Thinking outside the sandbox

Violating trust boundaries in uncommon ways

Brian Gorenc
Manager, Vulnerability Research
HP Security Research

Jasiel Spelman
Security Researcher
HP Security Research

Attacking the modern browser and its plug-ins is becoming harder as vendors employ numerous mitigation technologies to increase the cost of exploit development. An attacker is now forced to uncover multiple vulnerabilities to gain privileged-level code execution on his targets. First, an attacker needs to find a vulnerability, leak an address to get around ASLR, and bypass DEP to gain code execution within the renderer process. The attacker then needs to bypass the application sandbox to elevate their privileges, which will allow them to execute malicious code. Our journey begins at the sandbox and investigates some of the more obscure techniques used to violate this trust boundary.

Understanding trust boundaries

Over the last several years, most major browser and plug-in vendors released some form of application sandboxing to shore up their security posture. They could no longer rely on existing mitigation technologies like Stack Cookies (/GS), Data Execution Prevention (DEP), and Address Space Layout Randomization (ASLR) to stop determined adversaries. These vendors went back to the drawing board, analyzed their architectures, and defined a trust boundary (also known as a sandbox).

The primary purpose of a trust boundary is to define a clear separation inside an application where untrusted data crosses into a part of the application that expects data to be "trusted". At this boundary, the untrusted data can be validated and security policies applied to ensure it is well-formed. The code that handled the rendering of the user-supplied web page or document and corresponding data are considered "untrusted". This code is provided a confined operating environment and a limited set of APIs with which to work. This resulting sandbox enforces the boundary thus mitigating the impact of any code execution vulnerabilities that may exist within the untrusted sections of the application.

Browser and plug-in vendors, of course, rely heavily on the underlying operating system's security frameworks to implement their sandboxes. In 2007, Microsoft's David LeBlanc provided application developers with guidance on implementing "Practical Windows Sandboxing"[1]. In this guidance, LeBlanc recommended sandbox applications utilize restricted access tokens, job object limitations, and window station/desktop isolation to segment themselves from the other running processes. Microsoft, Adobe, and Google have implemented these recommendations to varying degrees. Let's investigate some of the sandboxing strategies that are applicable to the attacks described in this paper.

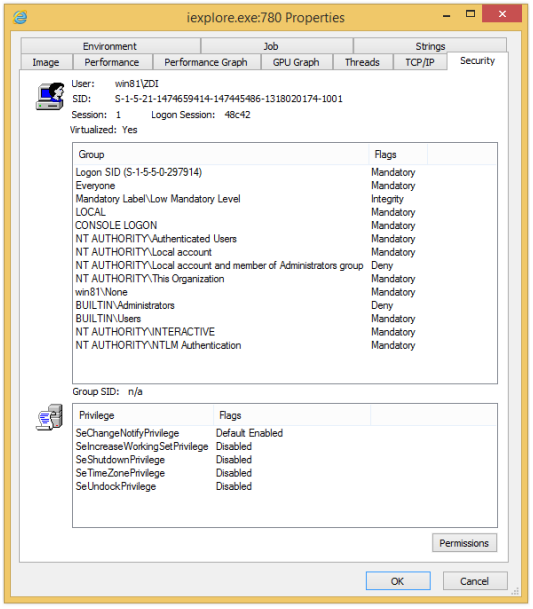

Figure 1: Microsoft Internet Explorer Restrictions

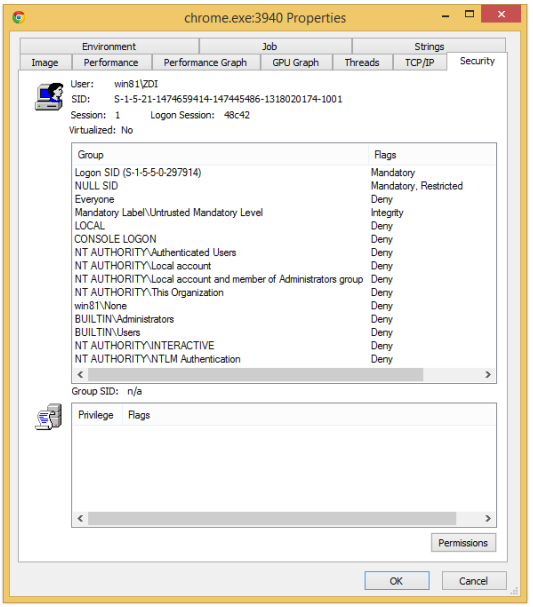

Figure 2: Google Chrome Restrictions

What are Restricted Access Tokens?

According to MSDN, an access token is an object that describes the security context of the process or thread[2]. It includes information such as the identity and privileges of the user account that are associated with the process. A restricted access token is exactly what it sounds like; an access token with disabled security identifiers (SIDs), deleted privileges, or restricted SIDs. A restricted access token can be obtained by calling CreateRestrictedToken or AdjustTokenPrivileges. Each browser and plug-in vendor has constrained their sandboxed processes using Restricted Access Tokens in unique ways. These differences can be inspected visually using Process Explorer[3].

What are Job Object Limitations?

A job object provides application developers a way of managing a group of processes as a unit[4]. From the sandboxing perspective, limitations can be applied to a job object and these limitations apply to all the processes associated with the job object. For example, the JOBOBJECT_BASIC_LIMIT_INFORMATION structure has the ability to limit the number of active processes associated with the job. For Adobe Reader and Google Chrome, this value is limited to 1 active process.

Another set of limitations that can be applied to a job object are user interface restrictions which are offered through JOBOBJECT_BASIC_UI_RESTRICTIONS[5]. Using this

[1] http://blogs.msdn.com/b/david_leblanc/archive/2007/07/27/practical-windows-sandboxing-part-1.aspx
[2] http://msdn.microsoft.com/en-us/library/windows/desktop/aa379316(v=vs.85).aspx
[3] http://technet.microsoft.com/en-us/sysinternals/bb896653.aspx
[4] http://blogs.msdn.com/b/david_leblanc/archive/2007/07/27/practical-windows-sandboxing-part-2.aspx
[5] http://msdn.microsoft.com/en-us/library/windows/desktop/ms684152(v=vs.85).aspx

structure, application developers can prevent the processes associated with the job from doing the following:

- Creating and switching desktops
- Changing display settings
- Exiting Windows
- Accessing global atoms
- Using USER handles not associated with the same job
- Reading data from the clipboard
- Changing system parameters
- Writing data to the clipboard

These restrictions go a long way in reducing what is possible if an attacker achieves code execution within the sandboxed process. For example, Adobe Reader and Google Chrome enable all of the above limitations on their sandboxed process[6]. Microsoft Internet Explorer, on the other hand, does not leverage job object limitations in their sandbox implementation.

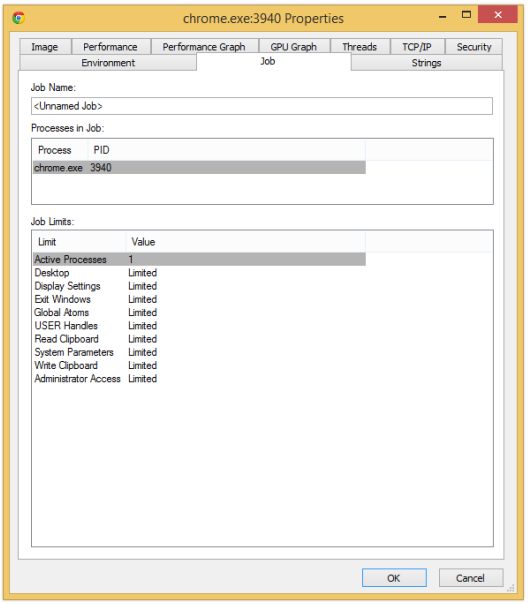

Figure 3: Google Chrome Job Object Limitations

What is Window Station and Desktop Isolation?

One of the final recommendations of "Practical Windows Sandboxing" is sandboxed applications should be placed on a separate window station and desktop[7]. Processes running on the same desktop can communicate with each other using window messages or hook procedures. As such, it is possible for a compromised process to leverage other processes running on the same desktop to gain elevated privileges (i.e. shatter attacks).

The primary goal of isolating the sandboxed process this way is to prevent attacks that use window messages and hook procedures. A compromised sandboxed process that is running by itself on a separate window station and desktop has limited ability to leverage window messages in the way previously described. This sandboxing technique is not consistently applied across browser and plug-in vendors.

What is Mandatory Integrity Control?

Microsoft introduced Mandatory Integrity Controls into their security architecture with the release of Windows Vista[8]. These were intended to represent the level of trust one could have in a process, file, or other securable objects and to provide another layer of control beyond the existing security features. The operating system provides the following integrity levels:

- Untrusted
- Low
- Medium
- High
- System

The lower the integrity level assigned to the process the fewer resources (files, registry keys, etc.) it will be allowed to modify if compromised. User Interface Privilege Isolation, also introduced in Vista, prevents processes with lower integrity levels from sending window messages to or installing hooks in processes running at a higher level further restricting message type attacks. The browser and plug-in vendors were quick to leverage mandatory integrity controls in their sandbox

[6] http://blogs.adobe.com/security/2010/10/inside-adobe-reader-protected-mode-part-2-the-sandbox-process.html
[7] http://blogs.msdn.com/b/david_leblanc/archive/2007/07/27/practical-windows-sandboxing-part-3.aspx
[8] http://msdn.microsoft.com/en-us/library/bb625963.aspx

implementations. For example, Microsoft Internet Explorer's broker process is running with medium integrity level and its render process is running with low integrity level. Google Chrome uses the untrusted integrity level for its rendering process and medium integrity level for its broker process.

How does the sandboxed process communicate?

Beyond these restrictions, isolated processes running at different integrity levels need to be able to communicate with the underlying operating system to provide the rich feature sets consumers demand. As a result, browser and plug-in vendors developed a restricted set of APIs which the sandboxed process must use to execute privileged functionality. The broker process provides all the handlers for the exposed APIs and is responsible for enforcing any policies or restrictions being placed on a specific APIs. The APIs typically take the form of a shared memory Inter-Process Communication (IPC) framework to handle requests back and forth between the sandboxed process and the broker process[9]. Microsoft Internet Explorer also provides a COM-based IPC for part of the broker's interface with the sandboxed process[10]. For example, Adobe Reader relies heavily on the Chromium's sandbox IPC implementation though Adobe-specific IPC calls were implemented to support the functionality required by Reader[11].

As stated already, each vendor applies these restrictions in their sandbox designs differently. Google Chrome (and indirectly, Adobe Reader) uses a majority of these techniques to ensure their sandboxed processes are as isolated as possible. Microsoft Internet Explorer, on the other hand, does not apply the limitations provided by the use of a job object. It also does not run the sandboxed process on an isolated window station and desktop. These facts can leave a person scratching their head when it was Microsoft that provided the "Practical Windows Sandboxing" advice.

Attack Surface Archetypes

With all these security features, one would think that developing an exploit to break out of an application's sandbox would be difficult. It's definitely an additional challenge for exploit developers but there are still many opportunities to escape. Once the attacker achieves code execution within the sandboxed process, they need to trigger another weakness to elevate privilege in order to do real damage. These attackers will typically focus on the following areas as they have proven to be fruitful in the past.


Figure 4: Andreas Schmidt and Sebastian Apelt exploiting Microsoft Internet Explorer

Kernel APIs

One of the largest attack surfaces available to an exploit author from within the sandboxed process is the Windows kernel. Vulnerabilities within the kernel also offer the side benefit that after exploitation the resulting payload will be running with SYSTEM privileges. These types of vulnerabilities are difficult to discover as the kernel has been through many security reviews and has been highly tested prior to release. That said, researchers with a keen eye are able to uncover subtle defects that can be leveraged in a chained exploit to gain the highest level of privilege on the system.

[9] http://www.chromium.org/developers/design-documents/sandbox
[10] http://msdn.microsoft.com/en-us/library/ie/ms537319(v=vs.85).aspx
[11] http://blogs.adobe.com/security/2010/11/inside-adobe-reader-protected-mode-part-3-broker-process-policies-and-inter-process-communication.html

Though these issues are rare, contestants at the last two Pwn2Own contests demonstrated that this type of weakness can be quite successful. In Pwn2Own 2013, Jon Butler and Nils from MWR Labs obtained a SYSTEM-level compromise through Google Chrome. They chained a type confusion vulnerability[12] that occurred due to the use of static_cast with a vulnerability[13] in NtUserMessageCall that was a result of the misuse of a Boolean argument. At Pwn2Own 2014, Andreas Schmidt and Sebastian Apelt combined multiple use-after-free vulnerabilities[14] and a double-free vulnerability within AFD.sys to demonstrate the SYSTEM-level calc shown in the image at the right.

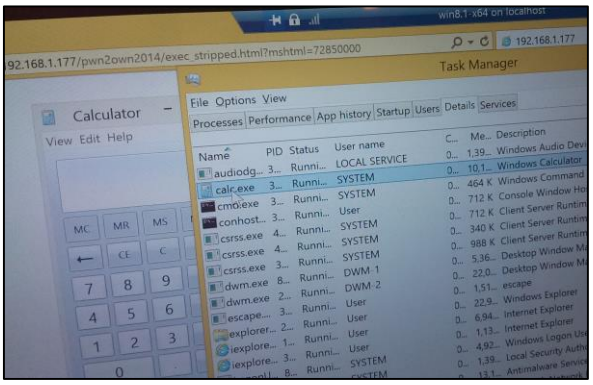

Figure 5: calc.exe payload running at SYSTEM

Inter-Process Communication (IPC) Handling

The next logical interface to attack is the IPC messages and infrastructure used by the sandboxed process to communicate with the medium-integrity broker. Although the APIs are limited, there is still a significant amount of functionality provided by them to support the feature sets of the browser or plug-in. For example, the brokers for Google Chrome and Adobe Reader each provide a large number of IPC messages or "Cross Calls" to the sandboxed process[15]. Microsoft Internet Explorer takes it further by providing not only an IPC framework but also a set of COM interfaces to the sandboxed process.

There are many defects one could look for inside of the processing to help them escape. Parsing errors in the handling of parameters being sent to the broker process is a common issue that can be uncovered. In fact, a heap overflow was the root cause of one of the first Adobe Reader sandbox escapes to be found in the wild. This overflow occurred due to the way the broker process handled the GetClipboardFormatNameW requests[16]. Logic errors are an obvious weakness that may manifest in the broker process and could easily be taken advantage of to elevate privilege. CVE-2013-4015 demonstrates how a logic error can be used to bypass the policy check and elevate privileges[17]. This vulnerability was due to how ieframe!GetSanitizedParametersFromNonQuotedCmdLine() handled the "\t" whitespace character. Using this character, it was possible to trick Internet Explorer into launching an attacker-specified executable name at medium integrity.

Shared Resources

Privileged resources that are shared (or leaked, accidently) between the sandboxed process and the broker process can provide an opportunity for escape. These resources can take the form of handles for sections, files, keys, etc. Depending on the access rights (e.g. write access) associated with these resources, an attacker may be able to gain privileges when the resource is handled. One common avenue for shared resources to be leaked is third-party DLLs that improperly use the various handles that are available. The browser developers are taking a proactive stance on this topic by blocking DLL access to the sandboxed process through blacklisting[18].

Of course, the previous attack surface archetypes are not the only way to jump out of the sandbox. Over the years, researchers have discovered innovative ways to attack this trust boundary including:

- Base Named Object Namespace Squatting

[12] http://zerodayinitiative.com/advisories/ZDI-13-064
[13] http://zerodayinitiative.com/advisories/ZDI-13-170/
[14] http://zerodayinitiative.com/advisories/ZDI-14-192/
[15] https://media.blackhat.com/bh-us-11/Sabanal/BH_US_11_SabanalYason_Readerx_WP.pdf
[16] https://blogs.mcafee.com/mcafee-labs/digging-into-the-sandbox-escape-technique-of-the-recent-pdf-exploit
[17] https://www.blackhat.com/docs/asia-14/materials/Yason/WP-Asia-14-Yason-Diving-Into-IE10s-Enhanced-Protected-Mode-Sandbox.pdf
[18] http://media.blackhat.com/bh-eu-11/Tom_Keetch/BlackHat_EU_2011_Keetch_Sandboxes-WP.pdf

- Null DACLs Abuse
- Socket-Based Attacks
- Policy Engine Subversion
- Third-party Software/Local Service Weaknesses

Vendors have spent a lot of money and hours auditing their implementations, but even with all this effort, memory corruption vulnerabilities in the kernel or IPC architecture are still one of the more common vectors for attackers. Logic errors within the broker process and corresponding policy engines have also been bountiful in the past. In the end, sandboxing a process is a difficult endeavor and trade-offs are made by application developers who need to balance security and performance. These trade-offs may leave just enough space for a skillful attacker to escape and do more damage.

Uncommon attack vectors

When looking for sandbox bypasses, it is important to keep the less common attack vectors in mind. This section will take an in-depth look at some of the more obscure escapes used by contestants of the Pwn2Own 2014 hacking contest. These techniques were used once the contestant achieved initial code execution from within the target's render process. They were selected to demonstrate the different approaches being investigated by the research community.

Save File Dialog abuse

Microsoft Internet Explorer is no stranger to exploitation at Pwn2Own. In fact, it is quite common for contestants targeting plug-ins to find a code execution vulnerability in the plug-in code and then shift their focus to escaping the Internet Explorer sandbox instead of the plug-in's sandbox. This was the technique VUPEN Security used when they targeted Adobe Flash. VUPEN began by abusing the handling of ExternalInterface. By manipulating a SWF's objects, they forced a dangling pointer to be reused after it was freed. This vulnerability (CVE-2014-0506) was leveraged to gain code execution within Adobe Flash[19]. Once this was complete, VUPEN abused the Save File Dialog to break out of the sandbox and elevate privileges.

Exploitation

As a result of running as low integrity processes, sandboxed processes have limited ability to save files to the file system. Specifically, these processes can only write to locations that have been explicitly marked as writable by a low integrity processes. There are a few instances where the sandboxed process will need to write to a location outside of these areas, such as when downloading files.

As such there is a remote procedure call within the Internet Explorer broker to handle this, reachable through the IProtectedModeAPI COM interface. The CProtectedModeAPI class, which implements the IProtectedModeAPI interface, exposes two functions to handle saving files to unrestricted locations:

- CProtectedModeAPI::ShowSaveFileDialog
- CProtectedModeAPI::SaveFileAs

The sandboxed process initiates the request by calling CProtectedModeAPI::ShowSaveFileDialog within the broker, which will result in the broker prompting the user for confirmation to save the file. This function takes several

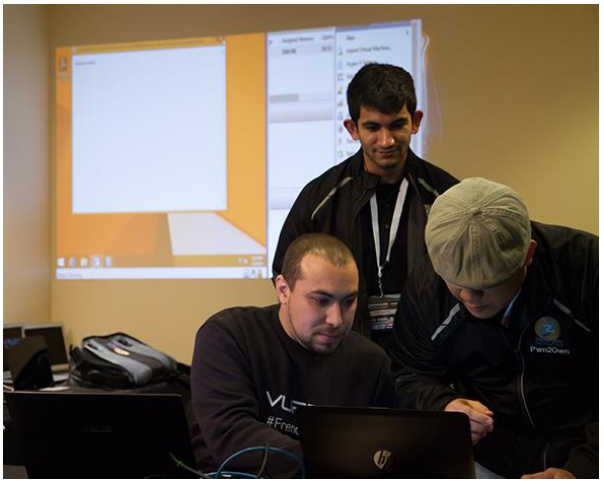

Figure 6: Chaouki Bekrar (left) of VUPEN Security exploiting Adobe Flash

```
IEFRAME!CProtectedModeAPI::ShowSaveFileDialog:
6880573f 8bff            mov     edi,edi
0:015> du poi(@esp+c)
042acaf0  "C:\Users\ZDI\AppData\Local\..\ro"
042acb30  "aming\Microsoft\Windows\Start Me"
042acb70  "nu\Programs\Startup\hello.hta\."
```

Figure 7: Save As Location

[19] http://zerodayinitiative.com/advisories/ZDI-14-092/

arguments, including the desired destination. This location is stored within the broker to prevent abuse later on.

Handling of the dialog box is delegated to the IEGetSaveFileName function, which will return 0, if the user accepted the request, or 1, if the user refused the request, and a negative value if there was an error. If the user accepts the file save, the CProtectedModeAPI goes into the "CProtectedModeAPI::SaveFileAs" state. At this point, the sandboxed process can proceed with writing the file to a location with low integrity. Once the file has been written, the sandboxed process makes the last request by calling the CProtectedModeAPI::SaveFileAs function within the broker. The broker verifies that it is in the "CProtectedModeAPI::SaveFileAs" state. If so, it moves the file to location specified earlier. Internet Explorer applies the Mark of the Web to downloaded files to ensure that they are loaded into the appropriate security zone when opened. As such this will have to be chained with something else before being a full sandbox escape. Figure 10 below shows an example of what the dialog box looks like.

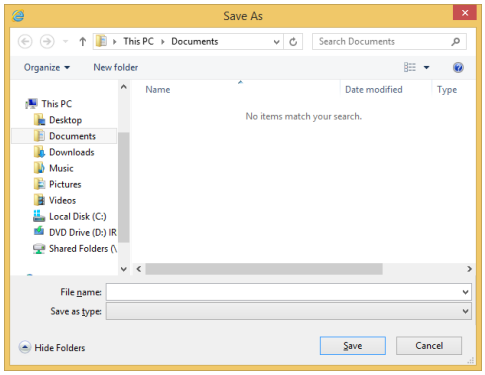

Figure 10: Save As Dialog Abuse

This vulnerability allows for files to be created at arbitrary locations, however, one of the downfalls is that the file is subject to the Mark of the Web. VUPEN maneuvered around this limitation by taking advantage of the way Internet Explorer handles recovering from a crash. In order to restore the page being viewed in the event of a crash information about the session is stored. The CRecoveryStore class handles this and is one of a few classes within the broker that can be instantiated from the sandboxed process. Control over the contents of the recovery store is limited, but arbitrary strings can be written by specifying invalid page titles and locations. Also, the location of the recovery store is at a predictable location and doesn't have the Mark of the Web applied to it.

All of the pieces of the sandbox bypass are now in place. It starts with creation of a controlled recovery store that contains malicious script. The next step is to trigger the file save dialog within the broker. Although this will result in the user having to close the dialog window, an attacker would be able to continue regardless of the response. The final step is to trigger the actual file save, which will now move the controlled recovery store from its predictable location into the user's Startup folder. If the resulting file can be processed as a HTML application then malicious script will be executed the next time the user logs in allowing for code execution outside of the sandbox.

Root cause analysis

The vulnerability exists due to the improper handling of the user's response to the file save request within the CProtectedModeAPI::ShowSaveFileDialog function. The broker defaults to the "CProtectedModeAPI::SaveFileAs" state and only changes back to an

```
IEFRAME!CProtectedModeAPI::SaveFileAs:
688041fc 8bff            mov     edi,edi
0:015> du poi(@esp+8)
0428ab14  "C:\Users\ZDI\AppData\Local\Micro"
0428ab54  "soft\Internet Explorer\Recovery\"
0428ab94  "Active\Microsoft.Website.9CB8E69"
0428abd4  "8.8730F9E8\{6DF57AB3-FBB7-11E3-9"
0428ac14  "729-000C2976B060}.dat"
```

Figure 8: Recovery Store Location

```
IEFRAME!CTabRecoveryData::SetCurrentTitle:
68641c26 8bff            mov     edi,edi
0:011> da poi(@esp+8)
04292de4  "<script language='vbscript'>Set "
04292e04  "obj = CreateObject("Wscript.Shel"
04292e24  "l").obj.Run "calc.exe"</script>"
04292e44  ""
```

Figure 9: Script Included in the Recovery Store

empty state if there was an error creating the window. This means that CProtectedModeAPI will still be in the "CProtectedModeAPI::SaveFileAs" state regardless of the button that the user clicked. In the event an error is returned, the renderer can simply reissue the request. At this point, the sandboxed process can call CProtectedModeAPI::SaveFileAs to move a file regardless as to whether or not the user allowed it. Exploitation of this vulnerability also leverages the fact that the broker does not properly validate the file specified in CProtectedModeAPI::SaveFileAs, allowing for files outside of the sandbox to be moved. At this point the attacker has an arbitrary file write with the constraint that the Mark of the Web will be applied to the written file.

Remediation

To patch this issue, Microsoft did a couple of things. They removed CRecoveryStore from the list of classes that are allowed to be instantiated from the sandboxed process. This was done wholly within the CIEUserBrokerObject::BrokerCreateKnownObject function, where they simply removed the CLSID for IERecoveryStore from the list of accepted values. To fix the issue with the save dialog box, they stopped assuming success, and switched to only changing to the success state when the user accepted the dialog box.

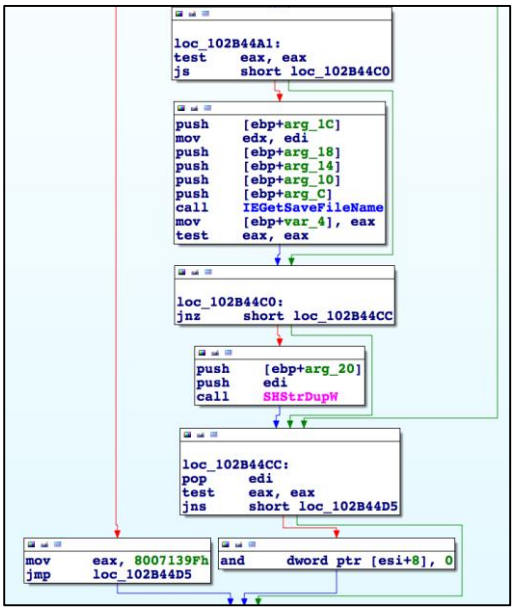

Figure 11: Pre-Patch Code

You can see in the pre-patched image that the state is only zeroed out if eax is a negative value, which completely misses the case where the user cancelled the dialog box resulting in eax being 1. The post-patch image shows the patched version of the function where the state is only changed to "CProtectedModeAPI::SaveFileAs" after the user has accepted the dialog box.

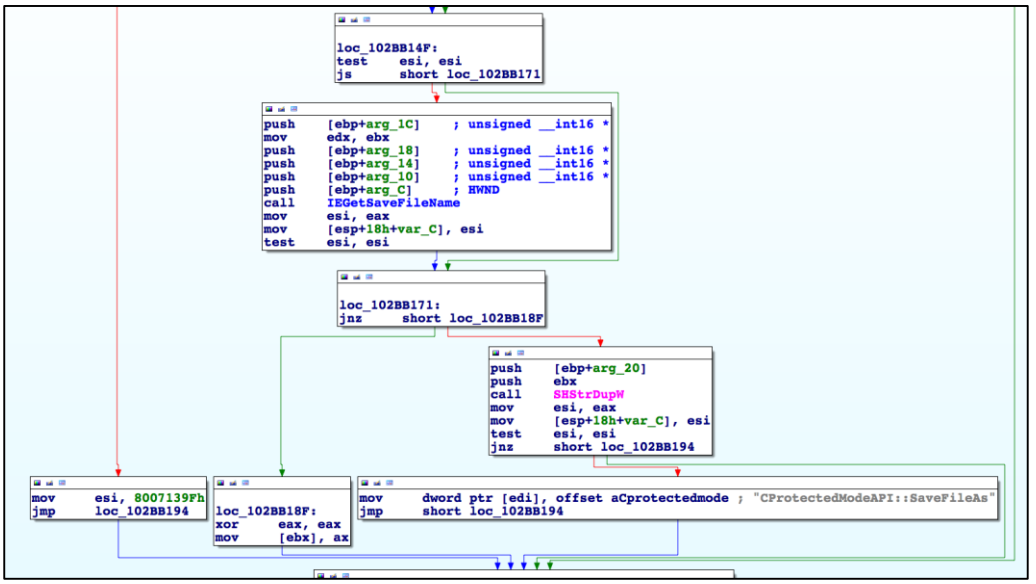

Figure 12: Post-Patch Code

Clipboard abuse

Long-time Pwn2Own winner, VUPEN, also discovered the next sandbox bypass which existed in Google Chrome. They started out by exploiting a use-after-free vulnerability in Blink bindings (CVE-2014-1713)[20]. Interestingly enough, it was discovered that this vulnerability also affected WebKit (e.g. Apple Safari) so it was also disclosed to Apple at the contest. VUPEN followed the exploitation of the use-after-free with a weakness in how Google Chrome allows specific types on the Clipboard.

[20] http://zerodayinitiative.com/advisories/ZDI-14-086/

Exploitation

One of the most common actions performed within a browser is the copying of data within a web page. With any application on Windows, this results in a call to SetClipboardData within user32.dll. Once on the clipboard, responsibility for proper handling falls to any application that accesses and processes it. Microsoft documentation for GetClipboardData explicitly states this[21]:

> **Caution** Clipboard data is not trusted. Parse the data carefully before using it in your application.

On Windows, processes within a window station share certain resources such as the clipboard contents. Chrome mitigates the security implications of this by having the renderer processes run within a restricted job object within a different window station.

In Chrome, there is an IPC to handle putting data onto the clipboard from the renderer process. This occurs within the ClipboardHostMsg_WriteObjectsAsync and ClipboardHostMsg_WriteObjectsSync cross calls. On Windows, both of these functions push a task to the worker thread that calls ClipboardMessageFilter::WriteObjectsOnUIThread, resulting in repeated calls to Clipboard::DispatchObject. Clipboard::DispatchObject then checks the type of object that is being written to the clipboard and calls the appropriate function to handle serializing it onto the clipboard. For example, if the desired object type is text, then the WriteText function will be called. Alternatively, if the desired object is arbitrary data, then the WriteData function will be called. Chrome maintains its own types for objects, which are then converted into the native operating system's object types when it comes time to place the object onto the clipboard.

Exploitation takes advantage of the MoreOlePrivateData clipboard format, which is represented by 0xC016. Among other things, this clipboard format can be used to instantiate an arbitrary COM control, which will occur at medium integrity. Since the kill bit is not checked on ActiveX controls loaded in this manner, it is possible to load a vulnerable ActiveX control and use it to achieve code execution.

Putting it all together, exploitation at medium integrity occurs by preparing a block of data to instantiate vulnerable COM controls. The renderer process then issues a ClipboardHostMsg_WriteObjectsAsync cross call with the ObjectType set to CBF_DATA. This leads to populating the clipboard with objects of type 0xC016. The next time Windows Explorer process tries to read from the clipboard, such as right-clicking on the desktop, it will instantiate the desired COM controls at medium integrity.

Root cause analysis

The vulnerability exists due to the failure to restrict the type of messages that are allowed to be posted to the clipboard from the renderer process. The renderer process can request that the broker put arbitrary message types onto the clipboard. Specifically, if the renderer process issues a ClipboardHostMsg_WriteObjectsSync or ClipboardHostMsg_WriteAsync cross call with an ObjectType of CBF_DATA, then a call to WriteData will be made where both the format and message will be read from the renderer. Exploitation at medium integrity now requires a vulnerability in anything that handles CBF_DATA from the clipboard.

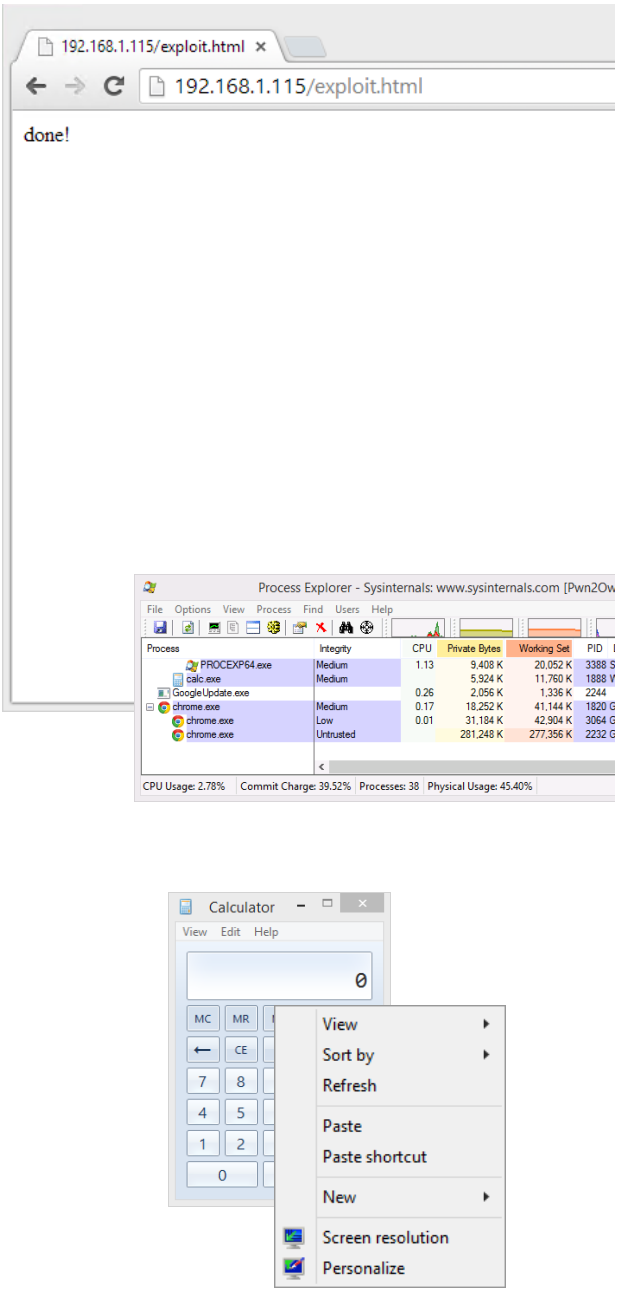

Figure 13: Sandbox Escape using MoreOlePrivateData

What is a Junction Point?

A junction point is a symbolic link to a directory. This feature of the NTFS file system acts as an alias to the directory pointed to by the junction point.

[21] http://msdn.microsoft.com/en-us/library/windows/desktop/ms649039(v=vs.85).aspx

Remediation

When Google patched the Pwn2Own bugs, they were nice enough to include links containing the bug IDs. Based off that, we know that commit edc1250e0cf03038db503086dfd31082ed694d69 was responsible for patching the vulnerability. This commit made several changes to the clipboard handling code but the root of the patch occurs within ScopedClipboardWrite::WritePickledData which now begins with the following check:

```
void ScopedClipboardWriter::WritePickledData(
    const Pickle& pickle, const Clipboard::FormatType& format) {
  // |format| may originate from the renderer, so sanity check it.
  if (!Clipboard::IsRegisteredFormatType(format))
    return;
```

Clipboard::IsRegisteredFormatType takes in a clipboard format and returns whether or not it has been explicitly registered as an allowed format. After the patch, a format type of 0xC016 will fail validation at this point.

Symbolic link abuse

A last minute anonymous entrant to Pwn2Own discovered this final sandbox escape. The participant started the exploit by leveraging a flaw that existed within the handling of TypedArray objects (CVE-2014-1705)[22]. By carefully manipulating an object the contestant could read and write data to any address allowing code execution under the context of the current process. Once the initial payload was running, the contestant used a vulnerability in the way the broker returns privileged file handles to gain medium integrity code execution.

Exploitation

Google Chrome uses a SQLite database to store data for an opened tab. There is an IPC to facilitate creation and access to this database from the renderer process. This occurs within the DatabaseHostMsg_OpenFile cross call, though this call will also create files despite the name. This cross call results in DatabaseMessageFilter::OnDatabaseOpenFile being called, which will call DatabaseUtil::GetFullFilePathForVfsFile if a file was specified. This function is responsible for merging the desired file with the base directory path, to ensure that access outside the sandbox does not occur. A check within GetFullFilePathForVfsFile shows that the Chrome team treated the supplied filename as potentially malicious:

```
// Watch out for directory traversal attempts from a compromised renderer.
if (full_path.value().find(FILE_PATH_LITERAL("..")) !=
        base::FilePath::StringType::npos)
  return base::FilePath();
return full_path;
```

VfsBackend::OpenFile is called after the call to GetFullFilePathForVfsFile, which in turn calls CreatePlatformFile. The file handle is finally created within CreatePlatformFileUnsafe, which CreatePlatformFile is a thin wrapper for, by making a call to the CreateFile Windows API. Lastly, the file handle is duplicated for use within the renderer and then returned through the IPC mechanisms.

The ability to create a file can be exploited due to a peculiarity of Windows. All files stored in NTFS have a stream, which can be accessed by appending the stream name and stream type to the end of the file path as colon separated values. In this case, "$INDEX_ALLOCATION" is the stream type that specifies a directory stream and "$I30" is the stream name that specifies the default stream name. By appending

[22] http://zerodayinitiative.com/advisories/ZDI-14-088/

":$I30:$INDEX_ALLOCATION" to the filename, the call to CreateFile specifies that it wants to access the default directory stream of the filename. This effectively sets PLATFORM_FILE_BACKUP_SEMANTICS without requiring that the flag actually be specified.

Putting this all together, the renderer process issues a DatabaseHostMsg_OpenFile cross call with "$I30:$INDEX_ALLOCATION" appended to the filename resulting in the broker creating a directory and returning back the handle. The renderer then makes a call to DeviceIoControl using FSCTL_SET_REPARSE_POINT as the IoControlCode to turn the newly created directory into a junction point to an arbitrary location such as the root of the C: drive. The last step is to create or modify a file off of this privileged handle, for example, within the user's Startup directory, to achieve code execution at medium integrity.

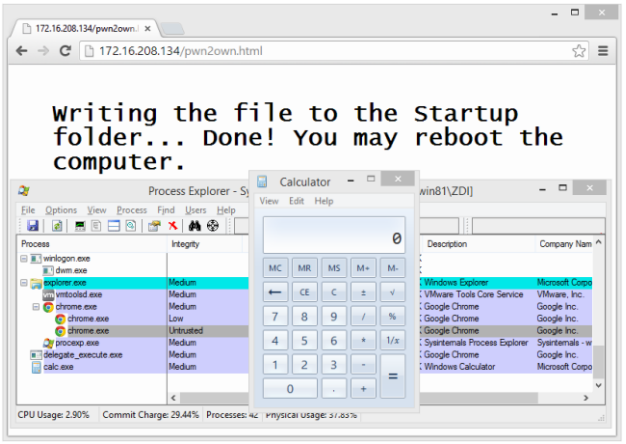

Figure 14: Junction Point Based Escape

Root cause analysis

The root of this vulnerability stems from an oddity in Windows. Although symbolic links cannot be created by a low privileged process, a junction point can. A junction point is a type of reparse point that essentially acts as a symbolic link to a directory. Furthermore, hard links would be a potential option if not for the fact that they take the paths as arguments whereas junction points are created using a file handle to a call to DeviceIoControl. One issue with junction points is that they require a file directory handle in order to be created. This would typically be handled by passing PLATFORM_FILE_BACKUP_SEMANTICS as a flag to CreateFile, but the DatabaseHostMsg_OpenFile cross call only allows certain flags through. By specifying "$I30:$INDEX_ALLOCATION" in the filename, we are able to indirectly set this flag.

Remediation

This vulnerability was patched with commit 693fcbe943b19153b14b3c4c18f6eb4edb42a555 within CreatePlatformFileUnsafe in platform_file_win.cc:

```
HANDLE file = CreateFile(name.value().c_str(), access, sharing, NULL,
                         disposition, create_flags, NULL);

if (INVALID_HANDLE_VALUE != file){
  // Don't allow directories to be opened without the proper flag (block ADS).
  if (!(flags & PLATFORM_FILE_BACKUP_SEMANTICS)) {
    BY_HANDLE_FILE_INFORMATION info = { 0 };
    BOOL result = GetFileInformationByHandle(file, &info);
    DCHECK(result);
    if (info.dwFileAttributes & (FILE_ATTRIBUTE_DIRECTORY |
                                 FILE_ATTRIBUTE_REPARSE_POINT)) {
      CloseHandle(file);
      file = INVALID_HANDLE_VALUE;
    }
  }
}
```

The patch opens the file as requested and queries the file attributes to ensure that unless a handle to a directory was requested, the file handle does not identify a directory, and that the file handle does not have an associated reparse point.

Conclusion

Sandboxes are one of the most recent mitigation strategies deployed by browser and plug-in vendors. These vendors isolated their applications by implementing restricted permissions and the best practices available in the operating system. They followed this by greatly limiting the APIs available to the sandboxed process. Many hours have been spent auditing these APIs for memory corruption issues and logic errors. The primary purpose of all this work was to provide a clear separation between untrusted and trusted sections of code.

While sandboxes are used to test unverified programs which may contain a virus or malignant code without allowing harm to the host device [OR While sandboxes are used to execute software in a restricted OS environment], attackers have discovered many techniques to violate this trust boundary. There are the traditional approaches: find a memory corruption vulnerability in IPC message handling or attack the kernel to get SYSTEM-level privilege escalation. Any of these will work, but they may not be the easiest way. The examples in this paper demonstrate many of the uncommon, yet highly effective, approaches that have been used to bypass the most advanced application sandboxes in use today. Understanding them provides a unique perspective for those working to find and verify such bypasses.

Learn more at
hp.com/go/hpsrblog
zerodayinitiative.com