The Sandbox Roulette: Are you ready for the gamble?

Rafal Wojtczuk rafal@bromium.com

Rahul Kashyap rahul@bromium.com

26 February 2013

What is a sandbox?

In computer security terminology, a sandbox is an environment designed to run untrusted (or potentially exploitable) code in a manner that prevents the encapsulated code from damaging the rest of the system. The reason to introduce a sandbox is the assumption that it is more feasible to isolate potentially malicious code than to build a large application that cannot be subverted by an attacker.

Many different products can be categorized as sandboxes. For this talk, we focus on Windows-based application sandboxes. Such a sandbox is designed to run as a single application on top of a Windows OS. Usually, from the point of view of user experience, the existence of a sandbox container should be as unobtrusive as possible while still providing an additional layer of security. Short overviews of mechanisms used to implement a few selected sandboxes are presented later in this paper.

The inherent weakness of application sandboxes

The sandbox approach gained significant adoption in the industry recently. In some cases, after introducing the sandbox, the number of publicly available exploits against the application dropped dramatically. Despite this fact, in this paper we try to stress the main architectural weakness of Windows application sandboxes, namely their reliance on kernel security. We believe that the Windows kernel presents a broad attack surface to the skilled attacker, and a successful exploit against the kernel can totally breach sandbox defenses. This fact is hardly news – practically every good paper about a given sandbox implementation mentions briefly the possibility of a kernel-based attack. We postulate that this attack vector should be considered as a real threat, at least as likely to be exploited as other possible vectors (e.g. bugs in the sandbox implementation). Particularly, we have developed concrete examples of exploit code capable of breaking out of four different sandbox implementations, if run on an old unpatched version of Windows that contain known kernel vulnerabilities.

There is empirical evidence that vulnerabilities in sandbox implementations are relatively rare. In comparison, Windows kernel vulnerabilities are frequent. In 2012, 25 CVE entries were allocated for Windows kernel-mode components which resulted in privilege escalation to kernel mode. The Microsoft security bulletins for February 2013 name 30 CVE items for vulnerabilities in *win32k.sys* (a kernel mode component). Exploitation of kernel vulnerabilities is more complex than exploitation of a user mode application and requires more knowledge on the details of the OS implementation. This is presumably the reason why malware authors have not yet used this attack vector widely (with Duqu malware [1] being the notable exception). Still, the number of Windows kernel vulnerabilities announced each month justifies our estimate that this attack vector should be considered as the dominant one, and may be more actively exploited in the near future. It means that each time a vulnerability is announced in the Windows kernel, sandbox users may be awaiting the equivalent of a roulette spin – was this vulnerability known before and was it exploited to breach my sandbox?

Sandbox Type 1: OS enhancement based sandbox

The main concept

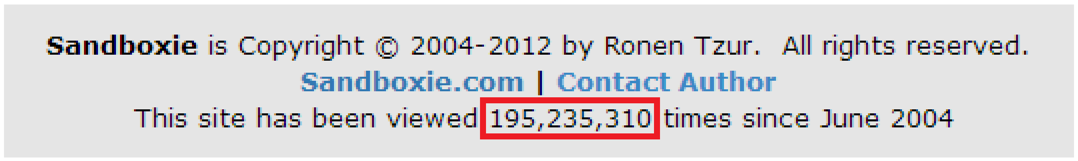
In this approach, there is a custom kernel driver installed that modifies Windows behavior for a sandboxed application in order to provide additional protection. For instance, if a sandboxed application tries to terminate a non-sandboxed process, such a request will be denied. If a sandboxed application tries to write to a file that is protected (the protected files are defined in the sandbox configuration), then the driver can transparently create a copy of the file that the sandboxed application can write to, while the original file is untouched. Such a copy-on-write mechanism allows the unchanged sandboxed application to work properly, while preventing damage or compromise to the rest of the system.

The main weakness

This design is focused on preventing access to the protected resources and if implemented exhaustively this goal can be achieved. However it does not significantly limit the possibilities of exploiting a kernel vulnerability – it allows access to most kernel functionality. The reason is the need to allow a broad range of applications to run in a sandbox which may require access to many kernel interfaces. Once the attacker has gained arbitrary code execution in kernel mode, he can completely bypass the sandbox, an application sandbox has no means to protect itself against code running at the same privilege level.

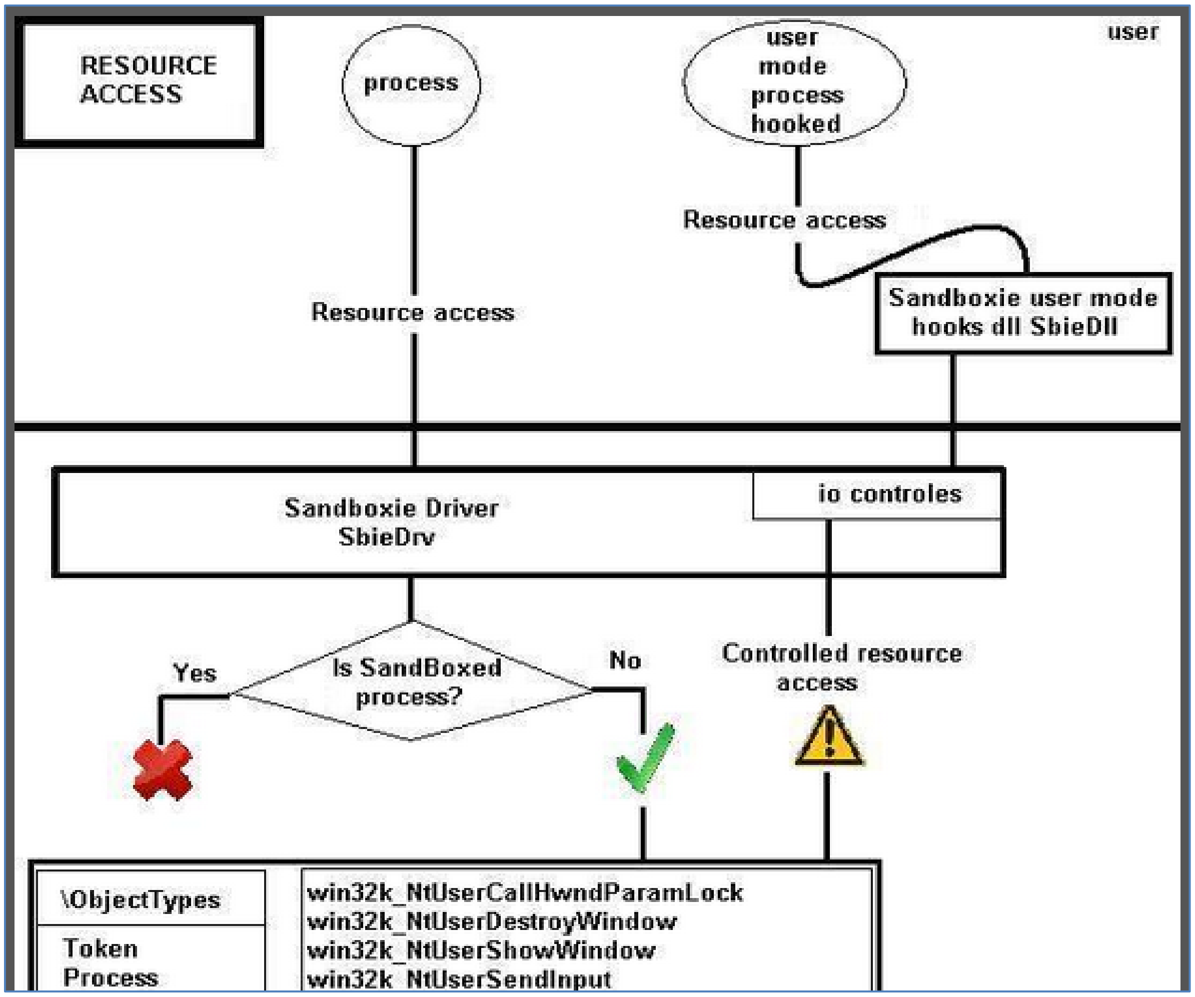
Type 1 Sandbox example: Sandboxie

Sandboxie [2] is a popular tool:


Sandboxie is Copyright © 2004-2012 by Ronen Tzur. All rights reserved.
Sandboxie.com | Contact Author
This site has been viewed 195,235,310 times since June 2004

It is widely used for malware analysis, and generally for controlled execution of applications.

Unfortunately, Sandboxie documentation does not describe the details of the implementation. The following diagram shows the main components of Sandboxie; note this diagram was copied from an unrelated site [3], so that it may be incomplete or outdated.



Due to the differences in *x86_32* and *x86_64* Windows kernels, Sandboxie's method of hooking crucial operations differs between these platforms. The documentation states that the "experimental" Sandboxie mode on *x86_64* provides nearly the same level of protection as in *x86_32* case.

We tested the exploit for the CVE-2012-0217 vulnerability (misleadingly named "User Mode Scheduler Memory Corruption" in Microsoft bulletin MS12-042) in the Sandboxie environment on the *x86_64* platform running the default Windows 7 SP1 kernel. This vulnerability is caused by not sanitizing the return address of a system call; the non-canonical return address results in an unexpected exception being raised that is handled incorrectly. The exploit worked flawlessly, providing the ability to run arbitrary code in kernel mode. A slight twist is that the usual kernel shellcode that just steals the SYSTEM access token is not particularly useful to the attacker. Although the attacker gains SYSTEM rights in his user mode process, the process is still confined by Sandboxie (and an attempt to kill a non-sandboxed process from this SYSTEM shell fails). The attacker needs to perform some extra work while in kernel mode, either:

1) Disable the Sandboxie driver (uninstall hooks, or just overwrite the driver code)

2) Migrate to another process that runs outside of the Sandboxie container

We chose the second method, because it is more generic. The required steps are:

1) Allocate kernel memory for exploit_syscall_handler() function
2) Hook all system calls via overwriting LSTAR MSR, (LSTAR:= exploit_syscall_handler)
3) When the exploit_syscall_handler() function detects it is running in the context of a process running outside of the sandbox, inject arbitrary shellcode into this process

The result is that after the exploit is run within a sandbox, the attacker can execute arbitrary code in the context of an arbitrary process. Careful readers certainly recognize that this procedure is similar to how remote kernel exploits (that often initially execute with raised IRQL) migrate to a stable user mode environment.

To sum up, we have verified that by exploiting the CVE-2012-0217 vulnerability, a Sandboxie-confined process can completely bypass Sandboxie protection.

Type 1 Sandbox example: BufferZone Pro [4]

This popular sandbox is similar in concept to Sandboxie. Naturally some implementation details are different – e.g. by default, it prevents read access to selected directories (to block data theft). The same CVE-2012-0217 shellcode-migrating exploit can be used to fully bypass BufferZone Pro protection running on unpatched Windows 7 SP1 on *x86_64* platform.

Sandbox type 2: Master/Slave sandbox

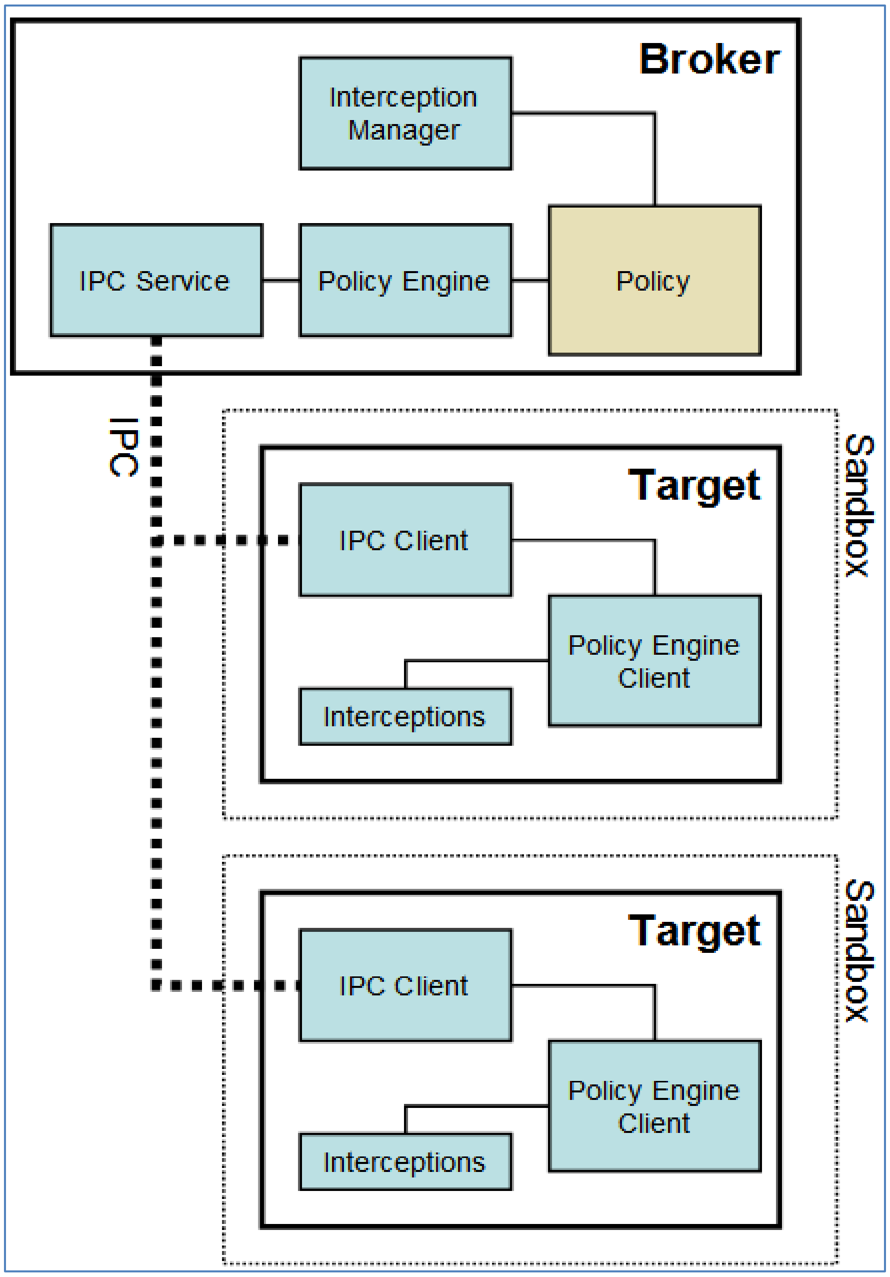
The main concept

In this approach, the application is split into two processes. The first one (Slave) runs with very low privilege (so, it is confined using the OS access control facilities); it is able to access almost none of existing OS resources. Whenever the Slave needs a particular resource (e.g. read a file), it sends a message to the other process (Master) via some IPC channel. Master validates the request and if is legit, performs the required operation on behalf of the Slave.

The idea is that the vast majority of the application code stays in Slave. If Slave is compromised, the OS prevents it from directly accessing any valuable resource. Master's codebase is small, therefore it should not have many bugs, thus it should be difficult for the compromised Slave to compromise the Master.

Type 2 Sandbox example: Chrome sandbox

The following diagram, borrowed from Chrome sandbox documentation [5], shows the main components:



The slave indeed runs with low privileges:

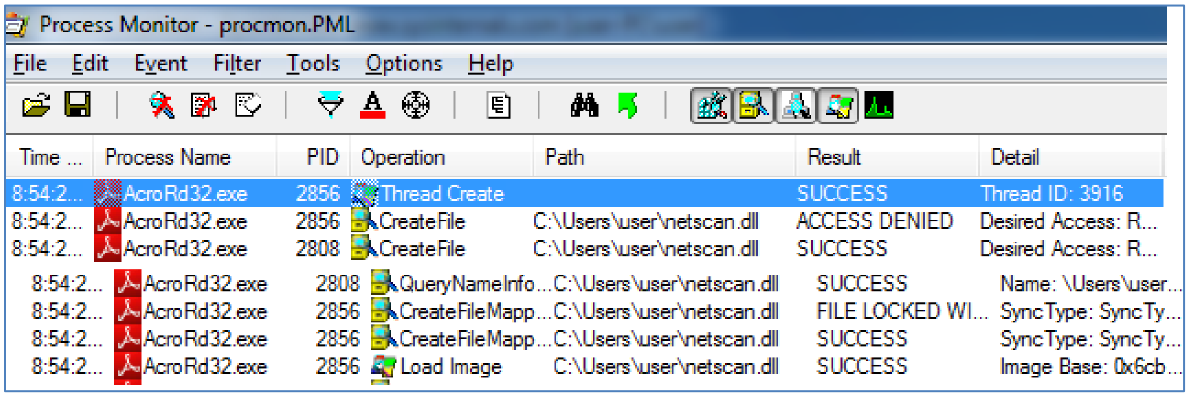
- Restricted token
- Job object
- Desktop object
- Lowered integrity level

According to the documentation, this privilege allows for very few operations:

- Access to mounted FAT or FAT32 volumes
- TCP/IP

- The rest of the existing resources (e.g. all files on an NTFS filesystem) cannot be accessed directly.

Let's see an example of how this sandbox operates:

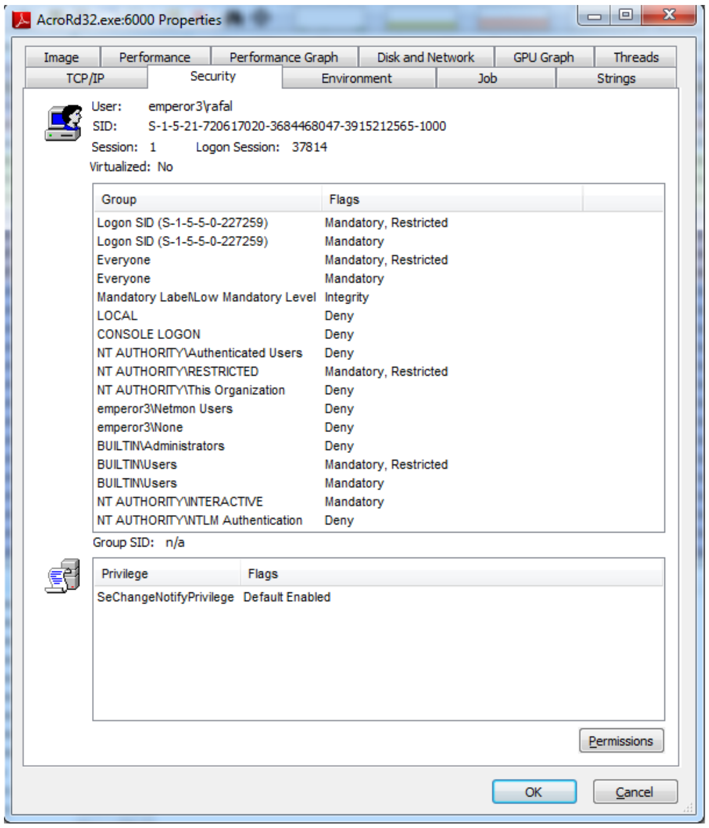


Process Monitor - procmon.PML
File Edit Event Filter Tools Options Help

Time ...	Process Name	PID	Operation	Path	Result	Detail
8:54:2...	AcroRd32.exe	2856	Thread Create		SUCCESS	Thread ID: 3916
8:54:2...	AcroRd32.exe	2856	CreateFile	C:\Users\user\netscan.dll	ACCESS DENIED	Desired Access: R...
8:54:2...	AcroRd32.exe	2808	CreateFile	C:\Users\user\netscan.dll	SUCCESS	Desired Access: R...
8:54:2...	AcroRd32.exe	2808	QueryNameInfo...	C:\Users\user\netscan.dll	SUCCESS	Name: \Users\user...
8:54:2...	AcroRd32.exe	2856	CreateFileMapp...	C:\Users\user\netscan.dll	FILE LOCKED WI...	SyncType: SyncTy...
8:54:2...	AcroRd32.exe	2856	CreateFileMapp...	C:\Users\user\netscan.dll	SUCCESS	SyncType: SyncTy...
8:54:2...	AcroRd32.exe	2856	Load Image	C:\Users\user\netscan.dll	SUCCESS	Image Base: 0x6cb...

In the above picture, process 2856 is the Slave. It tried to access a file, and OS denied this request. The Slave sends a message "open a file for me please" to the Master (unfortunately, Process Monitor does not show named pipes activity). The Master (process 2808) opens the file, and duplicates the handle into the Slave process. Later, Slave can access this file via the received handle.

Both Chrome browser and Adobe Acrobat Reader use the Chrome sandbox. The actual implementation details are slightly different. The following screenshot shows the Adobe Reader Slave access token details:



User: emperor3\rafal
SID: S-1-5-21-720617020-3684468047-3915212565-1000
Session: 1 Logon Session: 37814
Virtualized: No

Group	Flags
Logon SID (S-1-5-5-0-227259)	Mandatory, Restricted
Logon SID (S-1-5-5-0-227259)	Mandatory
Everyone	Mandatory, Restricted
Everyone	Mandatory
Mandatory Label\Low Mandatory Level	Integrity
LOCAL	Deny
CONSOLE LOGON	Deny
NT AUTHORITY\Authenticated Users	Deny
NT AUTHORITY\RESTRICTED	Mandatory, Restricted
NT AUTHORITY\This Organization	Deny
emperor3\Netmon Users	Deny
emperor3\None	Deny
BUILTIN\Administrators	Deny
BUILTIN\Users	Mandatory, Restricted
BUILTIN\Users	Mandatory
NT AUTHORITY\INTERACTIVE	Mandatory
NT AUTHORITY\NTLM Authentication	Deny

Group SID: n/a

Privilege	Flags
SeChangeNotifyPrivilege	Default Enabled

The following screenshots show the Chrome browser Slave (named "renderer" in Chrome documentation) access token details:

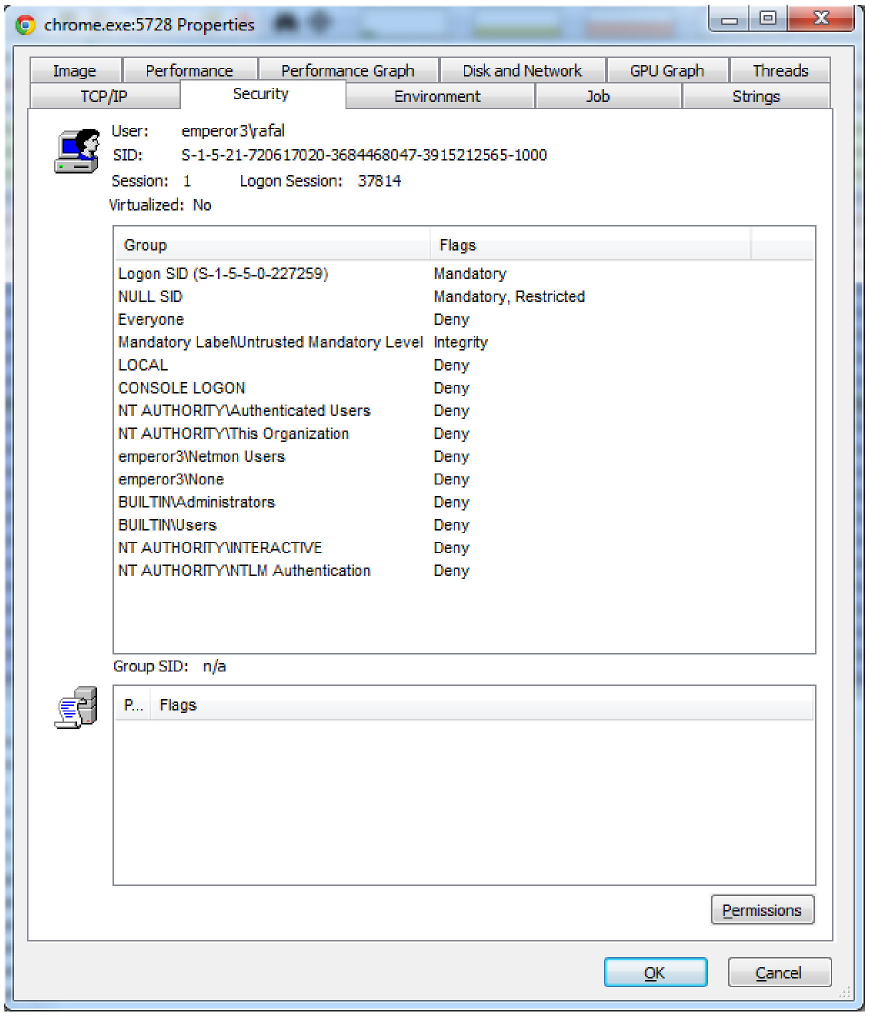

chrome.exe:5728 Properties

Image | Performance | Performance Graph | Disk and Network | GPU Graph | Threads
TCP/IP | Security | Environment | Job | Strings

User: emperor3\rafal
SID: S-1-5-21-720617020-3684468047-3915212565-1000
Session: 1 Logon Session: 37814
Virtualized: No

Group	Flags
Logon SID (S-1-5-5-0-227259)	Mandatory
NULL SID	Mandatory, Restricted
Everyone	Deny
Mandatory Label\Untrusted Mandatory Level	Integrity
LOCAL	Deny
CONSOLE LOGON	Deny
NT AUTHORITY\Authenticated Users	Deny
NT AUTHORITY\This Organization	Deny
emperor3\Netmon Users	Deny
emperor3\None	Deny
BUILTIN\Administrators	Deny
BUILTIN\Users	Deny
NT AUTHORITY\INTERACTIVE	Deny
NT AUTHORITY\NTLM Authentication	Deny

Group SID: n/a

P...	Flags

One immediately visible difference is the integrity level – "low" in Acrobat Reader, "untrusted" in Chrome browser. Notably, the Chrome browser Slave is not permitted by the OS to access FAT32 file systems (despite the explicit remark in the documentation), while the Adobe Reader Slave can access it freely. More important differences are in the policies enforced by the Masters:

1) In comparison with Chrome browser , Adobe Reader Master allows many more registry accesses
2) In comparison with Chrome browser, Adobe Reader Master allows many more file system accesses (e.g. any .dll file can be opened read-only).

While it is straightforward to use TCP/IP networking in the context of the Adobe Acrobat Slave (it is possible to just load the winsock library), it is not so in the context of the Chrome browser Slave. It has not been determined yet whether the Chrome browser Slave has sufficient privileges to actually create TCP/IP sockets (although it was verified that it can open the *Device*\Afd\Endpoint file, which is crucial in the process of creating a socket). The ability to connect to arbitrary ports on the local machine and the corporate network constitutes a possible escalation vector via exploiting a weakness in the accessed service.

The main weakness

Both the Chrome browser and the Adobe Reader sandboxes have a good vulnerability record. The first case of an Adobe Reader sandbox vulnerability being exploited in the wild was reported in February 2013 [6] (the escape was possible because of a bug in the Master). Similarly, the attacks against the Chrome browser Master are rare ([7], [8]). It is fair to state that if the vulnerabilities in the Master were the only problem, then due to the fact that the Master has a small code base, this type of sandbox would provide good protection.

However, this type of sandbox (when run on Windows) reduces the attack exposure of the underlying kernel to an unsatisfactory degree. While some of the operations that may be necessary for some exploits are not allowed, e.g.

- The job object prevents creation of processes
- Registry keys normally accessible by unprivileged users are not available

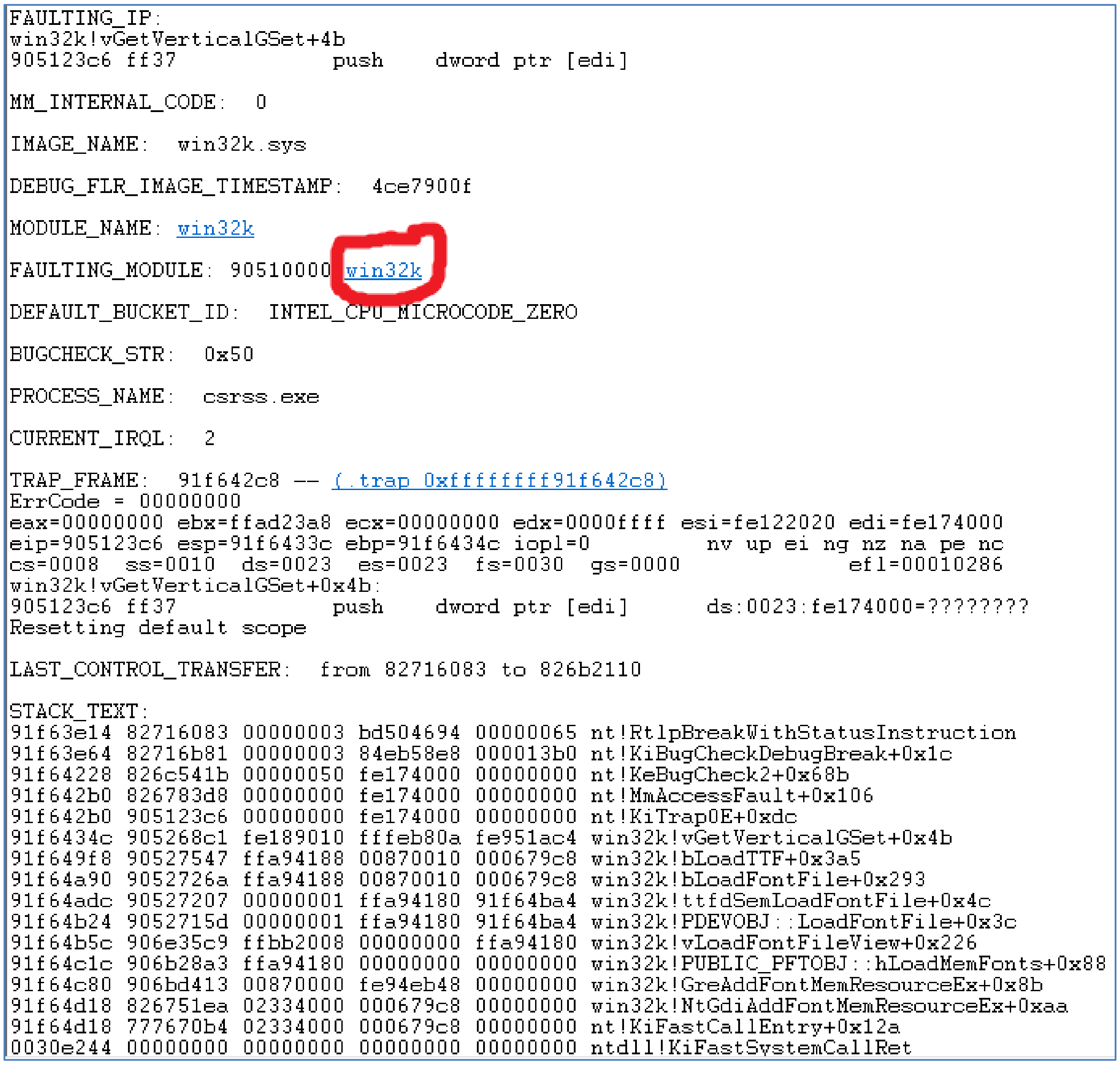
Many Windows kernel interfaces are still available for use by the Slave.

Note that on a Linux platform, there is an explicit mechanism available designed to reduce the kernel exposure. By using seccomp-bpf syscall filter [9], it is possible to significantly trim the access to functionality that is normally unneeded, but can contain exploitable vulnerabilities. The Chrome browser on Linux uses the seccomp-bpf syscall filter feature; unfortunately, equivalent functionality is not available in Windows.

In the Windows OS case, the largest potentially vulnerable kernel component, namely the graphics subsystem implemented in win32k.sys driver, is mostly accessible for the Chrome sandboxed process. While it is true that interactions with the windows of other processes is prevented, the sandboxed process can still create new windows on its own and exploit bugs related to the graphics support code.

CVE-2012-2897 example

This vulnerability (patched in MS12-075 Microsoft security bulletin) is related to a bug in true type fonts handling in the Windows kernel, namely an integer overflow in the kernel code responsible for parsing the *cmap* true type font table. It is enough to just open a crafted web page in a vulnerable Chrome browser running on a vulnerable Windows version to trigger the vulnerability. The following screenshot shows the kernel debugger backtrace:



```
FAULTING_IP:
win32k!vGetVerticalGSet+4b
905123c6 ff37            push    dword ptr [edi]

MM_INTERNAL_CODE:  0

IMAGE_NAME:  win32k.sys

DEBUG_FLR_IMAGE_TIMESTAMP:  4ce7900f

MODULE_NAME: win32k

FAULTING_MODULE: 90510000 win32k

DEFAULT_BUCKET_ID:  INTEL_CPU_MICROCODE_ZERO

BUGCHECK_STR:  0x50

PROCESS_NAME:  csrss.exe

CURRENT_IRQL:  2

TRAP_FRAME:  91f642c8 -- (.trap 0xffffffff91f642c8)
ErrCode = 00000000
eax=00000000 ebx=ffad23a8 ecx=00000000 edx=0000ffff esi=fe122020 edi=fe174000
eip=905123c6 esp=91f6433c ebp=91f6434c iopl=0         nv up ei ng nz na pe nc
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=00010286
win32k!vGetVerticalGSet+0x4b:
905123c6 ff37            push    dword ptr [edi]      ds:0023:fe174000=????????
Resetting default scope

LAST_CONTROL_TRANSFER:  from 82716083 to 826b2110

STACK_TEXT:
91f63e14 82716083 00000003 bd504694 00000065 nt!RtlpBreakWithStatusInstruction
91f63e64 82716b81 00000003 84eb58e8 000013b0 nt!KiBugCheckDebugBreak+0x1c
91f64228 826c541b 00000050 fe174000 00000000 nt!KeBugCheck2+0x68b
91f642b0 826783d8 00000000 fe174000 00000000 nt!MmAccessFault+0x106
91f642b0 905123c6 00000000 fe174000 00000000 nt!KiTrap0E+0xdc
91f6434c 905268c1 fe189010 fffeb80a fe951ac4 win32k!vGetVerticalGSet+0x4b
91f649f8 90527547 ffa94188 00870010 000679c8 win32k!bLoadTTF+0x3a5
91f64a90 9052726a ffa94188 00870010 000679c8 win32k!bLoadFontFile+0x293
91f64adc 90527207 00000001 ffa94180 91f64ba4 win32k!ttfdSemLoadFontFile+0x4c
91f64b24 9052715d 00000001 ffa94180 91f64ba4 win32k!PDEVOBJ::LoadFontFile+0x3c
91f64b5c 906e35c9 ffbb2008 00000000 ffa94180 win32k!vLoadFontFileView+0x226
91f64c1c 906b28a3 ffa94180 00000000 00000000 win32k!PUBLIC_PFTOBJ::hLoadMemFonts+0x88
91f64c80 906bd413 00870000 fe94eb48 00000000 win32k!GreAddFontMemResourceEx+0x8b
91f64d18 826751ea 02334000 000679c8 00000000 win32k!NtGdiAddFontMemResourceEx+0xaa
91f64d18 777670b4 02334000 000679c8 00000000 nt!KiFastCallEntry+0x12a
0030e244 00000000 00000000 00000000 00000000 ntdll!KiFastSystemCallRet
```

In this case, it is even not necessary to compromise the renderer first – it is enough to just pass it a crafted .ttf file reference in a .html page. However, in order to achieve code execution reliably, more control over the kernel memory layout is needed; for this, the ability to run arbitrary code in the renderer is very helpful.

CVE-2011-3042 example

This vulnerability (patched in MS11-087 Microsoft security bulletin) is also related to true type fonts parsing. The root cause is an insecure destination buffer address calculation when processing a malformed compound bitmap stored in the true type font file. It was exploited in the wild by Duqu malware [1], via malformed MS Office documents. We have verified that on Windows SP1 the host Slave is able to exploit this vulnerability and achieve arbitrary code execution in kernel mode. The typical token-stealing shellcode is all the attacker needs in this case. The following screenshot shows the state of the Adobe Reader Slave after successful exploitation:

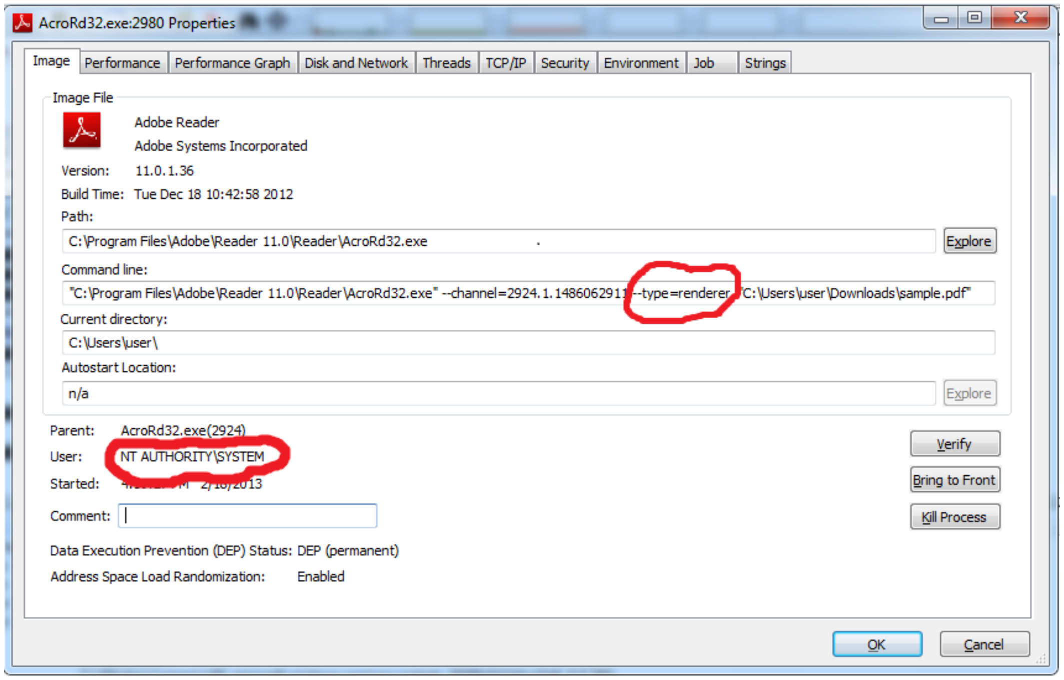

AcroRd32.exe:2980 Properties

Image | Performance | Performance Graph | Disk and Network | Threads | TCP/IP | Security | Environment | Job | Strings

Image File

Adobe Reader
Adobe Systems Incorporated
Version: 11.0.1.36
Build Time: Tue Dec 18 10:42:58 2012
Path:
C:\Program Files\Adobe\Reader 11.0\Reader\AcroRd32.exe
Command line:
"C:\Program Files\Adobe\Reader 11.0\Reader\AcroRd32.exe" --channel=2924.1.1486062911 --type=renderer "C:\Users\user\Downloads\sample.pdf"
Current directory:
C:\Users\user\
Autostart Location:
n/a

Parent: AcroRd32.exe(2924)
User: NT AUTHORITY\SYSTEM
Started: [illegible] 2/18/2013
Comment:

Data Execution Prevention (DEP) Status: DEP (permanent)
Address Space Load Randomization: Enabled

The following screenshot shows the state of Chrome browser Slave after successful exploitation:

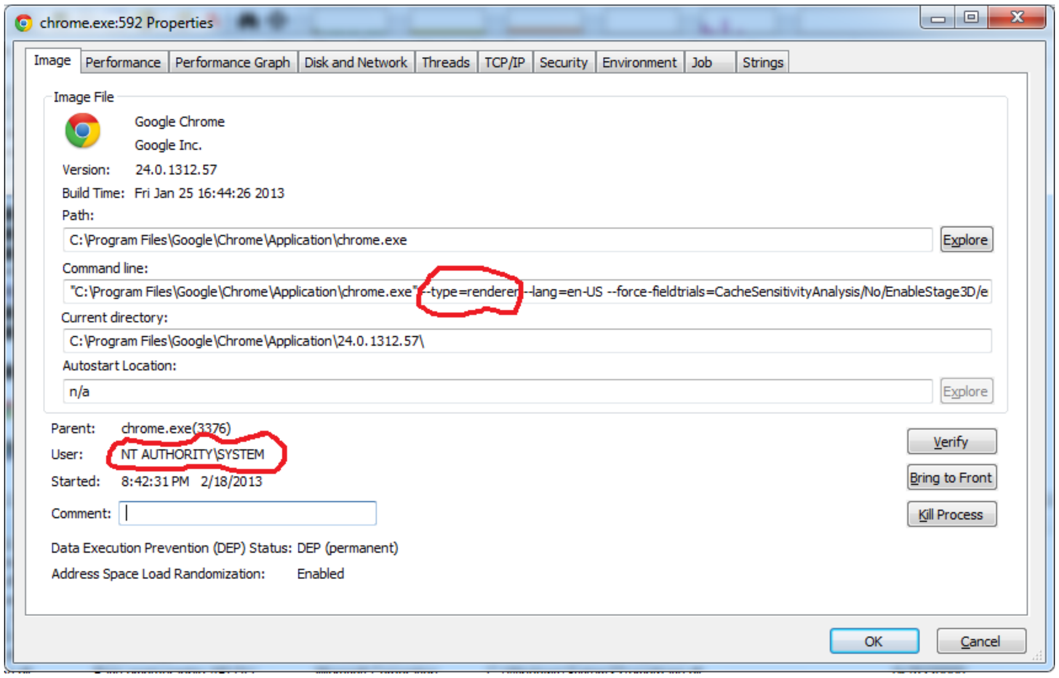

chrome.exe:592 Properties

Image | Performance | Performance Graph | Disk and Network | Threads | TCP/IP | Security | Environment | Job | Strings

Image File

Google Chrome
Google Inc.
Version: 24.0.1312.57
Build Time: Fri Jan 25 16:44:26 2013
Path:
C:\Program Files\Google\Chrome\Application\chrome.exe
Command line:
"C:\Program Files\Google\Chrome\Application\chrome.exe" --type=renderer --lang=en-US --force-fieldtrials=CacheSensitivityAnalysis/No/EnableStage3D/e
Current directory:
C:\Program Files\Google\Chrome\Application\24.0.1312.57\
Autostart Location:
n/a

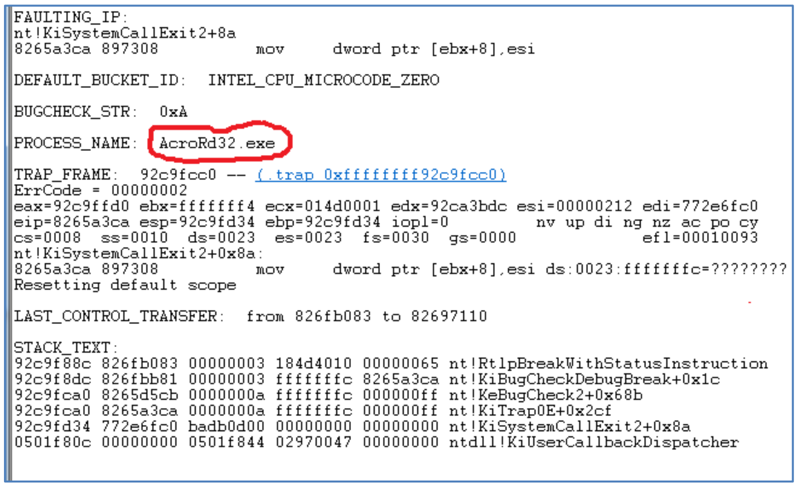
Parent: chrome.exe(3376)
User: NT AUTHORITY\SYSTEM
Started: 8:42:31 PM 2/18/2013
Comment:

Data Execution Prevention (DEP) Status: DEP (permanent)
Address Space Load Randomization: Enabled

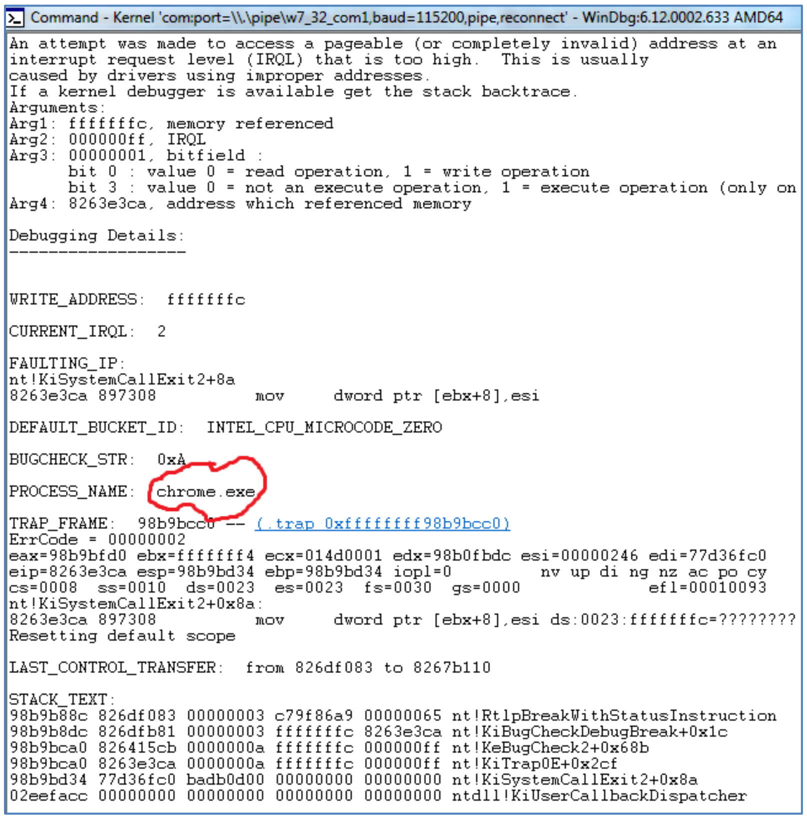
It is apparent that after the exploit has succeeded, the Slave runs with the highest user mode privilege, and is capable of taking full control over the OS.

CVE-2011-2018 example

Not just the graphics subsystem vulnerabilities can be triggered from a compromised Slave. CVE-2011-2018, misleadingly referred to as “Windows Kernel Exception handler Vulnerability” in the MS11-098 bulletin is caused by treating a particular segment selector value as special which is not allowed to be set by user mode (this assumption is wrong). In fact, using an alternate and unexpected code path when performing the return from a system call when this special value is seen is possible . This vulnerability can be used to cause at least a BSOD when exploited from within the Chrome browser or Adobe Reader sandbox:



```
FAULTING_IP:
nt!KiSystemCallExit2+8a
8265a3ca 897308          mov     dword ptr [ebx+8],esi

DEFAULT_BUCKET_ID:  INTEL_CPU_MICROCODE_ZERO

BUGCHECK_STR:  0xA

PROCESS_NAME:  AcroRd32.exe

TRAP_FRAME:  92c9fcc0 -- (.trap 0xffffffff92c9fcc0)
ErrCode = 00000002
eax=92c9ffd0 ebx=fffffff4 ecx=014d0001 edx=92ca3bdc esi=00000212 edi=772e6fc0
eip=8265a3ca esp=92c9fd34 ebp=92c9fd34 iopl=0         nv up di ng nz ac po cy
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=00010093
nt!KiSystemCallExit2+0x8a:
8265a3ca 897308          mov     dword ptr [ebx+8],esi ds:0023:fffffffc=????????
Resetting default scope

LAST_CONTROL_TRANSFER:  from 826fb083 to 82697110

STACK_TEXT:
92c9f88c 826fb083 00000003 184d4010 00000065 nt!RtlpBreakWithStatusInstruction
92c9f8dc 826fbb81 00000003 fffffffc 8265a3ca nt!KiBugCheckDebugBreak+0x1c
92c9fca0 8265d5cb 0000000a fffffffc 000000ff nt!KeBugCheck2+0x68b
92c9fca0 8265a3ca 0000000a fffffffc 000000ff nt!KiTrap0E+0x2cf
92c9fd34 772e6fc0 badb0d00 00000000 00000000 nt!KiSystemCallExit2+0x8a
0501f80c 00000000 0501f844 02970047 00000000 ntdll!KiUserCallbackDispatcher
```



Command - Kernel 'com:port=\\.\pipe\w7_32_com1,baud=115200,pipe,reconnect' - WinDbg:6.12.0002.633 AMD64

```
An attempt was made to access a pageable (or completely invalid) address at an
interrupt request level (IRQL) that is too high.  This is usually
caused by drivers using improper addresses.
If a kernel debugger is available get the stack backtrace.
Arguments:
Arg1: fffffffc, memory referenced
Arg2: 000000ff, IRQL
Arg3: 00000001, bitfield :
      bit 0 : value 0 = read operation, 1 = write operation
      bit 3 : value 0 = not an execute operation, 1 = execute operation (only on
Arg4: 8263e3ca, address which referenced memory

Debugging Details:
------------------


WRITE_ADDRESS:  fffffffc

CURRENT_IRQL:  2

FAULTING_IP:
nt!KiSystemCallExit2+8a
8263e3ca 897308          mov     dword ptr [ebx+8],esi

DEFAULT_BUCKET_ID:  INTEL_CPU_MICROCODE_ZERO

BUGCHECK_STR:  0xA

PROCESS_NAME:  chrome.exe

TRAP_FRAME:  98b9bcc0 -- (.trap 0xffffffff98b9bcc0)
ErrCode = 00000002
eax=98b9bfd0 ebx=fffffff4 ecx=014d0001 edx=98b0fbdc esi=00000246 edi=77d36fc0
eip=8263e3ca esp=98b9bd34 ebp=98b9bd34 iopl=0         nv up di ng nz ac po cy
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=00010093
nt!KiSystemCallExit2+0x8a:
8263e3ca 897308          mov     dword ptr [ebx+8],esi ds:0023:fffffffc=????????
Resetting default scope

LAST_CONTROL_TRANSFER:  from 826df083 to 8267b110

STACK_TEXT:
98b9b88c 826df083 00000003 c79f86a9 00000065 nt!RtlpBreakWithStatusInstruction
98b9b8dc 826dfb81 00000003 fffffffc 8263e3ca nt!KiBugCheckDebugBreak+0x1c
98b9bca0 826415cb 0000000a fffffffc 000000ff nt!KeBugCheck2+0x68b
98b9bca0 8263e3ca 0000000a fffffffc 000000ff nt!KiTrap0E+0x2cf
98b9bd34 77d36fc0 badb0d00 00000000 00000000 nt!KiSystemCallExit2+0x8a
02eefacc 00000000 00000000 00000000 00000000 ntdll!KiUserCallbackDispatcher
```

A note on methodology

In our experiments with type 2 sandboxes, we did not want to spend time on developing exploits that are able to get initial code execution in the context of Slave. After all, the core sandbox concept is that we expect vulnerabilities in the Slave to be present and exploitable. The changelogs of both Chrome browser and Adobe Reader confirm there is a noticeable number of coding errors fixed in the Slaves in each software release. Therefore, we have simulated just the second stage of an exploit, when an attacker is able to run arbitrary code in the context of Slave and tries to bypass the sandbox. We run the kernel exploits in the context of Slaves by manually copying the exploit code via *WriteProcessMemory* and trigger them via *CreateRemoteThread*. It is quite apparent that shellcode triggered by parsing malicious .html or .pdf document could perform the same actions and breach the sandbox.

A short note on hardware-virtualization based sandboxes

These kinds of sandboxes take a different approach – they wrap not just a single application, but the whole Windows OS with an application in it within a virtual container enforced by the hardware based VTx capabilities of the chip set. Thanks to this design, vulnerabilities in the OS are nonfatal – even if an attacker achieves arbitrary code execution in kernel mode, he is still confined by the hardware enforced virtual "sandbox", because the latter runs in a more privileged mode that the sandboxed kernel.

While an appealing concept, this approach is not straightforward. The hypervisor and the supporting environment introduce a new attack vector; possible vulnerabilities in the hypervisor can be used to fully bypass this type of isolation. Unfortunately, most popular hypervisor solutions are focused on maximal functionality, not on security. A customized virtualization solution built specifically for security isolation is required to limit this type of exposure. The amount of functionality exposed by a hardened hypervisor to the attacker, although not negligible, is orders of magnitude less than the equivalent Windows OS code – and consequently it is much less prone to attacks.

Bibliography

1) Duqu malware, http://em.wikipedia.org/wiki/Duqu
2) Sandboxie, http://www.sandboxie.com
3) Javier Vicente Vallejo, Sandboxie analysis, http://vallejo.cc/48
4) BufferZone Pro, http://www.trustware.com/BufferZone-Pro/
5) http://dev.chromium.org/developers/design-documents/sandbox
6) https://krebsonsecurity.com/2012/11/experts-warn-of-zero-day-exploit-for-adobe-reader/
7) http://blog.chromium.org/2012/10/pwnium-2-results-and-wrap-up_10.html
8) "Pwn2Own 2012: Google Chrome browser sandbox first to fall" http://www.zdnet.com/blog/security/pwn2own-2012-google-chrome-browser-sandbox-first-to-fall/10588
9) http://blog.chromium.org/2012/11/a-safer-playground-for-your-linux-and.html