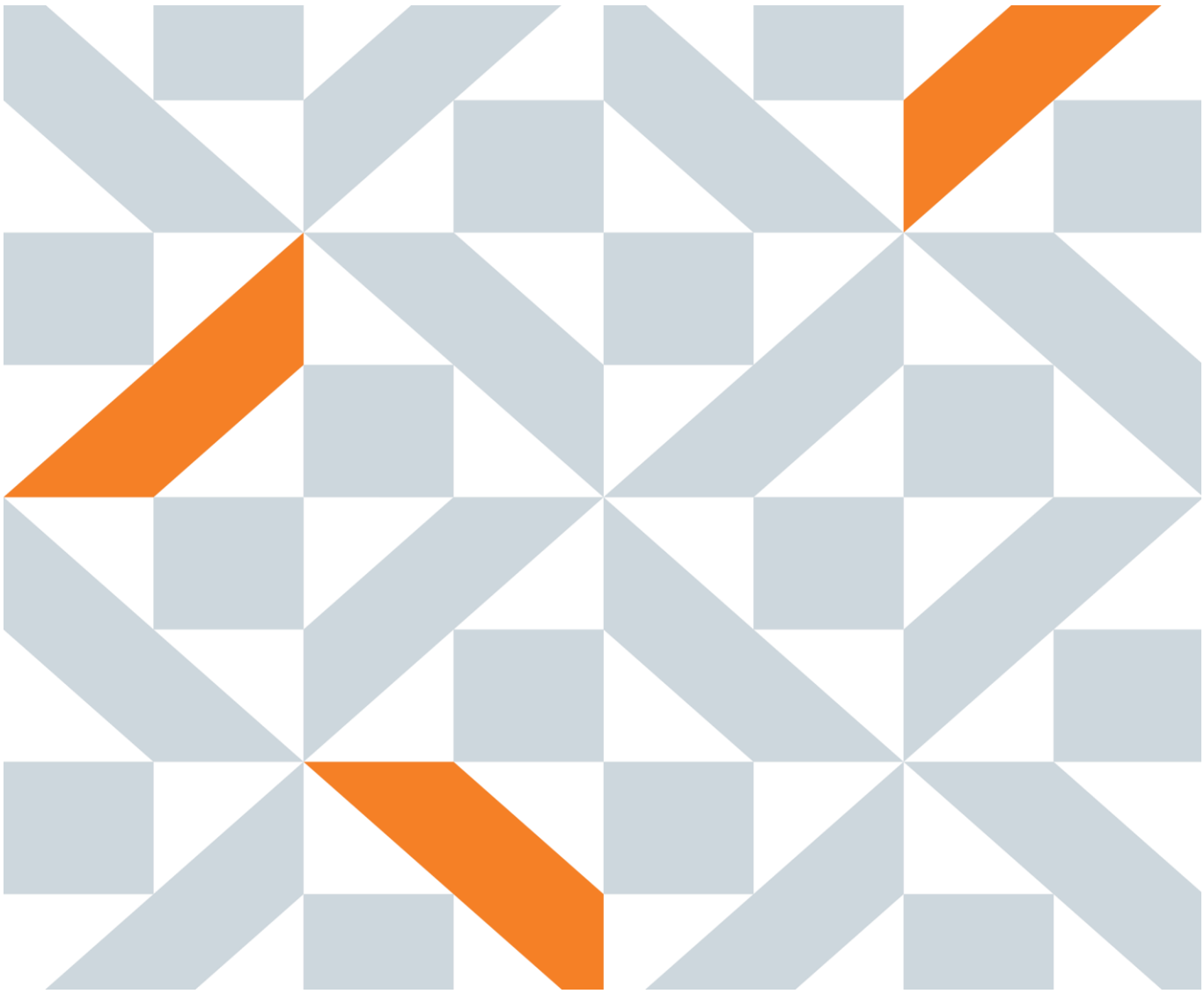


Table of Contents

1. Introduction

Sandbox is an increasingly popular technique adopted by software vendors to minimize the security impact of a compromised system. Wikipedia defines a sandbox as:

"…a sandbox is a security mechanism for separating running programs…A sandbox typically provides a tightly controlled set of resources for guest programs to run in ...A sandbox is implemented by executing the software in a restricted operating system environment, thus controlling the resources (…) that a process may use…" (1)

This definition suggests that imposing restrictions on an application's accessible system resources is central to sandboxing. And it is through these restrictions that the minimal security impact is achieved; the malware is still not able to access important information even though the system may be compromised. As a testament of its effectiveness, popular applications that have adopted sandboxing include Google Chrome, Internet Explorer and Adobe Reader.

The MS Office is no exception; its sandbox is implemented as the Protected-View feature since MS Office 2010. Unlike typical sandboxes, the Protected-View only sandboxes files that it deem as "untrusted", which is defined as Word, PowerPoint and Excel files that are opened from dangerous locations (2), and could also be identified from the "*Zone.Identifier:$DATA*" alternate-data-streams contents. In general, the Protected-View offers the user a read-only text-view representation of the file and disables all unnecessary features for this purpose.

The motivation for this research arises because at present, there are many excellent research performed on the sandboxes for Chrome (3), Internet Explorer Enhanced Protected Mode (IE EPM) (4) (5) and Adobe Reader (6) (7). However, no research has been performed on the MS Office Protected-View sandbox since its introduction 5 years ago. To add to the obscurity, Microsoft did not release any technical information on Protected-View as well. As a result, the wider community is left wondering about the mechanism behind it.

This whitepaper aims to bridge this gap in two parts. The first part attempts to describe the sandbox internals by discussing its architecture, how the sandbox is started and the system resource restrictions. The second part discusses the Inter-Process Communication (IPC) mechanism, including the mode of communication, internal objects that the broker uses, format of IPC messages and semantics of selected IPC messages. This whitepaper will also do a comparison with MS Office 2016 Preview (to check how the Protected-View sandbox may have evolve. Finally, the whitepaper will end with a conclusion.

With most of the Protected-View sandbox code found in MSO.DLL and WWLIB.DLL, this whitepaper is based on 32-bit MS Office 2013 v15.0.4420.1017, MSO.DLL v15.0.4420.1017 and WWLIB.DLL v15.0.4420.1017 running on Windows 8 (x86). For MS Office 2016 Preview, MSO.DLL v16.0.3930.1008 and WWLIB.DLL v16.0.3930.1008 are used.

2. Sandbox Internals

This first part of the paper aims to provide the reader with a holistic understanding of the Protected-View sandbox implementation through two sub-sections; the architecture of the sandbox and the system resources that the sandbox is restricted to. This knowledge should serve as the foundation for subsequent work to identify subtle weaknesses or attack surfaces.

2.1 Architecture

As the Protected-View sandbox architecture is unknown, the approach taken to "sketch" it is by comparison against the Internet Explorer (IE) Enhanced Protected Mode (EPM) sandbox model. It is chosen because of the likelihood of code reuse, design similarity and also the wealth of information from the community.

Like most architectures, the Protected-View model consists of the higher-privileged broker, lower-privileged sandbox and operating system. All untrusted files are rendered in the same sandbox process that utilizes the *AppContainer* or *Low-Integrity* technology, depending on the operating system, to restrict system resources.

In IE EPM, 3 main components are central to its implementation. These are the *Interception Component*, *Elevation-Policy Component* and *Inter-Process Communication (IPC) Component*. So it is fundamental to check whether these are present in Protected-View as well.

2.1.1 Interception Component

The IE EPM implements an *Interception Component* which uses a Shims mechanism to redirect selected API calls so that its sandbox can still support full functionalities and not disrupt users' experience. These API calls are redirected for three possible outcomes. First, it aims to seamlessly enable untrusted third-party plug-ins to continue functioning as before under sandbox restrictions. Second, some functionalities require higher privileges to execute (in the broker context). Third, it is forwarded to the *Elevation-Policy Component* to allow only trusted processes and COM to be created.

API hooking is one of the techniques to redirect API calls, and the patching of Import Address Table (IAT), Export Address Table (EAT) and function prologues are common methods to achieve this. In Protected-View, the presence of an *Interception Component* is verified by checking for these three methods of patching. The following results are yielded:

- All function prologues in both broker and sandbox processes are similar, indicating no inline-hooking.
- All function addresses in IAT and EAT references the respective modules range, indicating no IAT and EAT hooking.

2.1.2 Elevation Policy Manager

The *Elevation-Policy Component* is a subsequent decision-maker from *Interception Component*, and by deduction, the absence of the latter will indicate absence of the former as well. This hypothesis will be supported by another check for this component, but from another perspective. In IE EPM, the *Elevation-Policy*

Component is implemented in the *"HKEY_LOCAL_MACHINE\Software\Microsoft\Internet Explorer\Low Rights\ElevationPolicy\"* registry key, and contains the application names (*AppName*), paths (*AppPath*), *CLSID* and launch behaviour (*LaunchPolicyValue*).

To determine if this component is implemented in Protected-View, new registry keys found in MS Office 2013 (vs MS Office 2007) are identified. MS Office 2007 is chosen because it is the last version that does not offer the Protected-View feature. Therefore, if the component is implemented, its set of elevation policies should be found among these new registry keys, which are show below:

HKEY_CURRENT_USER\SOFTWARE\Microsoft\Office\15.0\	
Common\OverridePointerMode	Common\MathFonts*
Common\Feedback\AppUsageData_*	Common\ReviewCycle\ReviewToken
Common\General\ShownFirstRunOptin	Common\Roaming*
Common\General\LastAutoSavePurgeTime	Common\ServicesManagerCache\ ServicesCatalog*
Common\General\FileFormatBallotBox*	Word\File MRU*
Common\Identity\Version	Word\Options*
Common\Internet\WebServiceCache*	Word\Place MRU*
Common\LCCache\SmartArt\1033*	Word\Reading Locations\Document *
Common\LCCache\Themes\1033*	Word\Recent Templates\Change\ChangeI
Common\LCCache\WordDocBibs\1033*	Word\Resiliency\DisabledItems\
Common\LCCache\WordDocParts\1033*	Word\Security\Trusted Documents\LastPurgeTime
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Office\15.0	
Common\COM Compatibility\{CLSID}*	
Common\Config\{CLSID}*	
User Settings\Excel_Core\Create\Software\Microsoft\Internet Explorer\ProtocolExecute*\WarnOnOpen	
User Settings\PowerPoint_Core\Create\Software\Microsoft\Internet Explorer\ProtocolExecute*\WarnOnOpen	
User Settings\Word_Core\Create\Software\Microsoft\Internet Explorer\ProtocolExecute*\WarnOnOpen	
Word\Document Inspectors*	
Word\Text Converters\OOXML Converters\Import\IFDP*	
PowerPoint\Document Inspectors*	
Excel\Document Inspectors*	

And none of these keys have values similar to *AppName*, *AppPath*, *CLSID* and *LaunchPolicyValue*. Therefore, it is concluded that Protected-View does not implement the *Elevation-Policy Component*.

2.1.3 Inter-Process Communication

The Inter-Process Communication (IPC) is an important component in any sandbox architecture because both broker and sandbox processes would have to exchange data or make higher-privilege requests for a variety of reasons. IE EPM implements two types of IPC; the COM IPC and Shared-Memory IPC. In both cases, the broker would establish these 2 IPC channels (for COM IPC; initializing and marshalling the COM interface, for Shared-Memory IPC; creating shared memory sections in a private namespace with sandbox) and hand them over as part of the sandbox initialization.

In Protected-View, these 2 types of IPC channels are not instantiated during initialization, as discussed in *Sandbox Initialization* (section 2.2.1). Instead, it would implement only the Name-Pipe IPC as the mode of inter-process communication. The Name-Pipe IPC mechanism would be further discussed in *Inter-Process Communication (IPC) Mechanism* (section 3).

2.2 Sandbox Restrictions

This section examines the system resources that Protected-View sandbox can access by default. This will paint a clearer picture on the sandbox boundary and set the baseline of what may constitute a sandbox escape.

Starting with Windows 8, Microsoft introduces the *AppContainer* restriction mechanism that applications can utilize. In this mechanism, resources are restricted based on *capabilities* (8). Examples of standard capabilities are *musicLibrary*, *internetClient* and *webcam*, as defined in Winnt.h (9) or the registry key "*HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\DeviceAccess\CapabilityMappings*".

However, Microsoft does not assign standard capabilities to the Protected-View sandbox. Instead, it assigns only one undocumented *capability-SID S-1-15-3-2929230137-1657469040* that seem to be unique only to MS Office. Therefore understanding the system resources accessible to this capability will lead to a better understanding of the Protected-View sandbox restrictions. For this purpose, a brute-force approach is taken to list the file locations and registry keys that it can access, as well as its network ability. And likewise for the Protected-View *AppContainer-SID*. Unlike IE EPM, the "*ALL APPLICATION PACKAGES*" SID (S-1-15-2-1) is not being considered since MS Office is not a Metro application (yet).

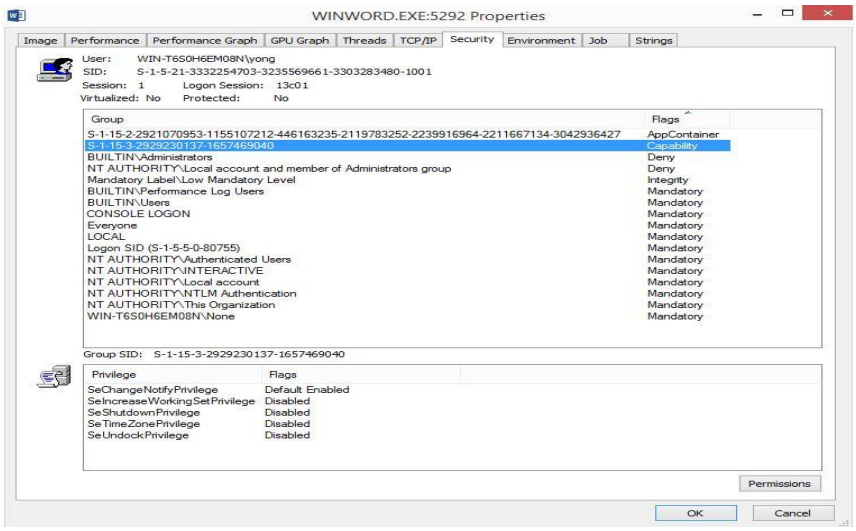
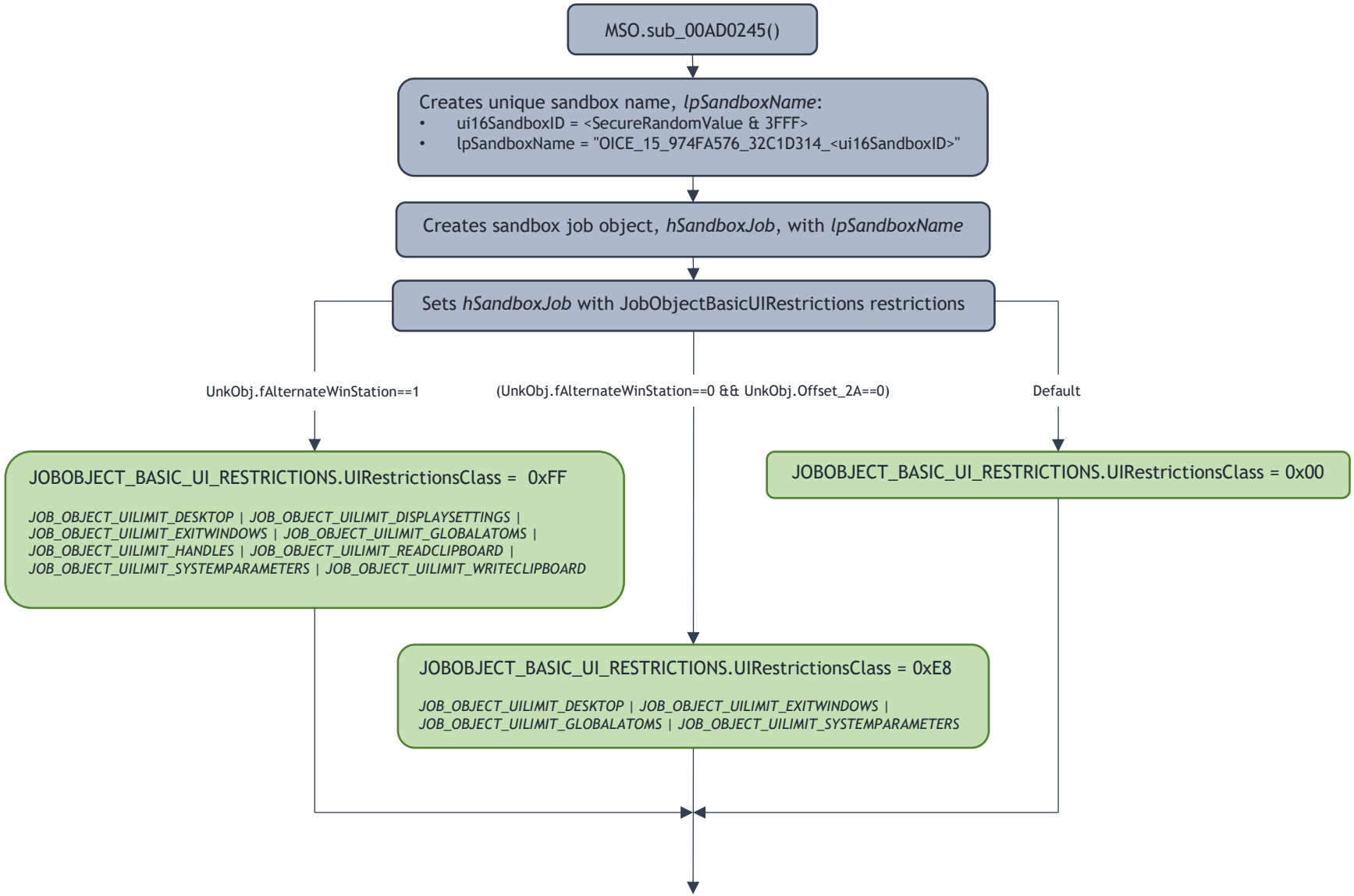


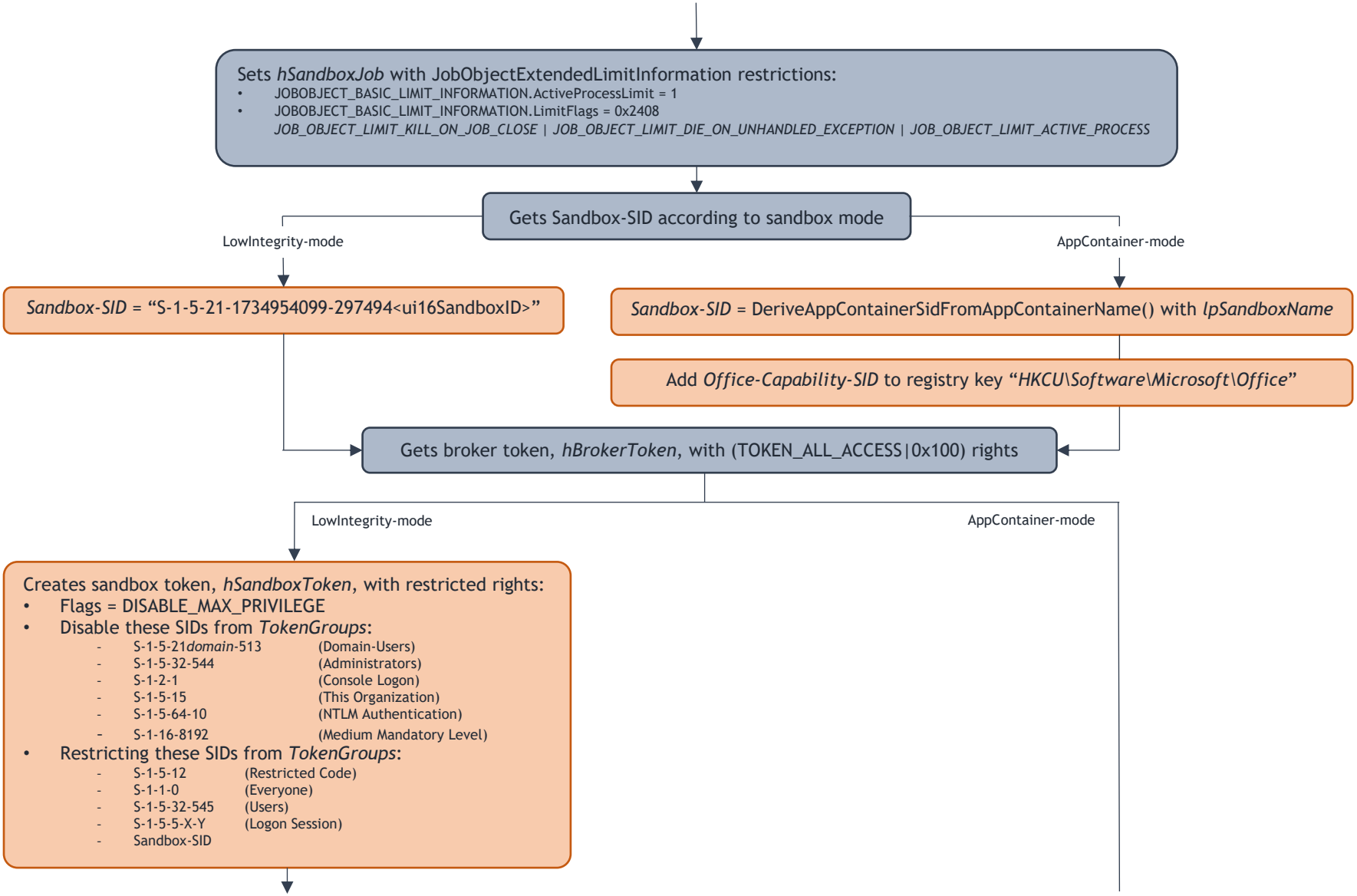
Figure 1: Capability-SID assigned to Protected-View sandbox

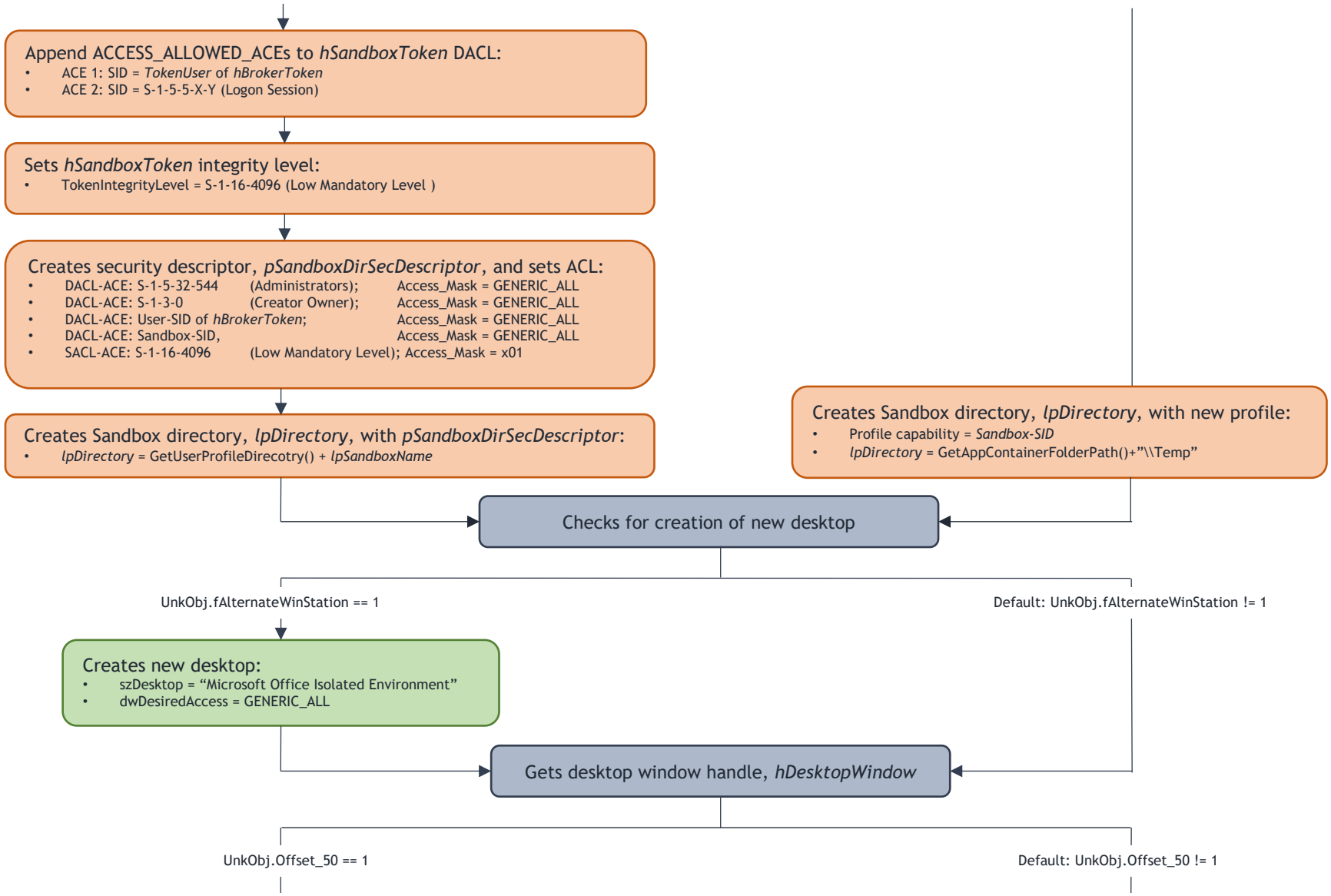
The first sub-section will describe the Protected-View initialization process to understand how the sandbox is to be created, and at the same time, identify any weakness. The subsequent sub-sections will list the accessible file locations, registry keys and network capability. For the rest of this whitepaper, *Sandbox-SID* and *AppContainer-SID* will be used interchangeably.

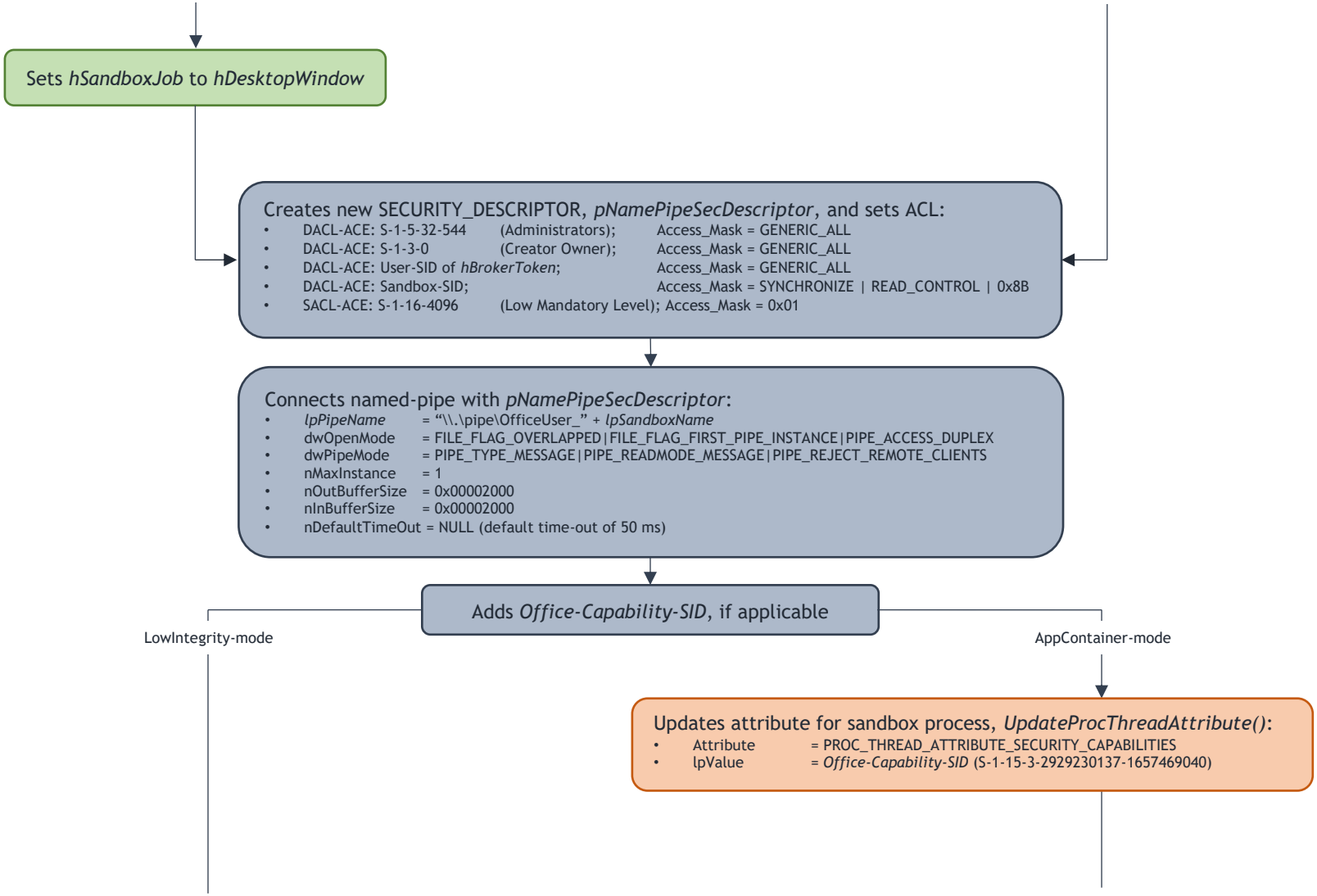
2.2.1 Sandbox Initialization

The sandbox initialization process begins at *MSO.sub_00AD0245()*, and undergoes this sequence of actions:


MSO.sub_00AD0245()
Creates unique sandbox name, lpSandboxName:
• ui16SandboxID = <SecureRandomValue & 3FFF>
• lpSandboxName = "OICE_15_974FA576_32C1D314_<ui16SandboxID>"
Creates sandbox job object, hSandboxJob, with lpSandboxName
Sets hSandboxJob with JobObjectBasicUIRestrictions restrictions
UnkObj.fAlternateWinStation==1
(UnkObj.fAlternateWinStation==0 && UnkObj.Offset_2A==0)
Default
JOBOBJECT_BASIC_UI_RESTRICTIONS.UIRestrictionsClass = 0xFF
JOB_OBJECT_UILIMIT_DESKTOP | JOB_OBJECT_UILIMIT_DISPLAYSETTINGS | JOB_OBJECT_UILIMIT_EXITWINDOWS | JOB_OBJECT_UILIMIT_GLOBALATOMS | JOB_OBJECT_UILIMIT_HANDLES | JOB_OBJECT_UILIMIT_READCLIPBOARD | JOB_OBJECT_UILIMIT_SYSTEMPARAMETERS | JOB_OBJECT_UILIMIT_WRITECLIPBOARD
JOBOBJECT_BASIC_UI_RESTRICTIONS.UIRestrictionsClass = 0xE8
JOB_OBJECT_UILIMIT_DESKTOP | JOB_OBJECT_UILIMIT_EXITWINDOWS | JOB_OBJECT_UILIMIT_GLOBALATOMS | JOB_OBJECT_UILIMIT_SYSTEMPARAMETERS
JOBOBJECT_BASIC_UI_RESTRICTIONS.UIRestrictionsClass = 0x00


Sets *hSandboxJob* with JobObjectExtendedLimitInformation restrictions:
• JOBOBJECT_BASIC_LIMIT_INFORMATION.ActiveProcessLimit = 1
• JOBOBJECT_BASIC_LIMIT_INFORMATION.LimitFlags = 0x2408
JOB_OBJECT_LIMIT_KILL_ON_JOB_CLOSE | JOB_OBJECT_LIMIT_DIE_ON_UNHANDLED_EXCEPTION | JOB_OBJECT_LIMIT_ACTIVE_PROCESS
Gets Sandbox-SID according to sandbox mode
LowIntegrity-mode
AppContainer-mode
Sandbox-SID = "S-1-5-21-1734954099-297494<ui16SandboxID>"
Sandbox-SID = DeriveAppContainerSidFromAppContainerName() with *lpSandboxName*
Add *Office-Capability-SID* to registry key "*HKCU\Software\Microsoft\Office*"
Gets broker token, *hBrokerToken*, with (TOKEN_ALL_ACCESS|0x100) rights
LowIntegrity-mode
AppContainer-mode
Creates sandbox token, *hSandboxToken*, with restricted rights:
• Flags = DISABLE_MAX_PRIVILEGE
• Disable these SIDs from *TokenGroups*:
- S-1-5-21*domain*-513 (Domain-Users)
- S-1-5-32-544 (Administrators)
- S-1-2-1 (Console Logon)
- S-1-5-15 (This Organization)
- S-1-5-64-10 (NTLM Authentication)
- S-1-16-8192 (Medium Mandatory Level)
• Restricting these SIDs from *TokenGroups*:
- S-1-5-12 (Restricted Code)
- S-1-1-0 (Everyone)
- S-1-5-32-545 (Users)
- S-1-5-5-X-Y (Logon Session)
- Sandbox-SID


Append ACCESS_ALLOWED_ACEs to hSandboxToken DACL:
• ACE 1: SID = TokenUser of hBrokerToken
• ACE 2: SID = S-1-5-5-X-Y (Logon Session)
Sets hSandboxToken integrity level:
• TokenIntegrityLevel = S-1-16-4096 (Low Mandatory Level)
Creates security descriptor, pSandboxDirSecDescriptor, and sets ACL:
• DACL-ACE: S-1-5-32-544 (Administrators); Access_Mask = GENERIC_ALL
• DACL-ACE: S-1-3-0 (Creator Owner); Access_Mask = GENERIC_ALL
• DACL-ACE: User-SID of hBrokerToken; Access_Mask = GENERIC_ALL
• DACL-ACE: Sandbox-SID, Access_Mask = GENERIC_ALL
• SACL-ACE: S-1-16-4096 (Low Mandatory Level); Access_Mask = x01
Creates Sandbox directory, lpDirectory, with pSandboxDirSecDescriptor:
• lpDirectory = GetUserProfileDirecotry() + lpSandboxName
Creates Sandbox directory, lpDirectory, with new profile:
• Profile capability = Sandbox-SID
• lpDirectory = GetAppContainerFolderPath()+"\\Temp"
Checks for creation of new desktop
UnkObj.fAlternateWinStation == 1
Default: UnkObj.fAlternateWinStation != 1
Creates new desktop:
• szDesktop = "Microsoft Office Isolated Environment"
• dwDesiredAccess = GENERIC_ALL
Gets desktop window handle, hDesktopWindow
UnkObj.Offset_50 == 1
Default: UnkObj.Offset_50 != 1


Sets hSandboxJob to hDesktopWindow
Creates new SECURITY_DESCRIPTOR, pNamePipeSecDescriptor, and sets ACL:
• DACL-ACE: S-1-5-32-544 (Administrators); Access_Mask = GENERIC_ALL
• DACL-ACE: S-1-3-0 (Creator Owner); Access_Mask = GENERIC_ALL
• DACL-ACE: User-SID of hBrokerToken; Access_Mask = GENERIC_ALL
• DACL-ACE: Sandbox-SID; Access_Mask = SYNCHRONIZE | READ_CONTROL | 0x8B
• SACL-ACE: S-1-16-4096 (Low Mandatory Level); Access_Mask = 0x01
Connects named-pipe with pNamePipeSecDescriptor:
• lpPipeName = "\\.\pipe\OfficeUser_" + lpSandboxName
• dwOpenMode = FILE_FLAG_OVERLAPPED|FILE_FLAG_FIRST_PIPE_INSTANCE|PIPE_ACCESS_DUPLEX
• dwPipeMode = PIPE_TYPE_MESSAGE|PIPE_READMODE_MESSAGE|PIPE_REJECT_REMOTE_CLIENTS
• nMaxInstance = 1
• nOutBufferSize = 0x00002000
• nInBufferSize = 0x00002000
• nDefaultTimeOut = NULL (default time-out of 50 ms)
Adds Office-Capability-SID, if applicable
LowIntegrity-mode
AppContainer-mode
Updates attribute for sandbox process, UpdateProcThreadAttribute():
• Attribute = PROC_THREAD_ATTRIBUTE_SECURITY_CAPABILITIES
• lpValue = Office-Capability-SID (S-1-15-3-2929230137-1657469040)

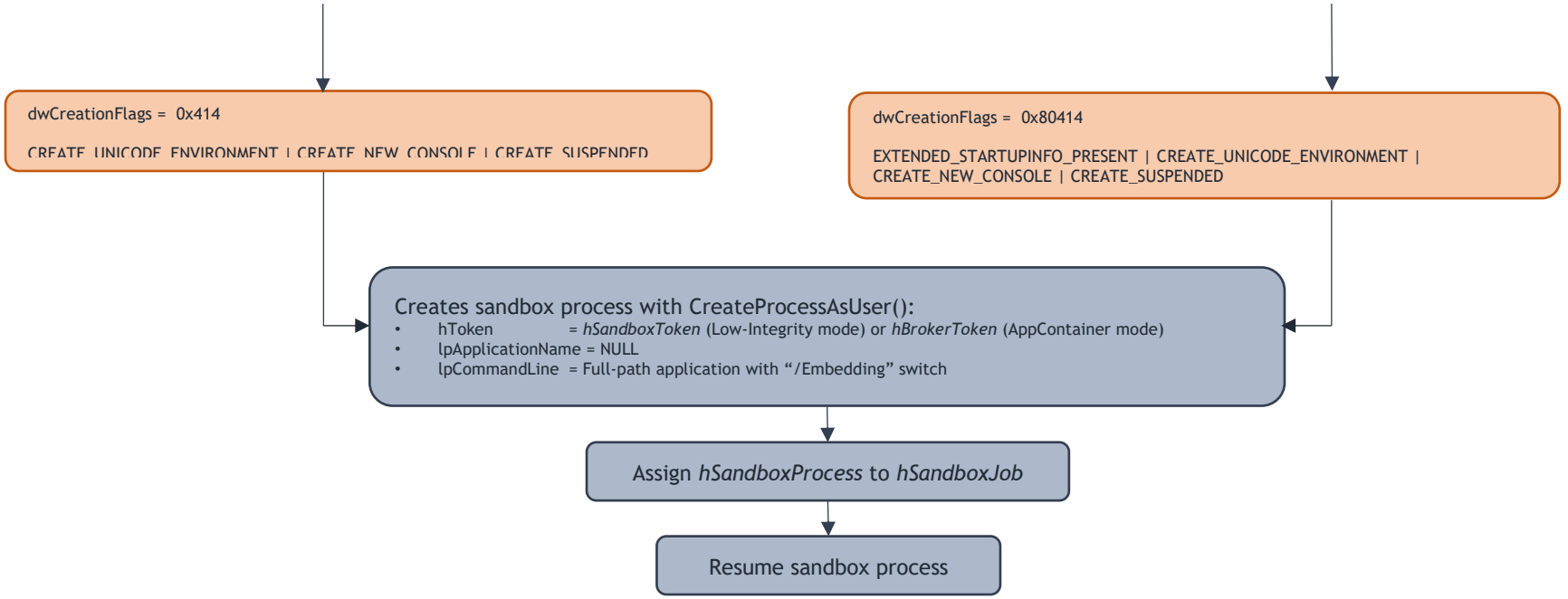


Figure 2: Flowchart of Protected-View initialization. Differences between Low-Integrity mode and AppContainer mode are shown in orange blocks. Differences due to application settings are shown in green blocks.

To start the sandbox as a *Low-Integrity* or *AppContainer* process depends on its operating system, which the broker will discover with either of these two methods:

1. Checks if *HKLM\Software\Microsoft\Office\15.0\Common\Security\UserAppContainer* is 1, or
2. Checks if *GetProcAddress*() for *Userrenv.DeriveAppContainerSidFromAppContainerName()* and Userrenv.*GetAppContainerFolderPath()* succeeds

2.2.2 File Locations

File Locations	Access Mask
Sandbox-SID (*S-1-15-2-*-*-*-*-*-*-)**	
%UserProfile%\AppData\Local\Packages\<lpSandboxName>*	STANDARD_RIGHTS_ALL | 0x1FF
Office-Capability-SID (*S-1-15-3-2929230137-1657469040*)	
None	None

The accessible file locations are determined only by *Sandbox-SID*, and restricts Protected-View only to the sandbox directory. This resonates with the observation that only the *Sandbox-SID* is granted to sandbox directory during initialization. The *Capability-SID* does not allow access to file locations.

2.2.3 Registry Keys

Registry Keys	Access Mask
Sandbox-SID (*S-1-15-2-*-*-*-*-*-*-)**	
HKCR\Local Settings\Software\Microsoft\Windows\CurrentVersion\AppContainer\Mappings\<*Sandbox-SID*>	KEY_READ
HKCR \Local Settings\Software\Microsoft\Windows\CurrentVersion\AppContainer\Mappings\<*Sandbox - SID*>\Children	KEY_ALL_ACCESS
HKCR \Local Settings\Software\Microsoft\Windows\CurrentVersion\AppContainer\Storage\<*lpSandboxName*>	KEY_ALL_ACCESS
HKCR \Local Settings\Software\Microsoft\Windows\CurrentVersion\AppContainer\Storage\<*lpSandboxName*>\Children	KEY_ALL_ACCESS

HKCR \Software\Classes\Local Settings\Software\Microsoft\Windows\CurrentVersion\AppContainer\Mappings\<*Sandbox-SID*>	KEY_READ
HKCR \Software\Classes\Local Settings\Software\Microsoft\Windows\CurrentVersion\AppContainer\Mappings\<*Sandbox-SID*>\Children	KEY_ALL_ACCESS
HKCR \Software\Classes\Local Settings\Software\Microsoft\Windows\CurrentVersion\AppContainer\Storage\<*lpSandboxName*>	KEY_ALL_ACCESS
HKCR \Software\Classes\Local Settings\Software\Microsoft\Windows\CurrentVersion\AppContainer\Storage\<*lpSandboxName*>\Children	KEY_ALL_ACCESS
HKEY_USERS\<WinUser-SID>\Software\Classes\Local Settings\Software\Microsoft\Windows\CurrentVersion\AppContainer\Mappings\<*Sandbox-SID*>	KEY_READ
HKEY_USERS\<WinUser-SID>\Software\Classes\Local Settings\Software\Microsoft\Windows\CurrentVersion\AppContainer\Mappings\<*Sandbox-SID*>\Children	KEY_ALL_ACCESS
HKEY_USERS\<WinUser-SID>\Software\Classes\Local Settings\Software\Microsoft\Windows\CurrentVersion\AppContainer\Storage\<*lpSandboxName*>	KEY_ALL_ACCESS
HKEY_USERS\<WinUser-SID>\Software\Classes\Local Settings\Software\Microsoft\Windows\CurrentVersion\AppContainer\Storage\<*lpSandboxName*>\Children	KEY_ALL_ACCESS
Office-Capability-SID (*S-1-15-3-2929230137-1657469040*)	
HKCU\Software\Microsoft\Office*	KEY_READ
HKEY_USERS\<*WinUser-SID*>\Software\Microsoft\Office*	KEY_READ

In summary, the *Sandbox-SID* allows access to sandbox-related registry keys (mostly with KEY_ALL_ACCESS), and the *Capability-SID* allows KEY_READ access to Office-related registry keys.

2.2.4 Network Connections

Without *InternetClient* or other network capabilities, the sandbox is not allowed to make outbound connections to Internet endpoints in *AppContainer* mode. And neither does the *Capability-SID* (*S-1-15-3-2929230137-1657469040*) allows the sandbox to do so, failing with the WSAEACCES “Permission Denied” error.

2.3 Comments

In summary, the final Protected-View architecture is illustrated below:

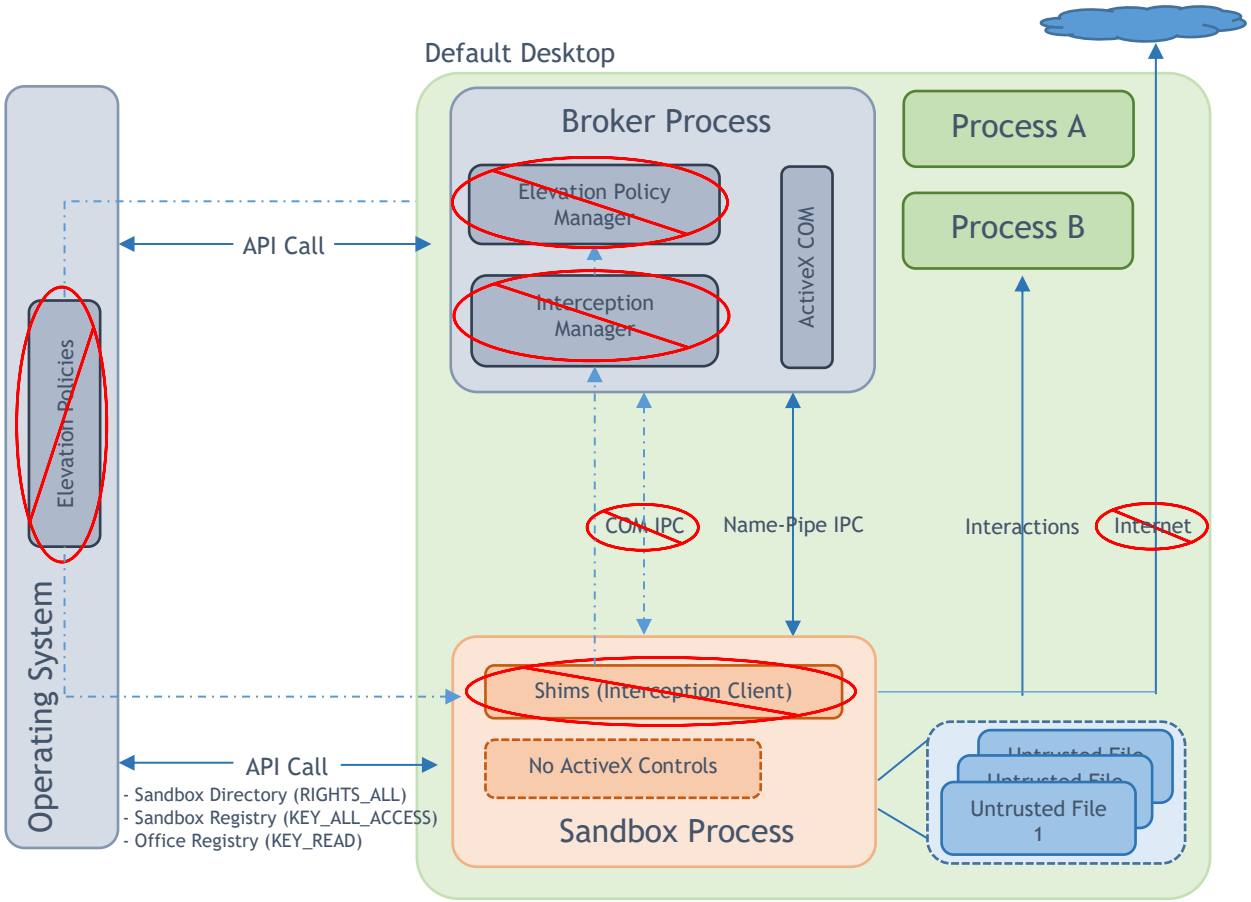


Figure 3: Illustration of the Protected-View Architecture

The absence of *Interception Component* and *Elevation-Policy Component* is not surprising because the Protected-View feature is intended to offer only a text-preview of the file. It does not have to support the full functionalities of the application and therefore, can simplify its architecture. The lack of support for ActiveX controls also implies that basic functions such as saving or printing are not available in the Protected-View mode.

In practice, sandbox process should be created with a restricted access token, restricted job object, and GUI sub-system isolation (10). But as the initialization process shows, by default, the Protected-

View is created with no job UI restrictions (11) and no desktop/windowstation isolation. As a result, the sandbox is able to read from/write to the clipboard shared with other processes in the default desktop, perform screen scraping and screen captures. All of these are known issues that affects IE EPM (4).

With *Capability-SID S-1-15-3-2929230137-1657469040*, the sandbox can read registry keys related to MS Office. Among these, 2 registry keys might be of interest to attackers. The first is

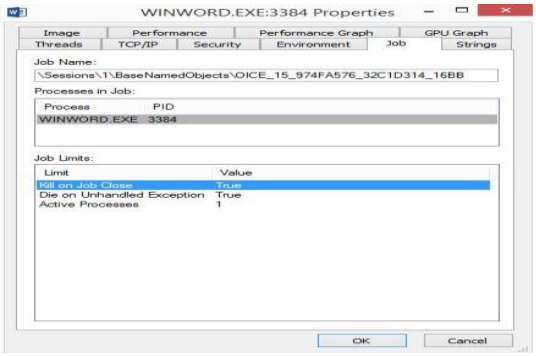


Figure 4: No job UI restrictions

"HKCU\Software\Microsoft\Office\15.0\Common\ServicesManagerCache", which stores the user's Microsoft Live account information such as the connection state account UUID if he signs in from MS Office. The second is *"HKCU\Software\Microsoft\Office\15.0\Word\Trusted Locations"*, which defines the file locations that are excluded from Protected-View mode.

The *Capability-SID* does not allow sandbox to connect to Internet endpoints, which interestingly, alleviate the issues from lack of job UI restrictions and desktop isolation because the disclosed information cannot be sent out from sandbox.

Finally, the Protected-View file locations restrictions does not apply to a VMWare-HGFS file system. Hence the sandbox can still read from/write to the mapped-drive from VMWare file-sharing feature.

Other system resource restrictions that are specific to *AppContainer* have been discussed in Mark Vincent Yason's IE EPM research (4) and shall not be repeated here.

3. Inter-Process Communication (IPC) Mechanism

The second part of this paper discusses the inter-process communication (IPC) mechanism used in the Protected-View sandbox model. As with all sandbox models, the IPC is an attractive attack vector for sandbox escape issues because of its interaction with the higher-privilege broker. Examples are such sandbox escapes are CVE-2013-0641 (memory corruption vulnerability in the Adobe Reader sandbox) CVE-2013-3186 and CVE-2013-4015 (policy-check vulnerabilities in IE EPM).

This part begins with a section on the internal objects used for sandboxing. Next, the code in MSO.DLL and WWLIB.DLL that is responsible for servicing the messages are identified. Finally, the format and meaning of some notable IPC messages are discussed before concluding this section with some overall comments.

3.1 IPC Objects

Since all untrusted files are parsed in a shared Protected-View sandbox process, the broker need to keep an overview of the sandboxing state (e.g. how many files are sandboxed, locations of these files, etc.) and also track the state of each file (e.g. identify keyboard inputs and hyperlink requests to corresponding files). This information is stored by the broker in these IPC objects, and referenced by

many IPC messages to retrieve or update the fields. An overview of how these objects are related is shown below.

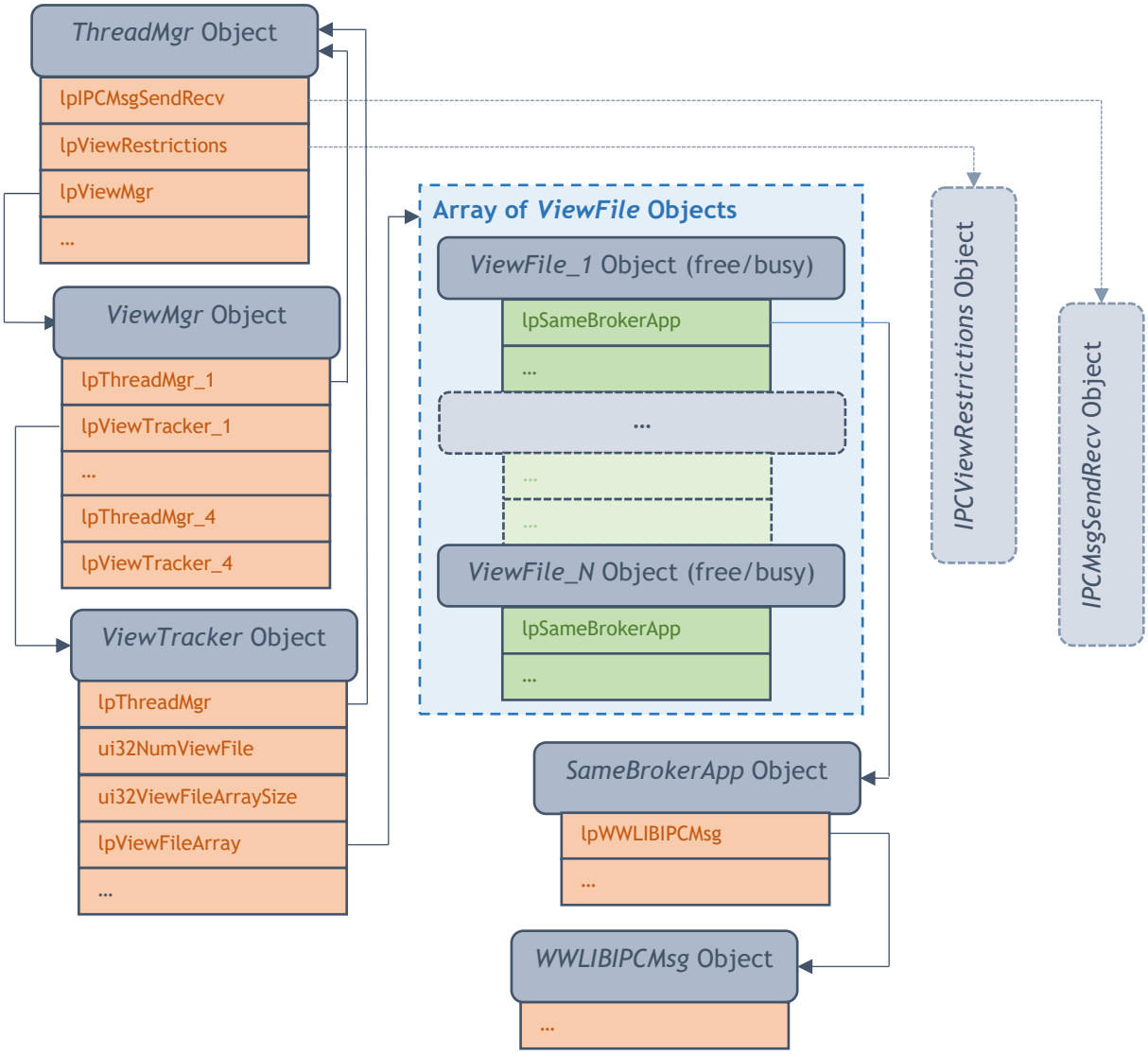


Figure 5: Overview of Protected-View IPC objects

The overview starts with the *ThreadMgr* object, which also hold references to the *IPCMsgSendRecv* and *IPCViewRestrictions* objects. The former describes the status of IPC pipe and its buffer to read from/write to. The latter describes the information used during sandbox initialization. Both are of little interest in understanding the IPC messages and not elaborated further.

The details of each object is shown here, with fields whose meanings are unclear omitted for brevity.

ThreadMgr Object			
Offset	**Size**	**Field**	**Comment**

00	LPVOID	lpVTable	RVA MSO.4044BC
04	UINT32	ui32Status	-
08	LPVOID	lpIPCMsgSendRecv	Pointer to an object that sends/receives IPC messages
0C	LPVOID	lpViewRestrictions	Pointer to an object describing the sandbox restrictions
10	LPVOID	lpViewMgr	Pointer to *ViewMgr* object

***ViewMgr* Object**			
Offset	**Size**	**Field**	**Comment**
00	LPVOID	lpVTable	RVA MSO.3B7A70
04	UINT32	ui32Num	Number of untrusted files + 2
08	LPVOID	lpThreadMgr_1	-
0C	LPVOID	lpViewTacker_1	Pointer to *ViewTracker* object
10	HANDLE	hEvent_1	-
1C	LPVOID	lpUnkObject	-
20	LPVOID	lpThreadMgr_2	NULL
24	LPVOID	lpViewTracker_2	NULL
28	HANDLE	hEvent_2	NULL
34	LPVOID	lpUnkObject	-
38	LPVOID	lpThreadMgr_3	NULL
3C	LPVOID	lpViewTracker_3	NULL
40	HANDLE	hEvent_3	NULL
4C	LPVOID	lpUnkObject	-
50	LPVOID	lpThreadMgr_4	NULL
54	LPVOID	lpViewTracker_4	NULL
58	HANDLE	hEvent_4	NULL
64	LPVOID	lpUnkObject	-

***ViewTracker* Object**			
Offset	**Size**	**Field**	**Comment**
00	LPVOID	lpVTable	RVA MSO.3B7A64
04	UINT32	ui32NumObjects	See explanation below
0C	LPVOID	lpThreadMgr	Pointer to *ThreadMgr* object
14	UINT32	ui32NumViewFile	Number of Protected-View files, or number of busy slots in *ViewFilesArray*
18	UINT32	ui32ViewFileArraySize	Size of *ViewFilesArray*
1C	UINT32	ui32Unknown	Higher 2 bytes are used to calculate new *ui32ViewFilesArraySize* when *ViewFilesArray* is full
20	LPVOID	lpViewFileArray	Pointer to an array of *ViewFile* objects

The *lpViewTracker->ui32NumObjects* shows the total number of *ViewFile* objects (i.e. *lpViewTracker->ui32NumViewFiles*) + *lpViewFile->lpSameBrokerApp* objects + 1. The *lpViewFile->lpSameBrokerApp* object is instantiated only if the broker is WINWORD.EXE, EXCEL.EXE or POWERPNT.EXE (File 1a). This is in contrast to opening the untrusted file in OUTLOOK.EXE directly (**Error! Reference source ot found.**File 2a) or with Word previewer in OUTLOOK.EXE (**Error! Reference source not ound.**File 1b and 2b), whereby *lpViewTracker->ui32NumObjects* = *lpViewTracker->ui32NumViewFiles* + 1.

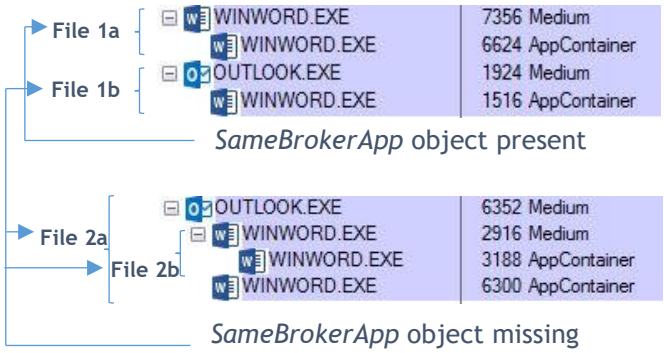


Figure 5: (File 1a) Directly from WINWORD.EXE. (File 1b, 2b) With Word previewer in OUTLOOK.EXE. (File 2a) Directly from OUTLOOK.EXE

The *lpViewTacker->lpViewFilesArray* points to a dynamic array of *ViewFile* objects that grows to a new size influenced b

ViewFile Object (size 0x1C)			
Offset	**Size**	**Field**	**Comment**
00	LPVOID	Field_00	Pointer to unknown object
04	UINT32	ui32ViewID	Unique ID to identify respective untrusted file. Corresponds to sequence number of file opened, starting from 1. Used in IPC messages.
08	HWND	hOPHWnd	hWnd for window with class “OPH Previewer Window”
0C	LPWSTR	lpwFileName	Pointer to full-path to original file
10	LPWSTR	lpwTemporaryFileName	Pointer to full-path to temporary file created in Protected-View directory
14	LPVOID	lpSameBrokerApp	Pointer to *SameBrokerApp* object
18	UINT32	ui32SessionEnableHyperlinks	Used in Tag 0x091000. If 1, user has allowed URLs for this session, and at least 1 URL has succeeded. Otherwise, 0.

SameBrokerApp Object			
Offset	**Size**	**Field**	**Comment**
00	LPVOID	lpVTable	RVA MSO.3B79A8
0C	LPVOID	lpWWLIBIPCMsg	Pointer to *WWLIBIPCMsg* object
18	UINT32	ui32ViewID	-

84	HWND	hOPHParentWnd	*hWnd* for window with class "OPH Previewer Parent Window". Used in 0x061000 and 0x101000 IPC message
90	LPVOID	lpDRMStream	Used in 0x081000 IPC message
A4	UINT32	ui32TaskListIndex	*lpTaskList* current index
A8	UINT32	ui32TaskListSize	*lpTaskList* maximum size
B0	LPVOID	lpTaskList	Pointer to an array of *ProtectedViewTask* objects. Used in 0x0B1000 IPC message

***WWLIBIPCMsg* Object**			
Offset	**Size**	**Field**	**Comment**
00	LPVOID	lpVTable	RVA wwlib.1A5B20
1C	UINT32	ui32IPC0A1100	0 or 1, depending on 0x0A1100 IPC request
20	UCHAR [0x2C]	uchIPC091100Contents	Buffer storing IPC 0x091100 message contents
4C	UINT32	ui32IPC071100MsgID	MsgID of IPC 0x071100 message
50	UINT32	ui32IPC081100MsgID	MsgID of IPC 0x081100 message
54	UINT32	ui32IPC091100MsgID	MsgID of IPC 0x091100 message
58	UINT32	ui32IPC031100MsgID	MsgID of IPC 0x031100 message
5C	UINT32	ui32IPC041100MsgID	MsgID of IPC 0x041100 message
60	UINT32	ui32IPC0E1100MsgID	MsgID of IPC 0x0E1100 message
64	UCHAR [0x24]	uchIPC041100Contents	Buffer storing IPC 0x041100 message contents
8C	UCHAR [0x1D4]	uchIPC031100Contents	Buffer storing IPC 0x031100 message contents
268	UINT32	ui32Boolean	0 or 1

In general, the *SameBrokerApp* object is mainly referenced by the 22 MSO.DLL IPC messages (*MSO-group*) while the *IPCWWLIBMsg* object is referenced by the remaining 16 WWLIB.DLL IPC messages (*WWLIB-group*). And because both objects are not always instantiated, of these 38 IPC messages, only a subset of 22 MSO.DLL (that references *ViewFile*) is always "serviceable".

Besides the instantiation of *SameBrokerApp* object, the global variable *MSO.015E9A9C* also represent the circumstance in which the untrusted file is opened. For example, if it is opened with OUTLOOK.EXE as broker, *MSO.015E9A9C* will be 6. If the file is opened with WINWORD.EXE as broker, then *MSO.015E9A9C* will be 0. The 0x011000, 0x081000 and 0x0A1000 messages are affected by *MSO.015E9A9C*.

3.2 IPC Parsers

The broker reads the pipe for IPC requests in the callback function *MSO.sub_00AD45C0()*, calls *lpViewMgr-> lpVTable.sub_003B7A80()* to service the completed message and sends back the result back to sandbox in *MSO.sub_011971A1()*.

The *lpViewMgr->lpVTable.sub_003B7A80()* function eventually calls into *MSO.sub_00AD4B06()* for the *MSO-group* or *WWLIB.sub_00E85DD3()* for the *WWLIB-group* to service individual IPC requests.

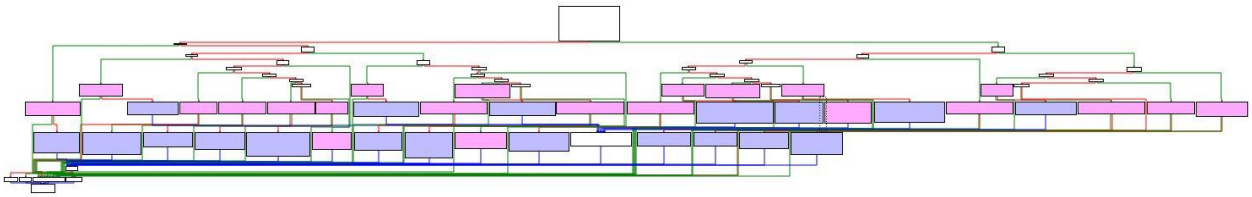


Figure 6: MSO.sub_00AD4B06()

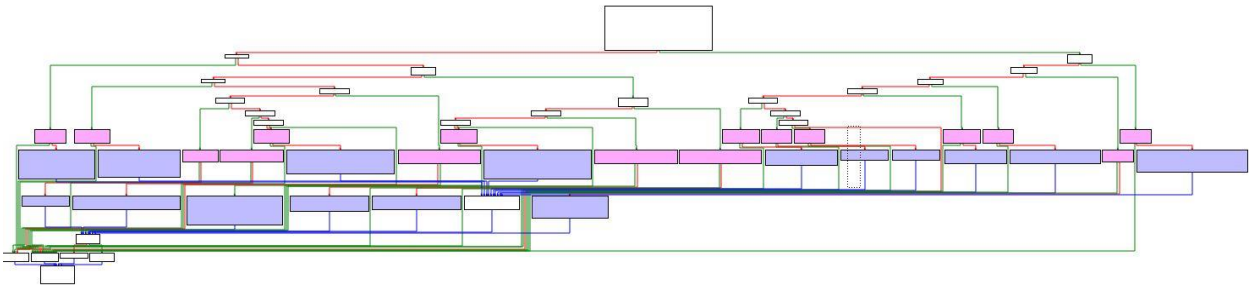


Figure 7: WWLIB.sub_00E85DD3()

In these two routines, the pink basic-blocks perform sanity checks on each respective IPC message, such as whether the message sizes are correct (either by specific value if message has fixed size, or by comparing bytes received with the *MsgSize* field), or the length of WCHAR strings, or to update WCHAR string pointers. Messages that fail sanity checks are discarded. Each subsequent purple basic-block relate to the code to service the IPC requests.

3.3 IPC Format

Both IPC requests and responses follow a certain format; each consist of a compulsory message header followed by a (sometimes optional) message-specific body. Each IPC message cannot exceed 0x2000 bytes in bytes, as specified by buffer size during name-pipe creation in sandbox initialization.

In general the 38 IPC requests can be classified into 4 groups; those pertaining to the sandbox features-specific purposes, for Office-alert prompts, for sandbox status updates and for other miscellaneous purposes. The following sections describe the format and semantics of selected IPC messages, and the rest are documented in *Appendix A*.

3.3.1 Message Header

The message headers for *MSO-group* and *WWLIB-group* are differentiated only by *ui32ViewID*, and is used by broker to fetch the corresponding *ViewFile*:

Message Header for *MSO-Group* (size 0x10-bytes)		
Size	**Field**	**Comment**
UINT32	ui32VirtualKey	Virtual-key code used only for the 0x071000 request. Otherwise ignored.
UINT32	ui32MsgTag	Type of (22) IPC request. For *MSO-group*, this ranges from 0x001000 to 0x151000 at an increment of 0x010000.
UINT32	ui32MsgID	Matches the IPC request to its reply. Increments from 0, but could also take arbitrary value.
UINT32	ui32MsgSize	Total size of the IPC message (ie: header + body), in bytes. Minimum value of sizeof(Header), maximum value of 0x2000. Otherwise message is considered malformed and discarded.
Message Header for *WWLIB-Group* (size 0x14-bytes)		
UINT32	ui32VirtualKey	-
UINT32	ui32MsgTag	Type of (16) IPC request. For WWLIB-group, this ranges from 0x001100 to 0x0F1100 at an increment of 0x010000.
UINT32	ui32MsgID	-
UINT32	ui32MsgSize	-
UINT32	ui32ViewID	Identifies the untrusted file that made the IPC request. Corresponds to *lpViewFile->ui32ViewID*

Although *lpViewFile->ui32ViewID* starts from 1 for the first untrusted file, a value of 0 would still service the same *lpViewFile* object.

The header for IPC responses has a similar 0x10-bytes size format, with the *ui32VirtualKey* field replaced by *ui32Status*, and indicates the status of IPC request. Below is a non-exhaustive list of known possible values for *ui32Status*:

ui32Status	Meaning
0	Requested IPC operation is successful
0x8007000E	Valid IPC message format , but error servicing/parsing request
0x80070057	Invalid IPC message format
0x80004001	Invalid *ui32MsgTag*
0x80004005	Invalid *ui32ViewID*, resulting in corresponding *ViewFile* object not found

3.3.2 Message 0x001000

The 0x001000 message body is optional and will be ignored. For this request, broker will duplicate the "*HKEY_CURRENT_USER\Software\Policies*" registry key if it is present, with KEY_READ access and return the duplicated handle to sandbox. The 0x001000 response takes this format:

Message Body for 0x001000 Response		
Size	**Field**	**Comment**
UINT32	MSO.15E9A9C	Global DWORD
HKEY	hDuplicatedKey	-
UINT32	ui32Unknown	1 if byte MSO.1624518!=0, 0 if otherwise

The broker duplicates this registry key on every request and does not record the duplication count.

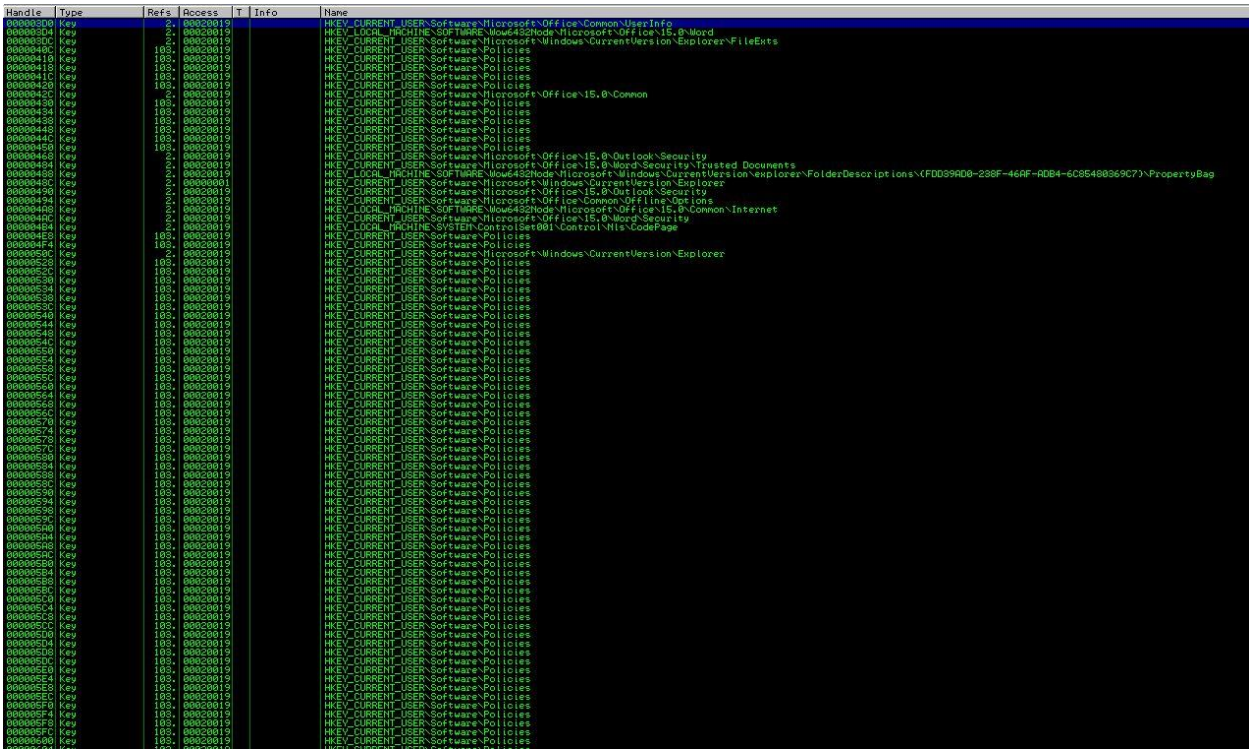


Figure 8: Registry key "HKEY_CURRENT_USER\Software\Policies" is duplicated on each request

This IPC request is probably intended as a work-around the registry keys restriction imposed on the sandbox (*Registry Keys* section 2.2.3).

3.3.3 Message 0x011000

The 0x011000 IPC request has this format:

Message Body for 0x011000 Request		
Size	**Field**	**Comment**
LPVOID	lpMarshaledInterface	Pointer to *uchMarshalledInterface*, or Null
UINT32	ui32MarshaledInterfaceSize	Size of *uchMarshaledInterface*, in bytes
_UUID	_uuid	See explanation below

UCHAR[]	uchMarshaledInterface	Marshalled interface as OBJREF structure (12)

The *_uuid* takes 4 possible UUID values for these types of Office previewer:

Previewer	UUID	Related Registry Keys
Word	{84F66100-FF7C-4FB4-B0C0-02CD7FB668FE}	• HKLM\SOFTWARE\Classes\CLSID\{*uuid*}\DisableLowILProcessIsolation = 0x00000001 • HKLM\SOFTWARE\Classes\CLSID\{*uuid*}\DisplayName = "Microsoft Word Previewer" • HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\PreviewHandlers\{*uuid*}= "Microsoft Word previewer"
Excel	{00020827-0000-0000-C000-000000000046}	• HKLM\SOFTWARE\Classes\CLSID\{*uuid*}\DisableLowILProcessIsolation = 0x00000001 • HKLM\SOFTWARE\Classes\CLSID\{*uuid*}\DisplayName = "Microsoft Excel Previewer" • HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\PreviewHandlers\{*uuid*} = "Microsoft Excel previewer"
PowerPoint	{65235197-874B-4A07-BDC5-E65EA825B718}	• HKLM\SOFTWARE\Classes\CLSID\{*uuid*}\DisableLowILProcessIsolation = 0x00000001 • HKLM\SOFTWARE\Classes\CLSID\{*uuid*}\DisplayName = "Microsoft PowerPoint Previewer" • HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\PreviewHandlers\{*uuid*}= "Microsoft PowerPoint previewer"
Visio	{21E17C2F-AD3A-4B89-841F-09CFE02D16B7}	• HKLM\SOFTWARE\Classes\CLSID\{*uuid*}\DisableLowILProcessIsolation = 0x00000001 • HKLM\SOFTWARE\Classes\CLSID\{*uuid*}\DisplayName = "Microsoft Visio Previewer" • HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\PreviewHandlers\{*uuid*}= "Microsoft Visio previewer"

If *_uuid* is one of these four previewers, the broker then unmarshals the *uchMarshaledInterface* interface and *IStream::Release* the *IStream* object:

WINOLEAPI CreateStreamOnHGlobal ()		
HGOBAL	uchMarshaledInterface	
BOOL	FALSE	
LPSTREAM*	ppstm	
HRESULT CoUnmarshalInterface ()		
LPSTREAM	pStm	*IStream* Object
REFIID	{00000001-0000-0000-C000000000000046}	-
LPVOID*	ppv	-

The 0x011000 IPC request is serviced only in the OUTLOOK.EXE broker scenario. This is presumably requested by sandbox to clean up after loading the MS Office Protected-View Previewer as a COM.

3.3.4 Message 0x061000

The 0x061000 message is a *PostMessageW()* request for broker, and contains the API parameters in this format:

Message Body for 0x061000 Request		
Size	**Field**	**Comment**
UINT32	ui32ViewID	-
DWORD	dwMsg	Any of WM_KEYDOWN, WM_KEYUP, WM_CHAR, WM_SYSKEYDOWN, WM_SYSKEYUP or WM_SYSCHAR
WPARAM	wParam	According to *dwMsg*
LPARAM	lParam	According to *dwMsg*

In general, a user can disable the Protected-View mode with "Enable Editing" either from FILE tab or preview/sandbox window, both of which are prevented by the broker when servicing 0x061000 requests. In the first scenario, the broker ensures that the preview window is in focus by checking the *lpViewFile->lpSameBrokerApp->hOPHParentWnd* windows hierarchy before dispatching the requested *PostMessageW()*. So while sandbox can navigate to FILE with the "ALT-F" keyboard-inputs, broker will ignore subsequent 0x061000 requests thereafter.

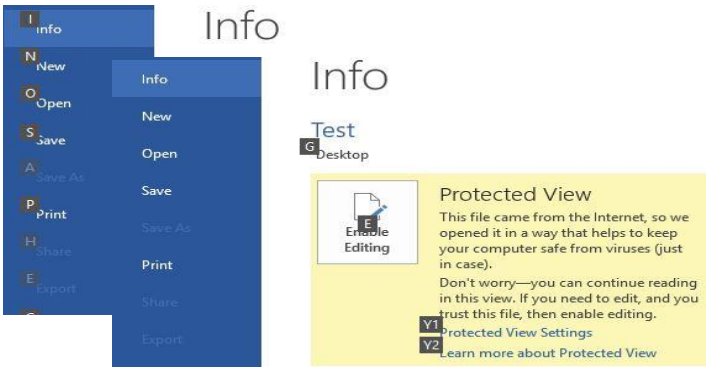


Figure 9: At FILE, preview window is not in focus. Hence these 0x061000 messages of these keyboard-inputs are ignored

In the second scenario, the broker prevents disabling the Protected-View mode with dispatching *PostMessageW()* for simulated mouse-clicks (BM_LBUTTONDOWN) from sandbox window by limiting the allowable set of *dwMsg* values.

3.3.5 Message 0x091000

In Protected-View mode, embedded URLs are still accessible to the user. The 0x091000 message is a *HlinkNavigateToStringReference()* request to the URL, subjected to user permission. The format for the 0x091000 request looks like:

Message Body for 0x091000 Request		
Size	**Field**	**Comment**
UINT32	ui32ViewID	-
LPWSTR	pwzTarget	Parameter in *HlinkNavigateToStringReference()*
LPWSTR	pwzLocation	Parameter in *HlinkNavigateToStringReference()*
DWORD	grfHLNF	Parameter in *HlinkNavigateToStringReference()*

UINT32	ui32Unknown	-
HWND	hWndParent	*CreateWindowExW()*, for user-permission prompt window
WCHAR[]	wzTarget	Parameter in *HlinkNavigateToStringReference()*
WCHAR[]	wzLocation	Parameter in *HlinkNavigateToStringReference()*

The user permission for URL-navigation is prompted per-file session, as tracked by *lpViewFile->ui32SessionEnableHyperlinks*. Once allowed, *lpViewFile->ui32SessionEnableHyperlinks* is set to 1 and subsequent URL prompts for the same file will be suppressed.

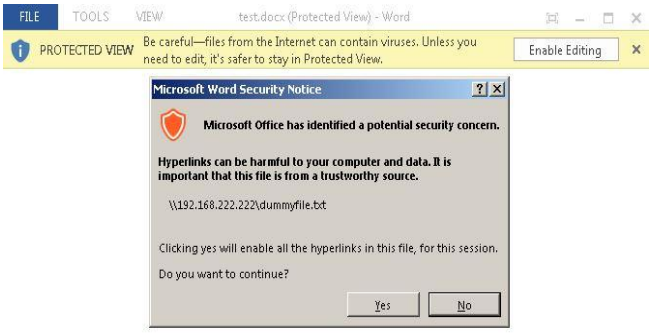


Figure 10: The user-prompt to navigate URL

3.3.6 Message 0x0A1000

The 0x0A1000 message is a request for broker to create the "Enter Password" messagebox for a password-protected file, of which the input is returned to sandbox, as shown by request-response message formats here.

Message Body for 0x0A1000 Request		
Size	**Field**	**Comment**
UINT32	ui32ViewID	-
Message Body for 0x0A1000 Response		
LPWSTR	lpwPassword	Pointer to *wzPassword*
UINT16	ui16PasswordSize	Wide-char length of *wzPassword*
WCHAR[]	wzPassword	-

It is serviced only when *SameBrokerApp* object is not instantiated, as the case when broker is OUTLOOK.EXE. This is because OUTLOOK.EXE does not parse WINWORD files itself. So when such attachment is opened, OUTLOOK.EXE will start the sandbox immediately to handle it. Sandbox will then realizes that a password is required and therefore sends the 0x0A1000 request for OUTLOOK.EXE to prompt the user.

This is unlike the case where the WINWORD.EXE broker prompts for the password before handing it over to the sandbox. And the WINWORD.EXE broker can avoid prompting for password again by checking for *SameBrokerApp* instantiation.

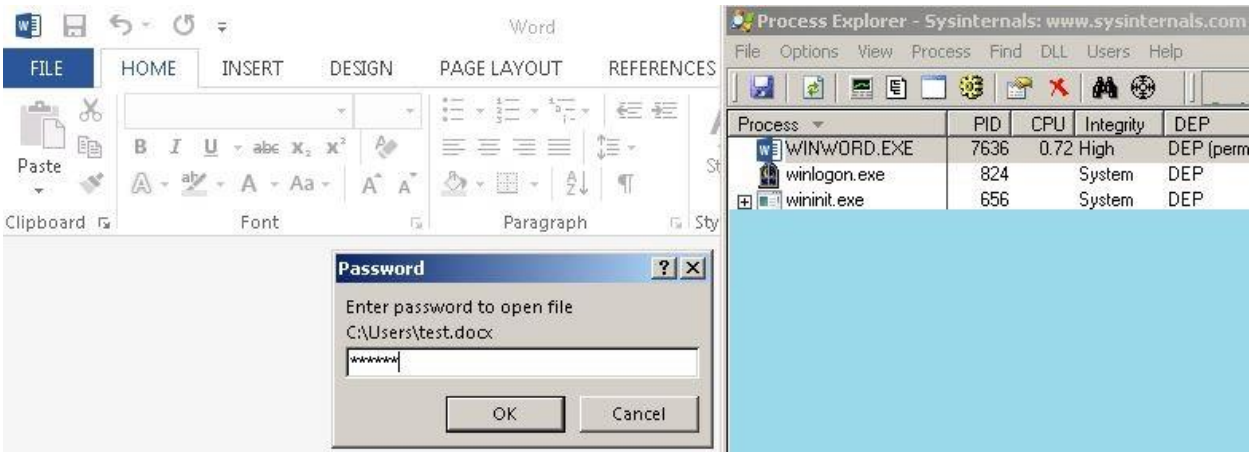


Figure 11: In WINWORD.EXE, the password is prompted only once, before Protected-View mode starts

In the OUTLOOK.EXE case, because it does not monitor the situation, OUTLOOK.EXE will unconditionally create a new worker object for the password prompt for every 0x0A1000 request, even after the file is opened.

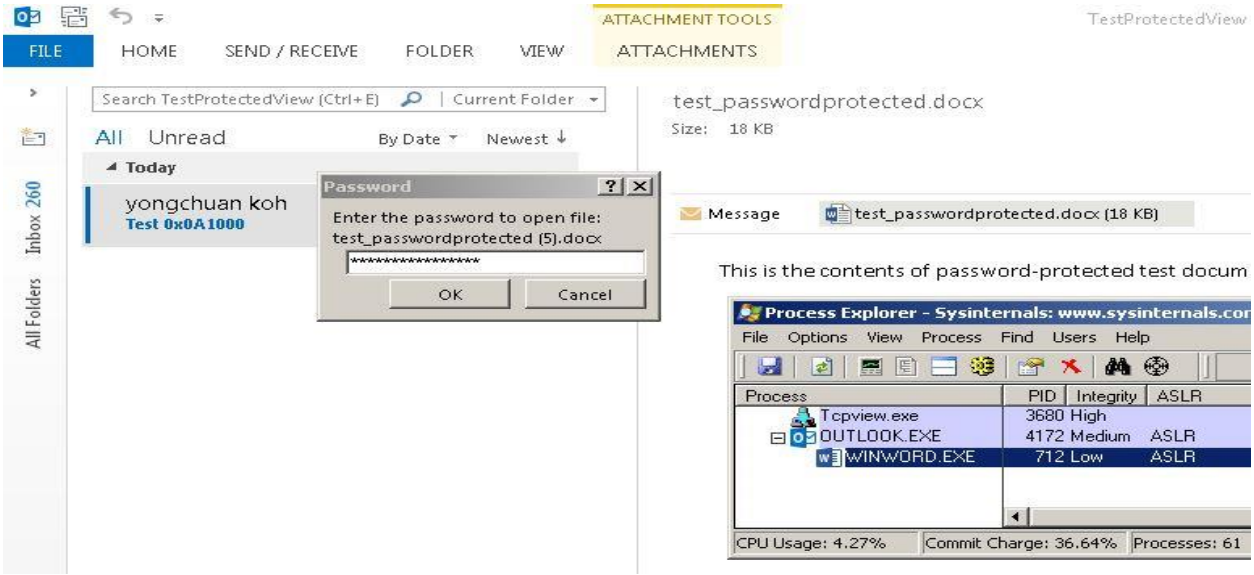


Figure 12: In OUTLOOK.EXE, the password-prompt is always created on every 0x0A1000 request, even if file is already opened in Protected-View mode shown in background. The password is sent back as the IPC response.

3.3.7 Message 0x0C1000

In a typical Windows Error Reporting (WER) scenario, the application will launch DWWIN.EXE, and passes to it the exception information which will be relayed to a WER server as part of error reporting. However, the Protected-View sandbox cannot launch DWWIN.EXE directly because its *hSandboxJob*'s JOBOBJECT_BASIC_LIMIT_INFORMATION.ActiveProcessLimit is restricted to 1. Therefore, the sandbox has to send the 0x0C1000 request for broker to start DWWIN.EXE on its behalf, in the shared-memory

mode (13). In this case, the exception information is sent from sandbox to broker, then from broker to DWWIN.EXE.

Message Body for 0x0C1000 Request		
Size	**Field**	**Comment**
HANDLE	hSandboxSharedMem	-
HANDLE	hEventBrokerIsDone	Broker sets this event after DW20.EXE exits and before it terminates sandbox
LPWSTR	lpwAdditionalWerFileName	Pointer to *wzAdditionalWerFileName*, or Null
UINT16	ui16AdditionalWerFileNameSize	-
WCHAR[]	wzAdditionalWerFileName	An additional file to be submitted to WER server, to be created by Protected-View sandbox

In the IPC request format, the *hSandboxSharedMem* is a duplicated handle for the exception information that sandbox wants to send to broker. Upon receiving the request, the broker will similarly duplicates the *hBrokerShareMem* handle to DWWIN.EXE.

The format for *hBrokerShareMem* shared-memory is not documented publicly, but through reverse-engineering, it should be similar to:

Possible Format of *hBrokerShareMem* Shared Memory		
Size	**Field**	**Comment**
UINT32	0x00009C9C	Size of *hBrokerSharedMem* memory
UINT32	0x00020000	Constant, MSO.00F25F79
UINT32	ui32ProtectedViewPID	Sandbox Process-ID
UINT32	ui32ProtectedViewTID	TID of faulting thread in sandbox. Copied from *hSandboxSharedMem* offset 0x0C.
UINT32	uiProtectedViewEIP	EIP of faulting instruction in sandbox. Copied from *hSandboxSharedMem* offset 0x10.
LPVOID	lpProtectedViewPEP	Exception pointers in sandbox. Copied from *hSandboxSharedMem* offset 0x14.
HANDLE	hEventDone	-
HANDLE	hEventNotifyDone	-
HANDLE	hEventAlive	-
HANDLE	hMutex	-
HANDLE	hSandboxProcess	Sandbox process handle
UINT32	ui32AdditonalFileName	1 if *lpwAddtionalWerFileName*, Null if otherwise
UINT32	0x0000000C	Constant, MSO.00F26008
UINT32	0x00000000	Constant, MSO.00F2600F
UINT32	0x00000025	Constant, MSO.00F26012
UINT32	0x00000002	Constant, MSO.00F26019
UINT32	ui32LCID	Locale ID used by Microsoft Office
UINT32	0x00000001	Constant, MSO.0119CD8E
UINT32	0x00000000	Constant, MSO.0119CCB0
UINT32	0x00000011	Constant, MSO.0119CD94

WCHAR[0x104]	wzMSOfficeVersion	"Microsoft Office 15"
WCHAR[0x104]	wzFullPathApp	Full-path of WINWORD.exe
UCHAR[0x104]	uchUnknown	-
CHAR[0x38]	szUnknown	-
CHAR[0x104]	szSkulcid	“skulcid=%d”, MSO.15E7A70
UCHAR[0x104]	uchUnknown	-
WCHAR[0x104]	wzModulesList	List of modules loaded in sandbox, separated by Null
WCHAR[0x400]	wzWerSubmitFilesList	List of files to submit to WER with *CWatsonReport::AddFilesToReport()*, separated by ‘|’: • **Sandbox-directory + *wzAdditionalWerFileName*** • %Temp% + “winword.exe.OsrHost.dmp.dat” • %Temp% + “winword.exe.OsrHost.cvr.dat”
UCHAR[0x1000]	uchUnknown	-
WCHAR[0x38]	wzAppName	“Microsoft Word (Protected View)”
UCHAR[0x6254]	uchUnknown	-
UINT32	ui32CrashParamFlag	If Null, next 11 fields are ignored. Copied from *hSandboxSharedMem* offset 0x8470
WCHAR[0xFF]	wzEventType	*pwzEventType* in *WerReportCreate()* Copied from *hSandboxSharedMem* offset 0x8474
WCHAR[0xFF]	wzParam0	*pwzValue* for WER_P0 in *WerReportSetParameter()* Copied from *hSandboxSharedMem* offset 0x8672
WCHAR[0xFF]	wzParam1	*pwzValue* for WER_P1 in *WerReportSetParameter()* Copied from *hSandboxSharedMem* offset 0x8870
WCHAR[0xFF]	wzParam2	*pwzValue* for WER_P2 in *WerReportSetParameter()* Copied from *hSandboxSharedMem* offset 0x8A6E
WCHAR[0xFF]	wzParam3	*pwzValue* for WER_P3 in *WerReportSetParameter()* Copied from *hSandboxSharedMem* offset 0x8C6C
WCHAR[0xFF]	wzParam4	*pwzValue* for WER_P4 in *WerReportSetParameter()* Copied from *hSandboxSharedMem* offset 0x8E6A
WCHAR[0xFF]	wzParam5	*pwzValue* for WER_P5 in *WerReportSetParameter()* Copied from *hSandboxSharedMem* offset 0x9068
WCHAR[0xFF]	wzParam6	*pwzValue* for WER_P6 in *WerReportSetParameter()* Copied from *hSandboxSharedMem* offset 0x9266
WCHAR[0xFF]	wzParam7	*pwzValue* for WER_P7 in *WerReportSetParameter()* Copied from *hSandboxSharedMem* offset 0x9464
WCHAR[0xFF]	wzParam8	*pwzValue* for WER_P8 in *WerReportSetParameter()* Copied from *hSandboxSharedMem* offset 0x9662
WCHAR[0xFF]	wzParam9	*pwzValue* for WER_P9 in *WerReportSetParameter()* Copied from *hSandboxSharedMem* offset 0x9860
UINT16	ui16Pad	Padding when copied from *hSandboxSharedMem*
UINT32[0x9]	ui32Unknown	-
UINT32	ui32DW20Status	Status result written by DW20.EXE
UCHAR[0x214]	uchUnknown	-

The first issue with the 0x0C1000 IPC request is a Read Access-Violation when the broker copies a fixed-size of 0x15F0 bytes from *hSandboxSharedMem* (offset 0x8470) to *hBrokerSharedMem* (offset 0x8470) for the *ui32CrashParamFlag* to *ui16Pad* fields, without checking if *hSandboxSharedMem* has the sufficient size to read from.

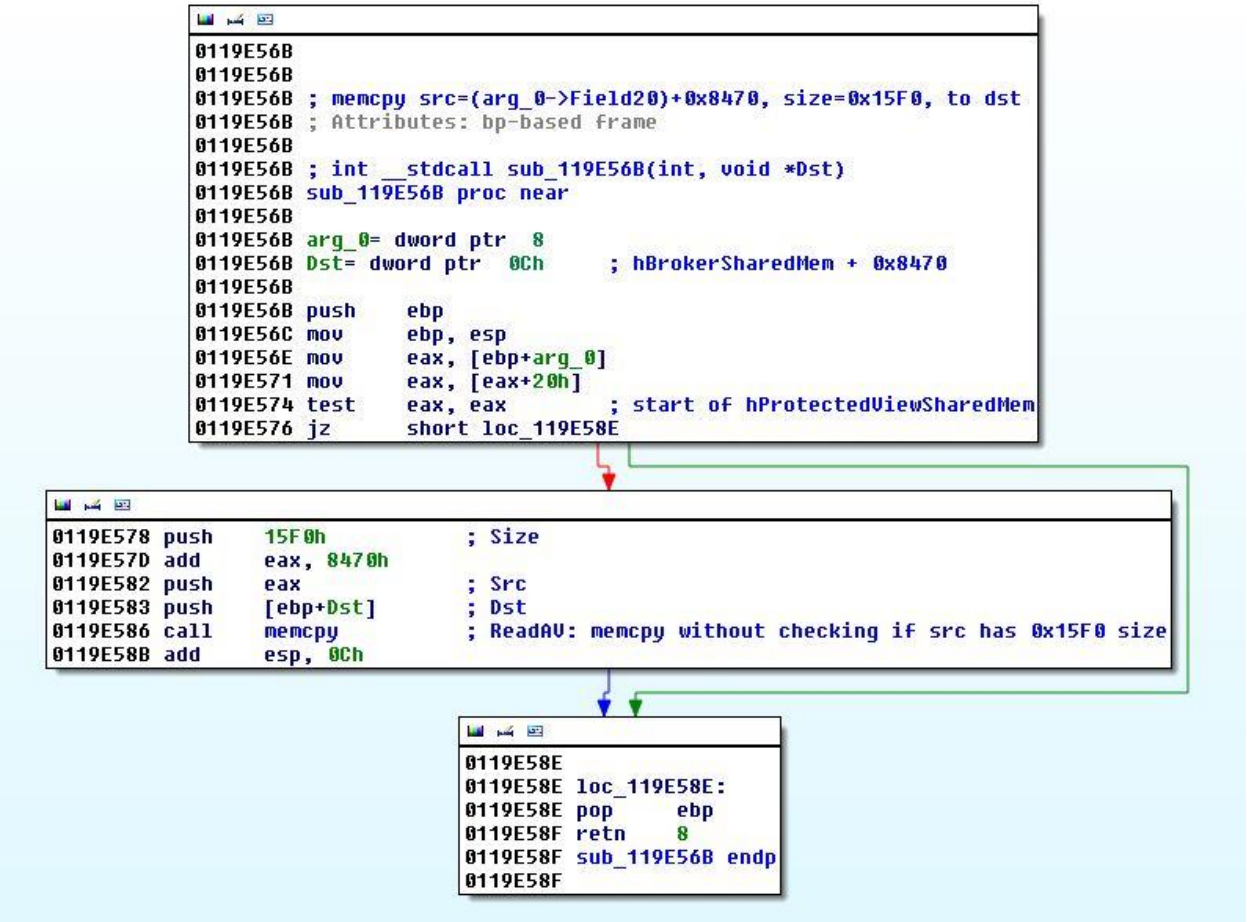


Figure 13: Read-AV when broker copies hSandboxSharedMem+0x8470 to hBrokerSharedMem+0x8470, size 0x15F0

The second issue is a directory-traversal out of the sandbox container with the *wzAdditionalWerFileName* field.

At the onset of WER, the user is given 3 options; "*Check online for a solution later and close the program*", "*Debug the program*" and "*Close the program*". With the first option, DWWIN.EXE will send an overview of this error to the WER server containing the operation system version, application version and exception information in an XML format. This is known as the "Level 1 Error Reporting Data". If WER server determines this error to be unique, it will request for additional files to be .cab compressed and uploaded as "Level 2 Error Reporting Data" (14). Finally DWWIN.EXE deletes these files by setting FILE_DISPOSITION_INFORMATION.DeleteFile to TRUE.

The sandbox can upload a chosen file to upload with the *wzAdditionalWerFileName* field in the IPC request, for which DWWIN.EXE will add to .cab with *CWatsonReport::AddFilesToReport()*. Since this is a file to be added by sandbox, it should reside in the sandbox container. The broker ensures this by:

1. Checking that there is no backslash character in the *wzAdditionalWerFileName* field.
2. Pre-pending the sandbox-directory to the *wzAdditionalWerFileName* field.

The purpose for checking backslash wide-characters is to prevent path traversal out of the sandbox directory.

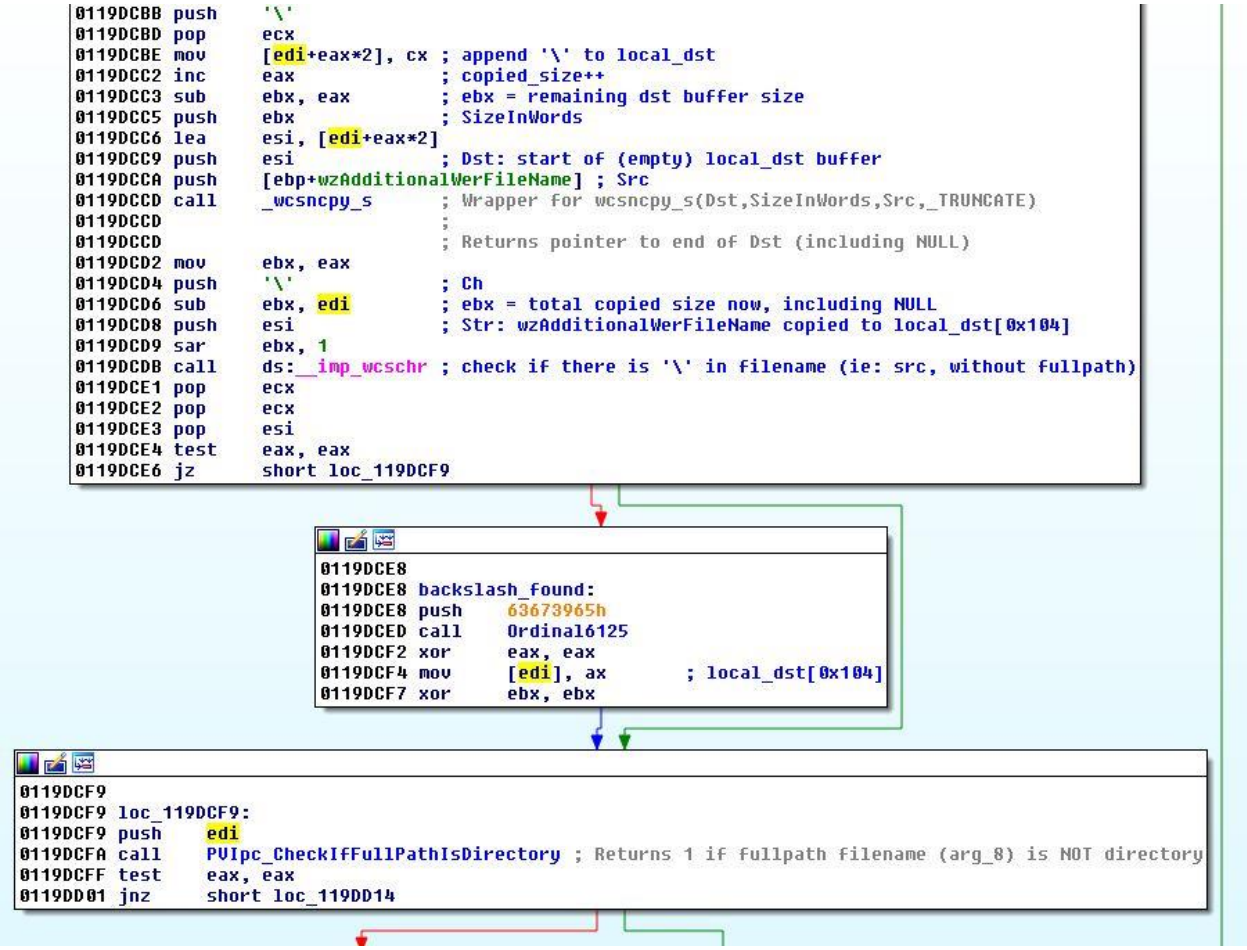


Figure 14: Checks only for "\" backslash in wzAdditionalWerFileName

However, because "*…File I/O functions in the Windows API convert "/" to "\" as part of converting the name to an NT-style name…*" (15), the sandbox can use forward-slash characters in *wzAdditionalWerFileName* to bypass the file-name check and traverse out of the sandbox directory, as shown below.

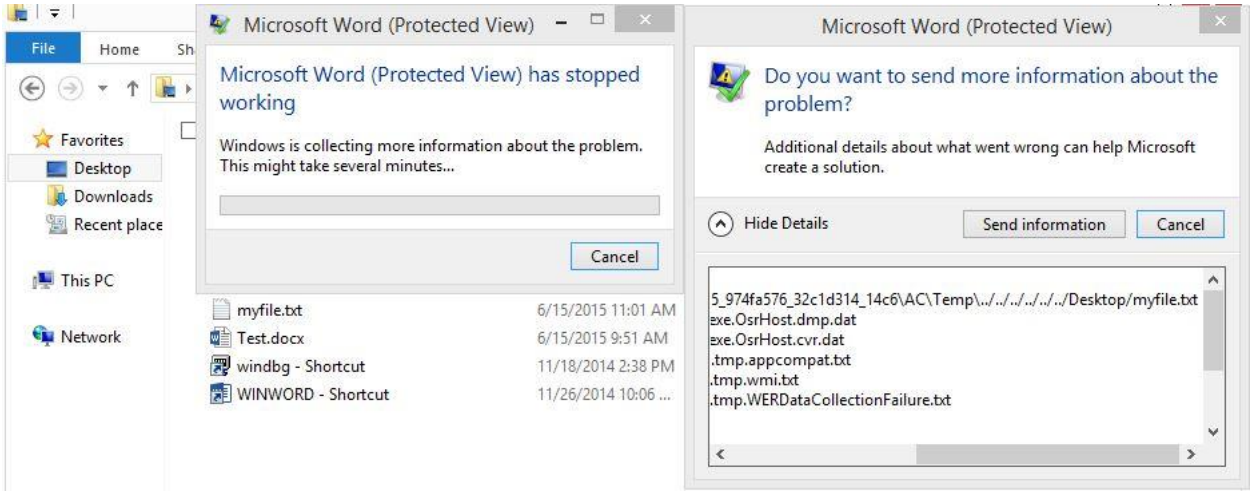


Figure 15: Uploading and deleting .txt file outside the sandbox directory with the wzAdditionalWerFileName field

Since the *wzAdditionalWerFileName* file is also deleted by DWWIN.EXE, an attacker could:

1. Steal arbitrary file if they had compromise the *WER* server ("*Watson.microsoft.com*" by default, or reconfigured to an enterprise *WER* server), and/or
2. Delete arbitrary file in the system, in the context of USER.

This issue was reported to Microsoft and the response was *"…We've completed our investigation on this issue but this doesn't meet our current bug bar for servicing. We are looking at incorporating this fix as part of our future security plans …"*

3.3.8 Message 0x0F1000

The 0x0F1000 IPC request simulates the "Microsoft Help" action in either of these two modes:

1. Running *CreateProcessW(""<FullPath-CLVIEW.EXE>" "WINWORD" "Word"")* or,
2. Navigating to the *Microsoft Office Developer Help* website; *HlinkNavigateToStringReference("http://odc.officeapps.live.com/odc/help/clientdeveloper")*.

Message Body for 0x0F1000 Request		
Size	**Field**	**Comment**
UINT32	ui32ViewID	-
UINT32	ui32MessageID	0x465 or 0x469
LPWSTR	lpwData	Pointer to *wzData*, or Null
LPWSTR	lpwHelpID	Pointer to *wzHelpID*, or Null
UINT16	ui16DataSize	-
WCHAR[0x104]	wzData	"DEV", "SHAPESHEET", or others
UINT16	ui16HelpIDSize	-
WCHAR[0x104]	wzHelpID	"NoHelp", or others

The mode of the "Microsoft Help" action is dependent on the IPC message fields, shown by this table:

	ui32MessageID	*lpwData*	*wzData*	*lpwHelpID*	*wzHelpID*
Mode 1a	0x465	Null	NA	NA	NA
	0x465	Non-Null	Null	NA	NA
	• Runs CLVIEW.EXE • Broker will *SendMessage()* to CLVIEW.EXE window with COPYDATASTRUCT = {0x465, 0x208, Data=Null}				
Mode 1b	0x465	Non-Null	Any	NA	NA
	• Runs CLVIEW.EXE • Broker will *SendMessage*() to CLVIEW.EXE window with COPYDATASTRUCT = {0x465, 0x208, Data=<*cwData*>}				
Mode 2a	0x465	Non-Null	"DEV" or "SHAPESHEET"	NA	NA
	• Navigates to help site with *HlinkNavigateToStringReference()*				
-	0x469	NA	NA	Null	NA
	• **Read Access-Violation (see below)**				
Mode 1c	0x469	NA	NA	Non-Null	Null
	• Same as *ui32MessageID*=0x465, depending on *lpwData* and *wzData*				
-	0x469	NA	NA	Non-Null	"NoHelp"
	• Does nothing				
Mode 1d	0x469	Null	NA	Non-Null	Any
	0x469	Non-Null	Null	Non-Null	Any
	• Runs CLVIEW.EXE • Broker will *SendMessage()* to CLVIEW.EXE window with COPYDATASTRUCT = {0x469, 0x410, Data=Null}				
Mode 1e	0x469	Non-Null	Any	Non-Null	Any
	• Runs CLVIEW.EXE • Broker will *SendMessage()* to CLVIEW.EXE window with COPYDATASTRUCT = {0x469, 0x410, Data=<*cwData*>}				
Mode 2b	0x469	Non-Null	"DEV" or "SHAPESHEET"	Non-Null	Any
	• Navigates to help site with *HlinkNavigateToStringReference()* • pwzTarget = "*…helpid=wzHelpID*"				

The Read Access-Violation occurs because of the Null *lpwHelpID* which broker did not check before comparing *wzHelpID* with "NoHelp".

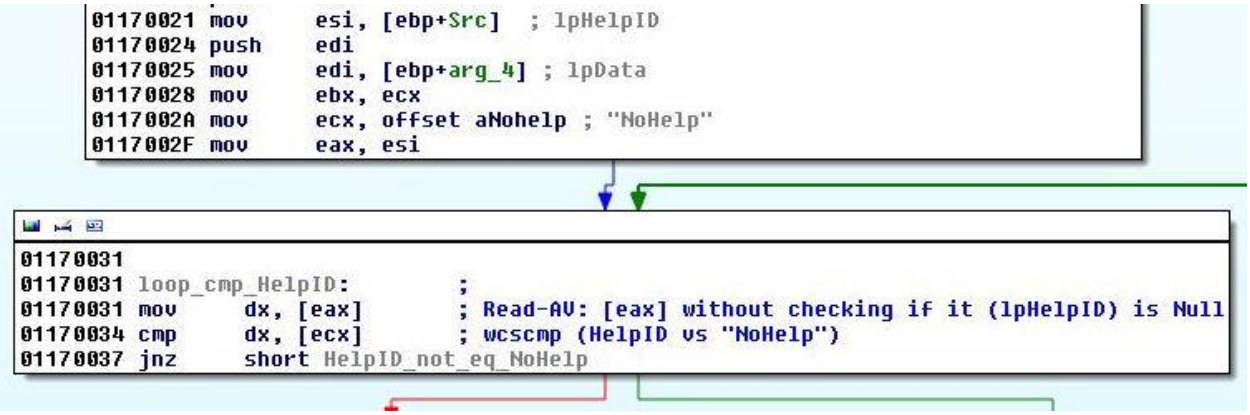


Figure 16: Read-AV when broker compares Null lpwHelpID with "NoHelp"

3.3.9 Message 0x141000

On receiving the 0x141000 request, the broker duplicates and returns the "*Global\FntCache<wzFontMappingObjectName>*" mapped file handle with DUPLICATE_SAME_ACCESS, or Null if it is not found.

Message Body for 0x141000 Request		
Size	**Field**	**Comment**
LPWSTR	lpwFontMappingObjectName	Pointer to *wzFontMappingObjectName*, or Null
UINT16	ui16FontMappingObjectNameSize	-
WCHAR[0x0F5]	wzFontMappingObjectName	-
0x141000 IPC Response		
HANDLE	hDuplicatedFontMappingObject	-

3.4 Comments

The Protected-View sandbox implemented only 38 IPC messages, significantly fewer than other sandboxes (Adobe Reader XI had 260 (16) cross-call functions). In these 38 messages, some were used for elevation purposes such as duplicating handles (0x001000 and 0x141000), sandbox GUI interactions (0x061000, 0x091000 and 0x0F1000), and to update or retrieve the IPC information from the internal objects.

Although there are guarded checks for memory corruption, issues still existed in these 38 IPC messages. The 0x0C1000 message allows an attacker to escape the sandbox restrictions to upload and delete arbitrary files. Both 0x0C1000 and 0x0F1000 messages have non-exploitable Read Access-Violations.

4. Microsoft Office 2016

Since the preview version of MS Office 2016 has been released around the time this research was completed, it will be interesting to study any changes that may be made. This can also be an indication of things to expect for where the Protected-View sandbox is heading.

4.1 Sandbox Restrictions

As the sandbox container boundary is effectively defined by capabilities, this should be one of the first places to check. As it turns out, no new capability is added. With the same *Capability-SID* (*S-1-15-3-2929230137-1657469040)*, another iteration of checking the accessible system resources is done. Again there are no new additions.

4.2 Sandbox Code

Identifying changes to the Protected-View sandbox will have been straightforward with binary-diffing tools such as *Diaphora*, if not for these two factors:

1. In Microsoft Office 2016, common routines are now moved to individual modules (eg: Mso20win32client.dll, Mso30win32client.dll, Mso40win32client.dll, Mso99win32client.dll and Mso40UIwin32client.dll)
2. In MS Office 2016, new security assertions introduced in Windows 8 (17) are used

As a result, the new compiled code looks different even though the routines may be semantically similar. For example, the two snapshots below shows comparison of the same routine to create restricted tokens between MS Office 2013 and MS Office 2016, and the new security assertion that changes the control-flow graph.

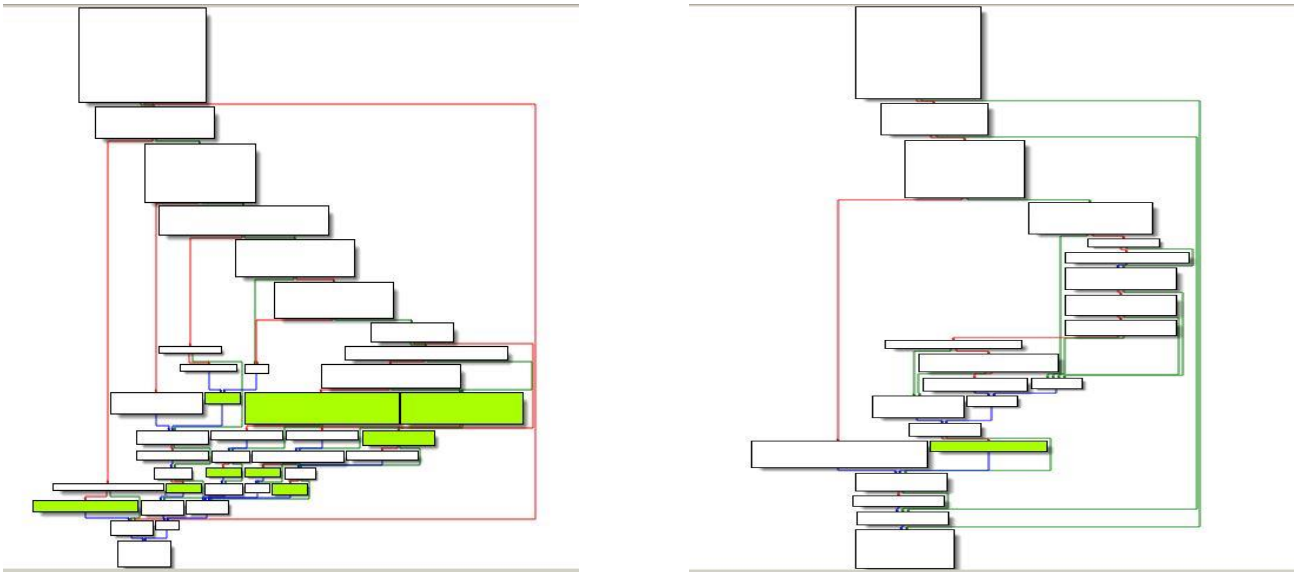


Figure 17: Green basic-blocks are calls to _MsoFreePv@4() in Microsoft Office 2013 (left), and its corresponding replacement calls to Mso20 Win32Client_934() in Microsoft Office 2016 (right)

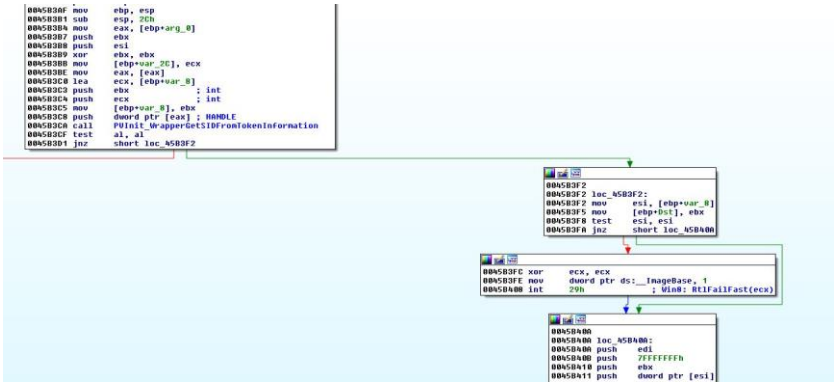


Figure 18: New security assertions in Microsoft Office 2016

Nonetheless, a manual audit to the binaries found a few differences.

4.2.1 Sandbox Initialization

In Microsoft Office 2016, there is a new basic-block to start the sandbox process as the current user (*CreateProcessW()* instead of *CreateProcessAsUserW()*). However it looks like dead-code currently because the conditional variable [EBX+81h] is not referenced in any parts of the binary.

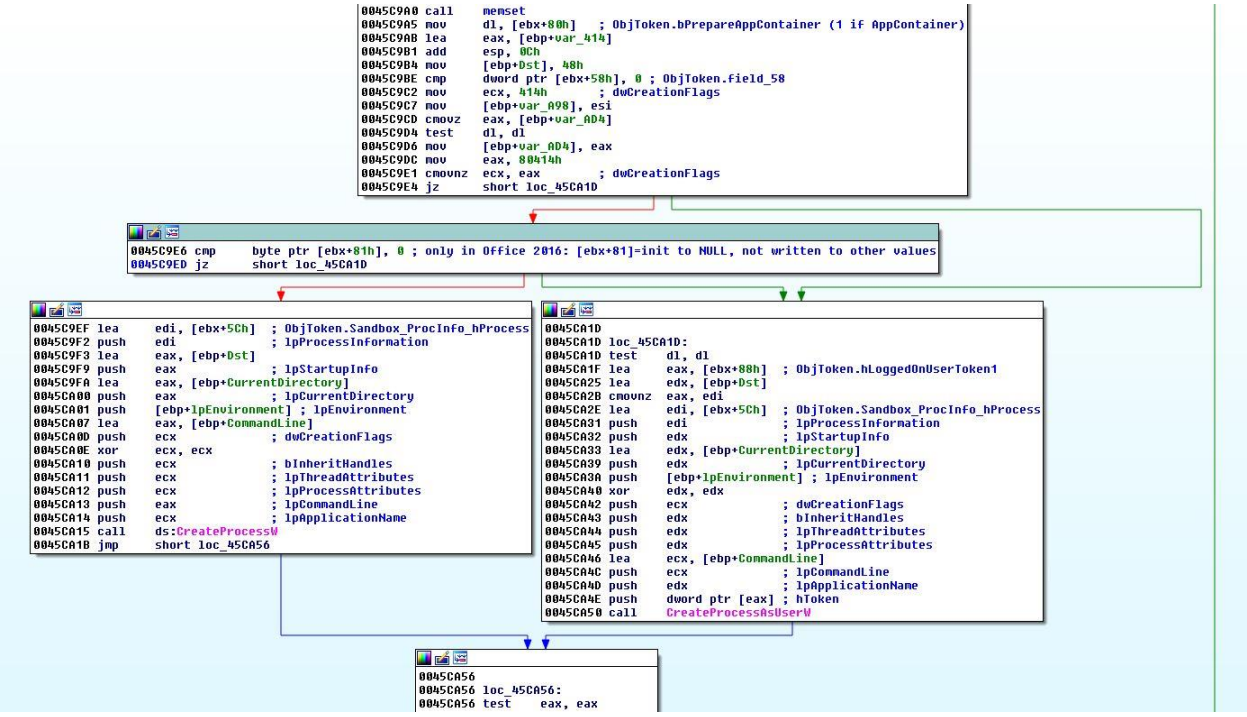


Figure 19: Additional left basic-block for CreateProcessW()

4.2.2 Message 0x0C1000

The directory-traversal issue in the 0x0C1000 message is still found in MS Office 2016, even though 2.5 months has passed since the time of MS response (that they will "*…incorporating this fix as part of our future security plans …*") to the release of MS Office 2016 Preview.

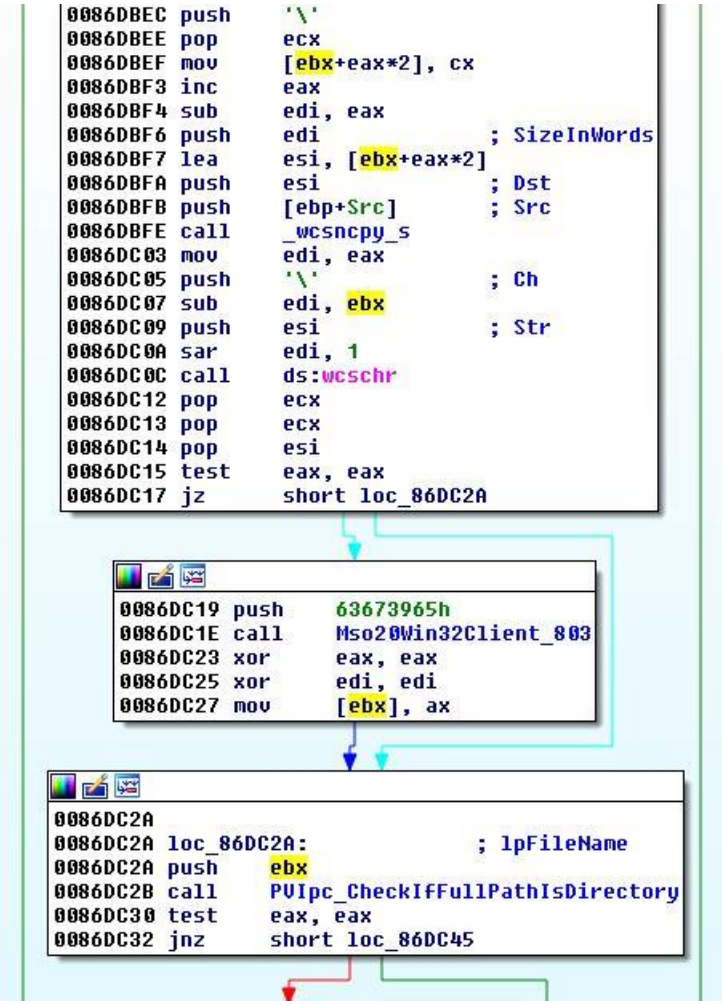


Figure 20: Directory-traversal issue still found in Microsoft Office 2016

This issue is probably one of the least priority to fix for MS because of the prerequisites of a remote-code-execution in the sandbox, then also user interactions to send "Level 2 Error Reporting Data".

4.2.3 Message 0x161000

The 0x161000 IPC request is a new message that is serviced only in the OUTLOOK.EXE, and is used to protect or unprotect the "OPH Previewer Window" sandbox window for RMS-protected document.

Message Body for 0x161000 Request		
Size	**Field**	**Comment**
UINT32	ui32ViewID	-
UINT32	hUnprotectWnd	Window handle to unprotect for.
UINT32	ui32Boolean	If 0, unprotects window with *IpcUnprotectWindow()* or *_IpcUnprotectWindowNoDRM()*. If 1, protects window with *IpcProtectWindow()* or *_IpcProtectWindowNoDRM()*

The *ui32ViewID* is used to iterate for the corresponding *lpViewFile* (and subsequently *lpViewFile->lpField_00->hOPHPreviewerWnd*), and *hUnprotectWnd* is used to directly find the matching *lpViewFile->lpField_00->hOPHPreviewerWnd*.

4.2.4 Message 0x031100

The 0x031100 IPC request has a larger message size (from 0x1E8 to 0x1F4) although there is no change in handling this message.

4.3 Comments

Microsoft seems to be quite happy with the Protected-View sandbox, and have not made drastic modifications to it in MS Office 2016. It is still bounded by similar container restrictions, and only minimal changes in the IPC mechanism.

5. Conclusion

In summary, the Protected-View sandbox implementation is unique from other sandboxes because it offers the user a “preview” of these files before they decide if Protected-View is to be disabled. The result is a very simplified sandbox architecture that needs to support only a small subset of original functionalities, unlike most sandboxes.

This approach greatly reduces the attack surfaces to escape the Protected-View sandbox. In particular, there are only 38 IPC messages to audit. In handling these messages, appropriate length checks, limiting message sizes, and using _TRUNCATE flag whenever possible, have eliminated basic memory corruption issues. However, subtle issues such as the directory-traversal out by 0x0C1000 request are still found.

In the future, attack surfaces not covered in this paper can be explored. These include exploiting the kernel, global shared resources, or any newly introduced IPC messages such as the 0x161000 message in MS Office 2016.

In reality, it may also be worthwhile for attackers to derive clever social engineering tricks to entice the users to disable the Protected-View mode.

6. Bibliography

1. **Wikipedia.** *"Sandbox (computer security)"*. [Online] https://en.wikipedia.org/wiki/Sandbox_(computer_security).

2. **Office.** "What is Protected View?". [Online] Microsoft Corporation. https://support.office.com/en-us/article/What-is-Protected-View-d6f09ac7-e6b9-4495-8e43-2bbcdbcb6653?ui=en-US&rs=en-US&ad=US.

3. **Dowd, Mark.** "The Chrome Sandbox". [Online] Azimuth Security. http://blog.azimuthsecurity.com/2010/05/chrome-sandbox-part-1-of-3-overview.html.

4. **Yason, Mark Vincent.** "DIVING INTO IE 10'S ENHANCED PROTECTED MODE SANDBOX". [Online] https://www.blackhat.com/docs/asia-14/materials/Yason/WP-Asia-14-Yason-Diving-Into-IE10s-Enhanced-Protected-Mode-Sandbox.pdf.

5. **Forshaw, James.** "IE11 Sandbox Escapes". [Online] https://github.com/tyranid/IE11SandboxEscapes.

6. **Sabanal, Paul and Yason, Mark Vincent.** "PLAYING IN THE READER X SANDBOX". [Online] https://media.blackhat.com/bh-us-11/Sabanal/BH_US_11_SabanalYason_Readerx_WP.pdf.

7. **David Middlehurst, James Loureiro.** *"Why Bother Assessing Popular Software?"*. [Online] https://labs.mwrinfosecurity.com/publications/2015/06/05/why-bother-assessing-popular-software/.

8. **Ollie.** "Windows 8 App Container Security Notes". [Online] Recx Ltd. http://recxltd.blogspot.com/2012/03/windows-8-app-container-security-notes.html.

9. **MSDN.** "Capability SID Constants". [Online] Microsoft Corporation. https://msdn.microsoft.com/en-us/library/windows/desktop/hh448474(v=vs.85).aspx.

10. **Keetch, Tom.** "Practical Sandboxing on the Windows Platform". [Online] https://www.hackinparis.com/slides/hip2k11/12-EscapingWindowsSandboxes.pdf.

11. **MSDN.** *"JOBOBJECT_BASIC_UI_RESTRICTIONS structure"*. [Online] https://msdn.microsoft.com/en-us/library/windows/desktop/ms684152(v=vs.85).aspx.

12. **Wikipedia.** "OBJREF". [Online] http://en.wikipedia.org/wiki/OBJREF.

13. **MSDN.** "How to: Configure Microsoft Error Reporting". [Online] Microsoft Corporation. https://msdn.microsoft.com/en-us/library/office/bb219076(v=office.12).aspx.

14. **MSDN**. *[MS-CER2]: Corporate Error Reporting V.2 Protocol*. [Online] https://msdn.microsoft.com/en-us/library/dd942170.aspx.

15. **MSDN**. "Naming Files, Paths, and Namespaces". [Online] Microsoft Corporation. https://msdn.microsoft.com/en-us/library/windows/desktop/aa365247(v=vs.85).aspx.

16. **Vreugdenhil, Peter.** "ADOBE SANDBOX: WHEN THE BROKER IS BROKEN". [Online] https://cansecwest.com/slides/2013/Adobe%20Sandbox.pdf.

17. **Ionescu, Alex.** "New Security Assertions in Windows 8". [Online] http://www.alex-ionescu.com/?p=69.

7. Appendix A: IPC Format of Remaining Messages

Message 0x021000

For this message, the broker returns the *ui32ViewID* for the *wzFileName* untrusted file.

Message Body for 0x021000 Request		
LPWSTR	lpwFileName	Pointer to *wzFileName*. or Null
UINT16	ui16FileNameSize	-
WCHAR[]	wzFileName	-
Message Body for 0x021000 Response		
UINT32	ui32ViewID	-

Message 0x031000

For this message, the broker returns the full-path of the temporary file that is created in the sandbox directory.

Message Body for 0x031000 Request		
LPWSTR	lpwFileName	-
UINT16	ui16FileNameSize	-
WCHAR[]	wzFileName	-
Message Body for 0x031000 Response		
LPWSTR	lpwTemporaryFileName	Pointer to *wzTemporaryFileName*, or Null
UINT16	ui16TemporaryFileNameSize	-
WCHAR[]	wzTemporaryFileName	Temporary file created in sandbox directory. Eg: *"%UserProfile%\AppData\<sandbox-dir>\D6D48720.docx"*

Message 0x041000

This is a request to re-parent the *hChildWnd* window handle:

Message Body for 0x041000 Request		
UINT32	ui32ViewID	-
HWND	hChildWnd	To re-parent to *lpViewFile->hOPHWnd*

The *hChildWnd* is the window handle for the Protected-View sandbox, and will be set as the child of the window with the class "OPH Previewer Window", reference by *ViewFile->hOPHWnd*. It is not possible to set an arbitrary window as the child because the broker checks the Process-ID of *hChildWnd* window against the corresponding *ViewFile* object before re-parenting.

Message 0x051000

This message causes the broker to parse the compressed *wzTmpFileName* file in the sandbox directory:

Message Body for 0x051000 Request		
UINT32	ui32ViewID	-
LPWSTR	lpwTmpFileName	Pointer to *wzTmpFileName*, or Null
UINT16	us16TmpFileNameSize	-
WCHAR[]	wzTmpFileName	Parsed by *MsoHrGetFileByteStream*() and *MsoHrOpenPackage*()

Message 0x071000

The 0x071000 request does not have a message body. The broker returns the result of *GetAsyncKeyState*(*ui32VirtualKey*), where *ui32VirtualKey* is read from the message header and is any of these virtual key codes:

ui32VirtualKey		
0x01	VK_LBUTTON	Left mouse button
0x02	VK_RBUTTON	Right mouse button
0x03	VK_CANCEL	Control-break processing
0x04	VK_MBUTTON	Middle mouse button (three-button mouse)
0x05	VK_XBUTTON1	X1 mouse button
0x06	VK_XBUTTON2	X2 mouse button
0x10	VK_SHIFT	SHIFT key
0x11	VK_CONTROL	CTRL key
0x12	VK_MENU	ALT key
0x1B	VK_ESCAPE	ESC key

Message 0x081000

The broker returns the Digital Rights Management COM stream for the *ui32ViewID* file.

Message Body for 0x081000 Request		
UINT32	ui32ViewID	
Message Body for 0x081000 Response		
LPVOID	lpDRMStreamMemory	Pointer to *uchDRMStreamMemory*, or Null
UINT32	ui32DRMStreamMemorySize	-
UINT32	*lpSameBrokerApp->Field_98*	-
UCHAR[]	uchDRMStreamMemory	*lpViewFile->lpSameBrokerApp->lpDRMStream*

Message 0x0B1000

The 0x0B1000 message updates the broker on the task that the sandbox is executing.

Message Body for 0x0B1000 Request		
UINT32	ui32ViewID	-
UINT32	ui32TaskID	See below
UINT32	ui32TaskAction	See below

LPWSTR	lpwTaskName	Pointer to *wzTaskName*, or Null
LPWSTR	lpwTaskDescription	Pointer to *wzTaskDescription*, or Null
UINT32	ui32TaskCurrentProgress	See below
UINT32	ui32TaskMaxProgress	See below
UINT32	ui32Unknown	-
UINT16	ui16TaskNameSize	-
WCHAR[0x40]	wzTaskName	-
UINT16	ui16TaskDescriptionSize	-
WCHAR[0x100]	wzTaskDescription	-

The six possible values for *ui32TaskAction* are:

uiTaskAction	Comments
0	Add active task
2	Update active task
3	Remove active task
4	Splash-screen task
5	Final cleanup tasks

A scenario whereby 0x0B1000 messages are used is at the onset of opening the untrusted file, when the sandbox adds an active task in the broker with:

Fields	Values	Comments
uiTaskID	0x00000001	Arbitrary value
uiTaskAction	0x00000000	Add active task
uiTaskCurrentProgress	0x00000000	Current progress of task, at 0%
uiTaskMaxProgress	0x00000064	Max progress of task, at 100%
wzTaskName	“Opening: test.docx”	Arbitrary string
wzTaskDescription	NULL	Arbitrary string

The broker tracks the task with a new *ProtectedViewTask* object, and updates the *lpSameBrokerApp->ui32TaskListIndex* and *lpSameBrokerApp->lpTaskList* accordingly.

Next, the sandbox updates broker on the task progress with a subsequent 0x0B1000 message:

Fields	Values	Comments
ui32TaskID	0x00000001	Arbitrary value (same as previous message)
ui32TaskAction	0x00000002	Update active task
ui32TaskCurrentProgress	0x00000000	Current progress of task, at 100%
ui32TaskMaxProgress	0x00000064	Max progress of task, at 100%
wzTaskName	“Opening: test.docx”	Arbitrary string
wzTaskDescription	NULL	Arbitrary string

With this message, the broker walks down *lpSameBrokerApp->lpTaskList* and updates *ui32TaskCurrentProgress* of the corresponding *ProtectedViewTask* object.

Finally, after the task is completed, the broker removes the task upon this request:

Fields	Values	Comments
ui32TaskID	0x00000001	Arbitrary value (same as previous message)
ui32TaskAction	0x00000003	Remove active task
ui32TaskCurrentProgress	0x00000000	Current progress of task, at 100%
ui32TaskMaxProgress	0x00000064	Max progress of task, at 100%
wzTaskName	"Opening: test.docx"	Arbitrary string
wzTaskDescription	NULL	Arbitrary string

The *ui32TaskAction* value of 5 is used as a final clean-up for *lpSameBrokerApp->lpTaskList*, which sets *lpSameBrokerApp->ui32TaskListSize* to Null.

Message 0x0D1000

The broker checks the registry key "*HKCU\Software\Policies\Microsoft\Office\15.0\Common\Shipasserts\disableshipasserts*" value, created as part of Active Directory group policies to enable/disable non-critical error reporting. This is probably used in conjunction with the 0x0E1000 message.

Message Body for 0x0D1000 Request	
UINT32	ui32Unknown

Message 0x0E1000

For the 0x0E1000 message, the broker creates the dump-related files ("*.dmp*", "*.cvr*", "*.cvr.dat*", "*.txt*") in the "%UserProfile%\ AppData\Local\Microsoft\Office\ShipAsserts" directory, if the non-critical reporting policy ("*HKCU\Software\Policies\Microsoft\Office\15.0\Common\ShipAsserts*") is enabled. It also execute DW20.exe in manifest mode.

Message Body for 0x0E1000 Request		
UINT32	ui32FileSalt	-
UINT32	ui32Unknown	-
LPWSTR	lpwFileName	Pointer to *wzFileName*, or Null
UINT16	ui16FileNameSize	-
WCHAR[]	wzFileName	Duplicated as file "*winword.exe.<ui32FileSalt>.cvr*"

Message 0x101000

There is no message body. Upon receiving this request, the broker does a *PostMessageW (lpSameBrokerApp->hOPHParentWnd, WM_PSD_YAFULLPAGERECT)* action on every untrusted file.

Message 0x111000

If the UINT32 *lpViewFile->Field_00->Field_CC* field is more than 0x00 and less than 0x800000000, the broker returns a *ui32Status* of 0x80004004. Else it returns a 0x00 *ui32Status*.

Message Body of 0x111000 Request	
UINT32	ui32ViewID

Message 0x131000

The broker executes three boundary feedback actions relating to Windows Touch Programming, which appears to be intended for the Touch Mode feature.

Message Body for 0x0131000 Request		
UINT32	ui32PanningAction	• 0: This is a *BeginPanningFeedback()* request • 1: This is a *UpdatePanningFeedback()* request • 2: This is a *EndPanningFeedback()* request
HWND	hWndProtectedView	Can be any HWND belonging to Protected-View
LONG	lTotalOverpanOffsetX	Only for *UpdatePanningFeedback()* (*ui32PanningAction*=1)
LONG	lTotalOverpanOffsetY	Only for *UpdatePanningFeedback()* (*ui32PanningAction*=1)
UINT32	ui32BooleanFlag	• If *UpdatePanningFeedback()*, this is the *fInInertia* flag • If *EndPanningFeedback()*, this is the *fAnimateBack* flag

Message 0x151000

The broker returns the full-path of the *ui32FileID* file.

Message Body for 0x151000 Request		
UINT32	ui32FileID	0x00 to 0x97. See below
Message Body for 0x151000 Response		
LPWSTR	lpwFileName	Pointer to *wzFileName*, or Null
UINT16	ui16FileNameSize	-
WCHAR[]	wzFileName	-

The non-exhaustive list of *ui32FileID*-to-*wzFileName* mapping is:

uiFileID	*wzFileName*
0x00	C:\Program Files\Microsoft Office\Office15\1033\ACCOLKI.DLL
0x01	C:\Program Files\Microsoft Office\Office15\MSACCESS.EXE
…	
0x42	C:\Program Files\Common Files\Microsoft Shared\OFFICE15\MSO.DLL
0x43	C:\Program Files\Common Files\Microsoft Shared\VBA\VBA7.1\VBE7.DLL
…	
0x95	C:\Program Files\Microsoft Office\Office15\WINWORD.EXE
0x96	C:\Program Files\Microsoft Office\Office15\WWLIB.DLL
0x97	C:\Program Files\Microsoft Office\Office15\GKWord.dll

Message 0x011100

Message Body for 0x011100 Request		
UINT8	ui8Unknown1	0 to 7
UINT8	ui8Unknown2	0 to 1
UINT8	ui8Unknown3	-
UINT8	ui8Unknown4	0 to 1
UINT32	ui32Unknown5	*ui32Unknown5* & 0xFF
LPVOID	lpMarshaledInterface	Pointer to *uchMarshalledInterface*, or Null
UINT32	ui32MarshaledInterfaceSize	-
UCHAR[]	uchMarshaledInterface	Marshalled interface as OBJREF structure

Message 0x051100

The broker moves the *ui32ViewID* untrusted file to the sandbox directory. The request does not have a message request body and its response is:

Message Body for 0x051100 Response		
LPWSTR	lpwNewUntrustedFile	Pointer to *wzNewUntrustedFile*, or Null
UINT16	ui16NewUntrustedFileSize1	*ui16NewUntrustedFileSize2* + 1
UINT16	ui16NewUntrustedFileSize2	Size of *wzNewProtectedViewFile*, excluding terminating Null
WCHAR[]	wzNewUntrustedFile	New location of untrusted file

Message 0x061100

The 0x061100 message has neither request nor response body. The broker returns 0 if the global fields *WWLIB.10B27C8->Field_18* is non-Null and *WWLIB.10B27C8* is 2. Otherwise it returns 0x80070057 or 0x80004005.

Message 0x0F1100

The broker will perform a Bing search for the *wzQuery* query, with this URL: "http://o15.officeredir.microsoft.com/r/rlidSearchWithBing?ver=15&app=winword.exe&clid=1033&lidhelp=0409&liduser=0409&lidui=0409&q=<wzQuery>"

Message Body for 0x0F1100 Request	
LPWSTR	lpwQuery
UINT16	ui16QuerySize
WCHAR[0x823]	wzQuery

Message 0x121000

The broker displays a "*Microsoft Office Customizable Alert*" prompt with *MessageBeep()* and *MessageBox()*, and also creates a "*Microsoft Office 15 Alerts*" system event log entry for this error message *ReportEvent()*.

Message Body for 0x121000 Request		
UINT32	ui32ViewID	
LPWSTR	lpwErrorCaption	Pointer to *wzErrorCaption*, or Null
LPWSTR	lpwErrorText	Pointer to *wzErrorText*, or Null
UINT32	ui32UTypeButton	*MessageBox()*.uType, 0x00 to 0x0F
UINT32	ui32UTypeIcon	*MessageBox()*.uType icon
UINT32	ui32UTypeDefaultButton	*MessageBox()*.uType default button
INT32	i32Unknown1	{-5, -2, 1, 0, 1, 2, 3, 4}
UINT32	ui32Unknown2	{0, 1}
UINT32	ui32Unknown3	-
INT32	i32ErrorMessageID	To query value from registry key *"HKCU\Software\Policies\Microsoft\office\15.0\Word\CustomizableAlerts \<i32ErrorMessageID>"*
UINT32	ui32EventLogP1	*ReportEvent()*
UINT32	ui32EventLogP4	*ReportEvent()*
UINT32	ui32Unknown4	{1,3,5,7}, affects event log P1 and P4
UINT32	ui32Unknown5	-
LPWSTR	lpButtonText1	Pointer to wzButtonText1, or Null
LPWSTR	lpButtonText2	Pointer to wzButtonText2, or Null
LPWSTR	lpButtonText3	Pointer to wzButtonText3, or Null
LPWSTR	lpButtonText4	Pointer to wzButtonText4, or Null
LPWSTR	lpButtonText5	Pointer to wzButtonText5, or Null
LPWSTR	lpButtonText6	Pointer to wzButtonText6, or Null
UINT32	ui32UTypeModal	*MessageBox()*.uType modality
UINT32	ui32EventLogP3	*ReportEvent()*
UINT32	ui32Unknown6	-
UINT16	ui16ErrorCaptionSize	-
WCHAR[]	wzErrorCaption	*MessageBox()* and *ReportEvent()*
UINT16	ui16ErrorTextSize	-
WCHAR[]	wzErrorText	*MessageBox()* and *ReportEvent()*
UINT16	ui16ButtonText1Size	-
WCHAR[]	wzButtonText1	Custom button text 1
UINT16	ui16ButtonText2Size	-
WCHAR[]	wzButtonText2	Custom button text 2
UINT16	ui16ButtonText3Size	-
WCHAR[]	wzButtonText3	Custom button text 3
UINT16	ui16ButtonText4Size	-
WCHAR[]	wzButtonText4	Custom button text 4
UINT16	ui16ButtonText5Size	-

WCHAR[]	wzButtonText5	Custom button text 5
UINT16	ui16ButtonText6Size	-
WCHAR[]	wzButtonText6	Custom button text 6

Message 0x021100

The 0x021100 message causes an error message to be display from a standard set of error messages in *WWLIB.007EF930*.

Message Body of 0x021100 Request		
UINT32	ui32Flag	1
UINT32	ui32Unknown1	-
UINT32	ui32ErrorCode	See Appendix B
INT32	i32Unknown2	If -1, *iUnknown2* is not used
LPWSTR	lpwWString	Pointer to *wzWString*, or Null
UINT16	u16WStringSize	-
WCHAR[]	wzWString	-

The non-exhaustive list of standard error messages is in *Appendix B*

Message 0x0A1100

The broker displays an end-of-search message prompt similar to:

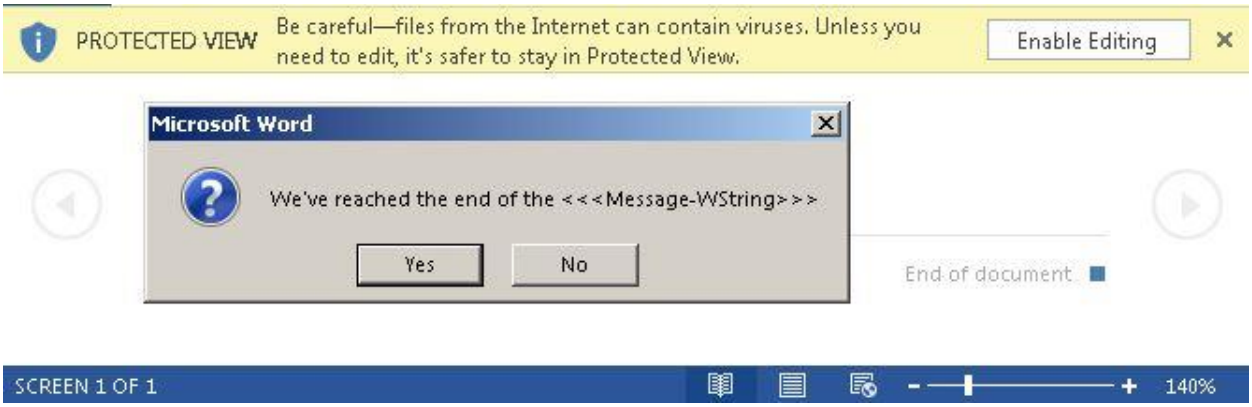


Figure 21: The message prompt for 0x0A1100 tag

Message Body for 0x0A1100 Request		
UINT32	ui32Unknown	If 0, *IPCMsg100->uiIPC0A1100* = 0, else *IPCMsg100->uiIPC0A1100* = 1
LPWSTR	lpwWString	Pointer to *wzWString*, or Null
UINT16	ui16WstringSize	-
WCHAR[]	wzWString	-

Message 0x0B1100

The 0x0B1100 request is also a standard error message display similar to the 0x021100 message, but with more control for placeholder-strings in *ui32ReplaceCode1* and *ui32ReplaceCode2*.

Message Body for 0x0B1100 Request		
INT32	i32XCoordinates	X-coordinates in *SetWindowPos()*
INT32	i32YCoordinates	Y-coordinates in *SetWindowPos()*
UINT32	ui32ErrorCode	See Appendix B
UINT32	ui32ReplacementCode1	See below
UINT32	ui32ReplacementCode2	See below
UINT32	ui32Unknown	-

The *ui32ReplacementCode1* and *ui32ReplacementCode2* combine for these eventual placeholder-strings:

ui32ReplacementCode1	*ui32ReplacementCode2*	Placeholder-Strings
0x0004	Any	“Building Blocks”
0x0100	Any	“comments”
0x0400	0x0006	“footnote separator”
0x0400	0x0007	“footnote continuation separator”
0x0400	0x0008	“footnote continuation notice”
0x0400	0x0009	“endnote separator”
0x0400	0x000A	“endnote continuation separator”
0x0400	0x000B	“endnote continuation notice”
0x0400	Any	“header/footer”
0x0800	Any	“footnotes”
0x2000	Any	“shape”
0x4000	Any	“endnotes”
0x8000	Any	“text box”
Any	Any	“document”

Message 0x031100

The broker updates *lpWWLIBIPCMsg->ui32IPC031100MsgID* and *lpWWLIBIPCMsg->uchIPC031100Contents* if the current *ui32MsgID* (in the message header) is greater than *lpWWLIBIPCMsg->ui32IPC031100MsgID*.

Message Body of 0x031100 Request		
UCHAR[0x1D4]	uchIPC031100Contents	Series of 0x00s and 0x01s (binary).

Message 0x041100

The broker updates *lpWWLIBIPCMsg->ui32IPC041100MsgID* and *lpWWLIBIPCMsg->uchIPC041100Contents* if the current *ui32MsgID* (in the message header) is greater than *lpWWLIBIPCMsg->ui32IPC041100MsgID*. In addition, the *uchIPC041100Contents* is also stored in *WWLIB.10BDA24[0x60]*.

Message Body of 0x041100 Request	
UCHAR[0x24]	uchIPC041100Contents

Message 0x071100

Message Body for 0x071100 Request		
UINT32	ui32Unknown1	-
UINT32	ui32Unknown2	0x00 to 0x03FF
UINT32	ui32Unknown3	0x00 to 0x03FF
UINT32	ui32Unknown4	0x00 to 0x03
UINT32	ui32Unknown5	LSB from 0x01 to 0x62
UINT32	ui32Unknown6	LSB from 0x01 to 0x62
UINT32	ui32Unknown7	-

Message 0x091100

The broker updates *lpWWLIBIPCMsg->ui32IPC091100MsgID* and *lpWWLIBIPCMsg->uchIPC091100Contents* if the current *ui32MsgID* (in the message header) is greater than *lpWWLIBIPCMsg->ui32IPC091100MsgID*.

Message Body for 0x091100 Request	
UCHAR[0x2C]	uchIPC091100Contents

Message 0x001100

The 0x001100 request does not have a message body.

Message Body for 0x001100 Response		
UINT8	ui8Unknown1	-
UINT8	ui8Unknown2	-
UINT16	ui16PartialLeakedAddr	Uninitialized UINT16 which would have higher-order address of *ViewTracker* object
UINT32	ui32Unknown3	-
UINT32	WWLIB.104D1F6	-

Message 0x081100

Message Body for 0x081100 Request		
INT32	i32Unknown1	-
INT32	i32Unknown2	-
UINT8	ui8bUnknown3	-
UINT8	ui8Unknown4	-

UINT8	ui8Unknown5	-
UINT8	ui8Pad	-

Message 0x0C1100

Message Body for 0x0C1100 Request		
UINT32	ui32Boolean	0 or 1

Message 0x0D1100

Message Body for 0x0D1100 Request		
UINT32	ui32Boolean	0 or 1

Message 0x0E1100

Message Body for 0x0E1100 Request		
INT32	i32Unknown	-
UINT32	ui32Boolean	0 or 1, and if *i32Unknown* > 0

8. Appendix B: Standard Error Codes and Messages

This section lists the *ui32ErrorCode* values and the corresponding error messages for the 0x021100 and 0x0B1100 messages. Some error messages contain curly and angle brackets, and are meant as placeholders.

Bracket Code	Placeholder
<A>	WSTRING “Microsoft Word”
<a>	WSTRING “Word”
<d>	UINT32 Error_Code
{b} / {c} / {i}	INT32 Various_Meanings
{e}	INT32 Unit_Twips_To_Be_Converted_To_Inches
{f}	WSTRING File_Name
{k} / {m}	WSTRING Various_Meanings
{k255}	WSTRING Various_Meanings, limited to 0xFF bytes
{n}	WSTRING_STRUCT Various_Meanings
{p1}, {p2}, {p3}	Respective _WSTRING in struct{ WSTRING* wstr1, WSTRING* wstr2,_WSTRING* wstr3}

The WSTRING_STRUCT type for {n} refers to such a structure:

```
WSTRING_STRUCT {
        WORD        wNumOfWChars;  //Number of WCHAR in WString
        WCHAR[]     WString;       //Not Null-terminated
}
```

Here is a non-exhaustive list of error messages:

ui32ErrorCode	Error Message
0053	Cannot insert a multi-user document as a subdocument.
01E9	Too many <a> documents are open. Please close a window.
0017	There is a serious disk error on file {m}.
01AC	All done. We made {i} replacements.
010A	Line spacing must be at least {e}.
010B	<a> cannot create the custom dictionary {k}.
01B0	{p1}{k} is an incorrect version of the {p2}{k} DLL.
01AF	{p1}{k} is not a valid {p2}{k} dll.
0161	<a> cannot open the {p1}{k} {p2}{k} for {p3}{k}.
01B4	The number must be between {c} and {c}.
025A	Word cannot open {k255} as a header file because it cannot be converted to a Word file format.
030A	The number must be a divisor of {b}.
0499	Cannot open {f}: This file has been protected with a password and cannot be opened.
04CE	This {n} cannot be deleted from the document.
	.....