Escaping The Sandbox
(Summer 2010)
SyScan-EuSecWest-ReCon

Stephen A. Ridley
Senior Researcher Matasano Security

stephen@sa7ori.org
@s7ephen (Twitter)
Who I am.

Stephen A. Ridley
Senior Security Researcher (Matasano)

• **Previously:** Senior Security Architect at McAfee, founding member of Security Architecture Group

• **Prior to that:** Researcher at leading Defense contractor. Directly supported U.S. Defense and Intelligence communities in realm of software exploitation and software reverse engineering

• Columnist for/interviewed by IT magazines (Wired, Ping!, Washington Post)

• Kenshoto DefCon CTF organizers for a few years

• blog: [http://www.dontstuffbeansupyourownose.com](http://www.dontstuffbeansupyourownose.com)

• Guest Lecturer/Instructor (New York University, Netherlands Forensics Institute, Department of Defense, Google, et al)

• **My Focus:** software reverse engineering, software development, software exploitation, software security, Kernels (Microsoft ones for now). Increasingly interested in embedded systems and mobile devices
What am I talkin’ ‘bout today?

★ Sandboxing Overview (very brief ;-)  
  • Goals, Sandbox Architecture (Chrome)

★ Sandboxes from a User-space Perspective  
  • Securable Objects and SID apertures  
  • Patches/Hooks/Interception  
  • user32 issues

★ Sandboxes from a Kernel-space Perspective  
  • Between User-space and Kernel-space  
  • Kernel supported “Quasi Securable Objects”, Native API  
  • Job Objects handle the rest, or do they?

★ Tools/Techniques/Demos  
  • SandKit Toolkit (code injection, copymem, memdiff, hookfix, sa7shell, bincompare, dumptoken, tokenbrute, handlebrute)  
  • Using Sandbox PoC Project (from Google)  
  • Using kernel debugger while attacking Chrome  
  • Triggering Chrome Bugs and where to start
Sandbox implementations are (by their nature) strongly coupled to the Operating System.

This presentation focuses on Microsoft Windows Operating Systems and the NT Kernel (XP and Vista).

- **Side Note**: Check out OSX’s DAC/Sandbox. (“man sandbox-exec”, “ls /usr/share/sandbox”) It’s pretty awesome! Scheme-like rules sent to a DAC engine with a Scheme-like interpreter in the Kernel! Nice idea!

This presentation uses Google Chromium because it’s the most popular of the Sandbox implementations.

Focus on blackbox/reversing approach to sandboxing technologies (less source source audit of IPC mechanisms, etc). For that approach see Azimuth Security’s excellent “The Chrome Sandbox” series.

Thursday, July 15, 2010
Sandboxing Overview
The Goal of the Sandbox

- Localize the damage by “containing” potentially malicious code

- Trapping malicious code is nuanced and tough but from a high level it consists mostly of:
  - Locking down all IPC mechanisms
  - Perform process monitoring
  - Basically not trusting any code within the Sandbox to do anything on the system without it first being checked by some authority
Chromium Sandbox Architecture

- A great number of resources currently exist on the architecture and design of sandboxes in general, especially for Google Chromium. Not going to echo-chamber.

- Mark Dowd and the team at Azimuth Security began releasing Sandboxing papers that happened to coincide with my talk and paper: [http://blog.azimuthsecurity.com/2010/05/chrome-sandbox-part-1-of-3-overview.html](http://blog.azimuthsecurity.com/2010/05/chrome-sandbox-part-1-of-3-overview.html)


- The Chromium Design Docs are all you really need, but other small bits can be gleaned from Infosec bloggers and research papers (Robert Hensing, David Leblanc, Nicolas Sylvain, and others). Not much *actual* code/tools/techniques/examples have been released though, this talk hopefully will help with this.
Chromium Sandbox Architecture
The Operating System is what does all the “hard work” for permissions and restrictions. Developers don’t need to reinvent this technology these days.

In the NT Kernel this is handled by using the DACL system built into the **Object Manager** and **Security Reference Manager**

These two components of the NT kernel implement and enforce the permissions system for “NT Secureable Objects” like:

- Files
- Processes
- Shared Memory Regions
- Lots more...
Locking down IPC and IO

★ IO and IPC on Windows is performed predominantly using these NT Objects. I really realized this more, the more kernel stuff I began doing.

★ “Almost everything in userspace is an NT Object, or is at some point supported by one.”....but there are still gaps. “Quasi-securable Objects”

★ Most of the functionality for interfacing with/manipulating these NT Objects is implemented within the Native API
  • Think: OpenFile, OpenProcess, CreateFile, CreateProcess, CreateThread, or basically anything in ntdll or kernel32)

★ There are some other public techniques for performing faux-IPC. (we will review these and some less popular ideas/techniques)
Sandboxes from a User-space Perspective
As malicious code, what would you try first?

- Accessing Out of Proc COM Servers?
- Accessing WMI Interfaces?
- Writeable locations on the disk?
- Injecting into Other processes (reading/writing other process memory)?
- Loading Drivers?
- Accessing LPC/RPC/LRPC endpoints?
- Accessing NamedPipes?
- Accessing RunAs Service?
- sending User32 messages?
- ...lots of other stuff?
- Let’s See Why Most of this Won’t Work!
These things are all good places to start. In fact we will demonstrate a new tool in the SandKit that you can use to assist with these kinds of tests. In other implementations you will mostly likely find bugs here.

HOWEVER, virtually all of these operations under the hood are (or are supported by) Securable Objects which fall under the purview of the Object Manager and Security Reference Manager.

Therefore, the proper restrictions on security descriptors will kill access to these in one fell swoop!
As malicious code, what would you try first?

- Accessing Out of Proc COM Servers?
- Accessing WMI Interfaces?
- Writeable locations on the disk?
- Injecting into Other processes (reading/writing other process memory)?
- Loading Drivers?
- Accessing LPC/RPC/LRPC endpoints?
- Accessing Named Pipes?
- Accessing RunAs Service?
- Sending User32 messages?

**COM is Named Pipes**

**WMI is COM which is LPC/Named Pipes**

Handles and IO objects are securable.

**Processes/Threads/IO objects are securable.**

**SCManager is all LPC/Named Pipes also**

“Load Driver” token perm covers it

**LPC/LRPC/RPC sit on Named Pipes**

**Named Pipes are obviously securable**

**ShellExecuteA(“runas”) is LPC/NamedPipe to LSASS**

User32 partitioned by “Desktop” and user32 handles are restricted by Job Object (XP) UAC (Vista)
Bootstrapping the Sandbox

“The beginning is a very delicate time...”
Frank Herbert’s Dune

★ The Broker starts all the Sandbox processes.

★ The “Broker” process is the Overseer, he starts the “Sandbox” processes.

★ The Broker performs “privileged” actions on behalf of Sandbox processes via code hooks and IPC mechanisms.

★ Let’s review the steps the Broker goes through when bootstrapping the Sandbox.
Bootstrapping the Sandbox

1. Before spawning Sandbox, the Broker process creates a restricted token using: CreateRestrictedToken() with the ‘SidsToRestrict’ array populated.

2. The Broker uses CreateProcess() with the fdwCreate argument set to CREATE_SUSPENDED and the restricted token to start sandbox “frozen”.

3. It is during this suspended time that the Broker then further restricts the Sandbox process by:
   1. Installing hooks (we will review these shortly)
   2. Performing some other setup

We’ll see later that the Broker also continues to “debug” the Sandbox process, catching his exceptions! Annoying for your fuzzing huh? ;-)
4. The Broker further adjusts the Sandbox’s Token with AdjustTokenPrivileges()

5. The Broker places the Sandbox into a very restrictive Job Object by setting restrictive members of JOB_OBJECT_BASIC_UI_RESTRICTIONS when calling SetInformationJobObject()

6. The Broker can then place the Sandbox into its own Desktop (depending on which “type” of Sandboxed process it is) if XP, or on Vista set low integrity token and use User Interface Privilege Isolation (UIPI which is just “UAC” stuff)

7. The Broker does other stuff I probably didn’t notice (or am forgetting ;-) and then resumes the Sandbox’s main thread.
Bootstrapping the Sandbox

Example from “Sandbox PoC” in Chrome Source Code

(/home/chrome-svn/tarball/chromium/src/sandbox/sandbox_poc/main_ui_window.cc)

```c
sandbox::TargetPolicy* policy = broker_->CreatePolicy();

policy->SetJobLevel(sandbox::JOB_LOCKDOWN, 0);

policy->SetTokenLevel(sandbox::USER_RESTRICTEDSAME_ACCESS,
                      sandbox::USER_LOCKDOWN);

policy->SetAlternateDesktop(true);

policy->SetDelayedIntegrityLevel(sandbox::INTEGRITYLEVELLOW);

// Set the rule to allow the POC dll to be loaded by the target. Note that
// the rule allows 'all access' to the DLL, which could mean that the target
// could modify the DLL on disk.

policy->AddRule(sandbox::TargetPolicy::SUBSYS_FILES,
                sandbox::TargetPolicy::FILES_ALLOWANY, dll_path_.c_str());

sandbox::ResultCode result = broker_->SpawnTarget(spawn_target_.c_str(),
                                                   arguments, policy,
                                                   &target_);
```
The restricted token pretty much will handle restricting the vast majority (~95%) of the things malicious code will try to do:

- COM Interfaces
- Files
- Processes
- Shared Memory Regions
- Named Pipes
- Load Drivers (access Drivers)
- LPC/LRPC endpoints

When implementing a sandbox however, this doesn’t mean all the work is done for you, you still have to build strong “filter” policies for the Policy Engine!

A hole in your SID filters and the whole sandbox falls apart!
Restricted Token :: CreateRestrictedToken()

/home/chrome-svn/tarball/chromium/src/sandbox/src/restricted_token_utils.cc

```c
53  case USER_NON_ADMIN: {
54      sid_exceptions.push_back(WinBuiltinUsersSid);
55      sid_exceptions.push_back(WinWorldSid);
56      sid_exceptions.push_back(WinInteractiveSid);
57      sid_exceptions.push_back(WinAuthenticatedUserSid);
58      privilege_exceptions.push_back(SE_CHANGE_NOTIFY_NAME);
59      break;
60  }
61  case USER_INTERACTIVE: {
62      sid_exceptions.push_back(WinBuiltinUsersSid);
63      sid_exceptions.push_back(WinWorldSid);
64      sid_exceptions.push_back(WinInteractiveSid);
65      sid_exceptions.push_back(WinAuthenticatedUserSid);
66      privilege_exceptions.push_back(SE_CHANGE_NOTIFY_NAME);
67      restricted_token.AddRestrictingSid(WinBuiltinUsersSid);
68      restricted_token.AddRestrictingSid(WinWorldSid);
69      restricted_token.AddRestrictingSid(WinRestrictedCodeSid);
70      restricted_token.AddRestrictingSidGetCurrentUser();
71      restricted_token.AddRestrictingSidLogonSession();
72      break;
73  }
74  case USER_LIMITED: {
75      sid_exceptions.push_back(WinBuiltinUsersSid);
76      sid_exceptions.push_back(WinWorldSid);
77      sid_exceptions.push_back(WinInteractiveSid);
78      privilege_exceptions.push_back(SE_CHANGE_NOTIFY_NAME);
79      restricted_token.AddRestrictingSid(WinBuiltinUsersSid);
80      restricted_token.AddRestrictingSid(WinWorldSid);
81      restricted_token.AddRestrictingSid(WinRestrictedCodeSid);
```
The restrictions on the Job Object will generally handle restricting the “other” ~4.999% of things malicious code might try to do:

- Accessing/Writing Clipboard (JOB_OBJECT_UILIMIT_READCLIPBOARD)
- Switching/Accessing other Desktops (JOB_OBJECT_UILIMIT_DESKTOP)
- Accessing other USER32 Handles (JOB_OBJECT_UILIMIT_HANDLES) This kills all user32 messaging basically and techniques: SetWindowsHookEx, OpenWindow(), PostMessage(), SendMessage(), PeekMessage()}

The Job Object restrictions also breaks some less popular techniques:

- SendMessageCallback()
- GlobalAtom access (JOB_OBJECT_UILIMIT_GLOBALATOMS)
- ChangeDisplaySettings()
The Separate Desktop

★ Placing the sandboxed application on a separate desktop is mostly an “XP” (pre-UAC/UIPI technique)

★ On XP, user32 functions take only “window handles” as arguments.

★ Window Objects are grouped in “Desktops”, so intra-Desktop messaging by Objects, was not possible w/out switching.

★ Vista UIPI/UAC fixes this

SendMessage Function

Sends the specified message to a window or windows. The SendMessage function sends a message to a window, or to a thread’s message queue, and does not return until the window procedure has processed the message. To send a message and return immediately, use the SendMessage function. The syntax is:

Syntax

LPARAM wParam,
 circumstance
LPARAM lParam
);
Atom Tables & Global Atoms

★ What is the deal with Atom Tables? (InitAtom(), AddAtom(), FindAtom(), etc)

★ Designed originally to support Microsoft DDE (Dynamic Data Exchange).

★ Essentially is a “kernel supported” key/value storage mechanism for simple primitives (strings and integers)

★ Atom Tables are generally stored on “per process” basis
But you can create “Global Atoms” which are accessible by any process. (GlobalAddAtom(), GlobalFindAtom(), etc)

Note: Sample code for Atoms included in SandKit
GlobalAtoms: (excerpt from Sandkit tool)
GlobalAtoms

GlobalAtoms can thus be used a rudimentary form of IPC.

MANY standard Microsoft APIs and DLLs use Atom Tables.

How many Third Party applications misuse them?

Misuse of AtomTables is like the misuse of User32 WM_USER: Insecure usage happens when developers use it as a form of “quick and dirty” IPC.
The Lesson GlobalAtoms teach us:

★ While GlobalAtoms are a known technique with a known mitigation, the “pattern” is a lesson:
★ GlobalAtoms are essentially just Kernel/Native API supported storage mechanisms.
★ Are there more?
★ If so, they can probably be found anywhere there is something abstracted to be accessed via a “descriptor” from userland functions.
★ Places to start?
  • NTOSKRNL export “names” list in IDA,
  • MSUICHE’s MSDN (http://msdn.msuiche.net),
  • ReactOS, Third-Party Drivers
  • Ionescu’s “Native NT Toolkit code”
  • Gary Nebbett’s Native API Reference
  • Break on ObCreateObject() and see who dynamically creates objects.
The Hooks: Call Interceptions

"- my one's and my two's got your whole town shook; You betta listen to your corner, and watch for the hook!"  
--Cool Breeze/Goodie Mob/Outkast  
"Watch For the Hook"

★ Intended as a mechanism to assist the Broker/Sandbox Policy Engine **NOT** an enforcement mechanism itself (so they say).

★ In Chromium developer parlance the act of calling into the Broker via IPC mechanisms is called a “CrossCall”.

★ All library hooks generally reroute to stubs that ultimately perform CrossCalls to the Broker

★ The code responsible for “interceptions” is implemented in the Interception Manager
Identifying Hooks

Finding them is easy manually, but SandKit has tools to help you do it automated. “memdiff” in SandKit will compare the same region of memory in two separate processes and log differences.

Windbg .writemem command and simple Python/Ruby/whatever script can do this as well. Something like the following (in both the sandbox and broker Windbg sessions):

```bash	navi-two:sandbox_research s7ephens$ cat dump_memory_in_range.wds

lm #to find ranges of ntdll and kernel32
.writemem kernel32_broker.dmp 0x7c800000 0x7c8f6000
.writemem ntdll_broker.dmp 0x7c900000 0x7c9af000

navi-two:sandbox_research s7ephens$
```
After differ native library dumps you’ll find hooks like:

From NTDLL:
ZwCreateFile()
NtOpenFile()
ZwOpenProcess()
ZwOpenProcessToken()
ZwOpenProcessTokenEx()
NtOpenThread()
ZwOpenThreadToken()
NtOpenThreadTokenEx()
ZwQueryAttributesFile()
ZwQueryFullAttributesFile()
NtSetInformationFile()
many many more

Page permissions kinda imply PE section. We only care about .text

Many other libraries are hooked as well.
The Hooks: In the source.

Although the Chome Sandbox source (as a framework) is BSD licensed and open as are all the policies and rules used in the Chrome distribution.

It may not seem particularly evident when you look through source because you will probably only see references to Interception Manager in test code.

```
/home/chrome-svn/tarball/chromium/src/sandbox/src/interception_unittest.cc
```

```c
InterceptionManager interceptions(target, true);

// Any pointer will do for a function pointer.
void* function = &interceptions;
interceptions.AddToUnloadModules(L"some01.dll");
// We don't care about the interceptor id.
interceptions.AddToPatchedFunctions(L"ntdll.dll", "NtCreateFile",
   INTERCEPTION_SERVICE_CALL, function,
   OPEN_FILE_ID);
interceptions.AddToPatchedFunctions(L"kernel32.dll", "CreateFileEx",
   INTERCEPTION_EAT, function, OPEN_FILE_ID);
interceptions.AddToUnloadModules(L"some02.dll");
interceptions.AddToPatchedFunctions(L"kernel32.dll", "SomeFileEx",
   INTERCEPTION_SMART_SIESTEP, function,
   OPEN_FILE_ID);
```
Google Chromium Team has long asserted that hooks themselves are not to be relied upon a security enforcement mechanism. This shows they “get it”. Hooks can be unhooked.

However one thing to note is the effectiveness of the “VirtualProtect()/WriteProcessMemory() hook Catch 22” which is:

Malicious code executing in the sandbox would have to use GetCurrentProcess()/VirtualProtect()/WriteProcessMemory() to “unhook”.

What if these functions are already hooked? In my opinion, this might be a significant hurdle to deter most exploit developers.
To circumvent the GetCurrentProcess()/VirtualProtect()/WriteProcessMemory() catch 22 a malware author could just use syscalls directly, and completely circumvent the library hooks.

FEATURE REQUEST? Why doesn’t Microsoft expose functionality for Syscall restriction/filtering on per-process bases? Other lesser sandbox technologies (like those for *nixes and SandboxIE use this as the core):

- Win7/Vista already kinda has some close with the less known EPROCESS.ProtectedProcess

Does EPROCESS.ProtectedProcess prevent: WriteProcessMemory(GetCurrentProcess())?
Finding Hooks Via Call Traces

Although more annoying to do, you can find hooks using call tracing.

I do my kernel call-tracing using custom tools or in Windbg:

```
bp /p <cid of target> kernel32!CreateFileW "du.poi(@esp+4);.process;k;g"
```

Alternatively for Win7 targets you might have to:

```
bp kernel32!CreateFileW "du.poi(@esp+4);.process;k;g"
```

If you are in user-space and want a “point and click” call-tracer, I suggest the surprisingly unpopular but extremely powerful: AutoDebugPro
Moving closer to kernel/user gap.

As we tunnel down to observe the Native API hooks put in place by the Broker we see that many of these are the Zw* Nt*

These are obviously the functions which are at the “edge of the precipice” between userland and kernel, one or two steps away from SysEnter/SysCall/INT 2e/call gate/etc.

This is where things get interesting and is perfect segue into how we can investigate Sandboxes from up in the Kernel.

(Kernel space is so much more relaxing. Its “quieter”.)
Sandboxes from a Kernel-space Perspective
Why Look at Sandboxes from Kernel?

- Perhaps investigating the relationship between Userspace/Kernelspace will reveal new attack surface.

- It’s so much “quieter” in the Kernel. It is a nice reprieve from the hustle and bustle of User-space.

- More control: Pause execution and the whole box freezes. This means the Broker AND the Sandbox, no loss of “sync”.

- Windbg Kernel Debugger (Kd) has commands we can’t use from User-space.

- Virtually everything on Windows is performed predominantly using NT Objects, all inspectable from Kd.
Kernel Components (refresher!)

- Object Manager (OB)
- Security Reference Monitor (SE)
- Process/Thread Management (PS)
- Memory Manager (MM)
- Cache Manager (CACHE)
- Scheduler (KE)
- I/O Manager, PnP, power, GUI (IO)
- Devices, FS Volumes, Net (DRIVERS)
- Lightweight Procedure Calls (LPC)
- Hardware Abstraction Layer (HAL)
- Executive Functions (EX)
- Run-Time Library (RTL)
- Registry/Consistent Configuration (CONFIG)
Kernel Components (refresher!)

★ **Object Manager** (OB)
★ **Security Reference Monitor** (SE)
★ Process/Thread Management (PS)
★ Memory Manager (MM)
★ Cache Manager (CACHE)
★ Scheduler (KE)
★ I/O Manager
★ Devices, FS Volumes, Net (DRIVERS)
★ Lightweight Procedure Calls (LPC)
★ Hardware Abstraction Layer (HAL)
★ Executive Functions (EF)
★ Run-Time Library (RTL)
★ Registry/Consistent Configuration (COMRC)

For Sandboxing technologies, these are mostly what we care about.
Here’s why OB/SE/IO matter most:

<table>
<thead>
<tr>
<th>Object Type</th>
<th>Represents</th>
<th>Defining Subsystem</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object type</td>
<td>Object type object</td>
<td>Object Manager</td>
</tr>
<tr>
<td>Directory</td>
<td>Object namespace</td>
<td>Object Manager</td>
</tr>
<tr>
<td>SymbolicLink</td>
<td>Object namespace</td>
<td>Object Manager</td>
</tr>
<tr>
<td>Event</td>
<td>Synchronization primitive</td>
<td>Executive</td>
</tr>
<tr>
<td>EventPair</td>
<td>Synchronization primitive</td>
<td>Executive</td>
</tr>
<tr>
<td>Mutant</td>
<td>Synchronization primitive</td>
<td>Executive</td>
</tr>
<tr>
<td>Semaphore</td>
<td>Synchronization primitive</td>
<td>Executive</td>
</tr>
<tr>
<td>Windows Station</td>
<td>Login session</td>
<td>Win32</td>
</tr>
<tr>
<td>Desktop</td>
<td>Windows desktop</td>
<td>Win32</td>
</tr>
<tr>
<td>Timer</td>
<td>Timer notifications</td>
<td>Executive</td>
</tr>
<tr>
<td>File</td>
<td>Tracks open files</td>
<td>I/O Manager</td>
</tr>
<tr>
<td>IoCompletion</td>
<td>Tracks I/O completion notifications</td>
<td>I/O Manager</td>
</tr>
<tr>
<td>Adapter</td>
<td>DMA resource</td>
<td>I/O Manager</td>
</tr>
<tr>
<td>Controller</td>
<td>DMA controller</td>
<td>I/O Manager</td>
</tr>
<tr>
<td>Device</td>
<td>Logical or physical device</td>
<td>I/O Manager</td>
</tr>
<tr>
<td>Driver</td>
<td>Device driver</td>
<td>I/O Manager</td>
</tr>
<tr>
<td>Key</td>
<td>Doorway to the Registry</td>
<td>Configuration Manager</td>
</tr>
<tr>
<td>Port</td>
<td>Communications channel</td>
<td>LPC Facility</td>
</tr>
<tr>
<td>Section</td>
<td>Memory mapping</td>
<td>Memory Manager</td>
</tr>
<tr>
<td>Process</td>
<td>Active process</td>
<td>Process Manager</td>
</tr>
<tr>
<td>Thread</td>
<td>Active thread</td>
<td>Process Manager</td>
</tr>
<tr>
<td>Token</td>
<td>Process security profile</td>
<td>Process Manager</td>
</tr>
<tr>
<td>Profile</td>
<td>Performance monitoring</td>
<td>Kernel</td>
</tr>
</tbody>
</table>
The NT Object Manager (OB):

- Provides underlying NT namespace
- Unifies kernel data structure referencing
- Unifies user-mode referencing via handles/descriptors
- Central facility for security protection Provides device & I/O support

Important Note: Objects are extensible. You can build your own based on the primitives. Many kernel code does just this dynamically.

credit: Dave Probert, Ph.D (Singapore 2006), Microsoft Corporation 2006
The Security Reference Monitor (SE):

- Based on discretionary access controls
- Single module for access checks (e.g. SeAccessCheck())
- Implements Security Descriptors, System and Discretionary ACLs, Privileges, and Tokens
- Collaborates with Local Security Authority Service (LSASS) to obtain authenticated credentials
- Provides auditing and fulfills other Common Criteria requirements

credit: Dave Probert, Ph.D (Singapore 2006), Microsoft Corporation 2006
How OB and SE interact:

Note:
A “Name” might be: `\\.pipe\protected_storage`

credit: Dave Probert, Ph.D (Singapore 2006), Microsoft Corporation 2006
Remember! Handles/Descriptors are just userland abstractions!

- Handles and Descriptors are just Userland abstractions to access Kernel structures.
- The functions you pass the Handles and Descriptors into (like fopen()) are userland “gateways” to the kernel.

credit: Dave Probert, Ph.D (Singapore 2006), Microsoft Corporation 2006
© Microsoft Corporation 2006
<table>
<thead>
<tr>
<th>NT Objects (the object “primitives”)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Adapter</strong></td>
</tr>
<tr>
<td><strong>Callback</strong></td>
</tr>
<tr>
<td><strong>Controller</strong></td>
</tr>
<tr>
<td><strong>DebugObject</strong></td>
</tr>
<tr>
<td><strong>Desktop</strong></td>
</tr>
<tr>
<td><strong>Device</strong></td>
</tr>
<tr>
<td><strong>Directory</strong></td>
</tr>
<tr>
<td><strong>Driver</strong></td>
</tr>
<tr>
<td><strong>Event</strong></td>
</tr>
<tr>
<td><strong>EventPair</strong></td>
</tr>
</tbody>
</table>
Listing/Investigating NT Objects

- WinObj (SysInternals)
- objdir.exe (DDK)
- ntddk.h
- Ob*() exports of ntoskrnl.exe
- “Undocumented Windows 2000 Secrets” Chapter 7 (w2k_def.h)
- dt nt!_object* (in Windbg (kd))
- !object \ (in Windbg (kd))
First things first...why go up here?

Reasons for using kernel debugger to assist us with investigating sandboxes:

1. Sandboxes use many NT Objects that have helpful WinDbg commands that don’t work from userspace:
   1. Jobs Objects for example! (!job)
   2. LPC inspection (!lpc)
   3. Better handle/descriptor visibility/tracking (!htrace)

2. “System-Wide” breakpoints: Breaking on ntdll! NtOpenFile() will hit whenever any process on the system calls it!

3. There are also some other less popular benefits to using kernel debugger (will demonstrate these with Google Chrome later :-)
Inspecting Securable Objects with Kernel Debugger

- !process <cid>
- !handle <cid>
- !job
- !token
- !tokenfields
- !object

- !acl
- !sid
- !lpc

Side Note: Did you know you don’t need to use gflags.exe to set pageheap/debugheaps? You can use Windbg’s !gflag

See “Determining the ACL of an Object” in the Windbg help for all the steps to obtaining a detailed security descriptor from an object.

Thursday, July 15, 2010
Other useful commands

★ .tlist : This also lists processes but only by CID and not process identifier.

★ !process 0 0 : List all cids/processes

★ .process

★ .reload /user : Reload userspace symbols

★ .sympath symsrv*symsrv.dll*c:\\syms*http://msdl.microsoft.com/download/symbols
  • Autodownload of symbols you dont have... VERY USEFUL!

★ lm u : list modules for userspace, needs a .process
A Note on Observing Hooks from Kernel Debugger

★ Important to remember: in the kernel only “one copy” of libraries (like ntdll) ever get loaded.

★ The “differences” between processes is all done via the magic of Page Table Entries. You will probably not be able to see installed library hooks if you don’t do the following in Windbg:

• use the /p switch with the .process command to force the debugger to update Page Table translation: .process /p <eprocess|cid>

★ This is done so that when you view the virtual address for NTDLL or Kernel32 or whatever, it correctly references the physical page, which differ because of the hooks.

★ Note: you may also want to check out the Windbg .pagein command. You might have to use this command as another way to force Windbg to update PTE translation.
Observing Broker Behaviors

There are a number of functions critical to the operation of Sandbox child processes that are interesting and useful to observe the Broker calling. Here are some suggestions:

Note: Most of these are “undocumented”.

- Zw/NtDuplicateToken()
- Zw/NtCreateToken()
- Zw/NtSetInformationToken()
- Zw/NtOpenProcess()
- Zw/NtDuplicateObject()
- DuplicateHandle()
- Zw/NtCreateProcess()
- Zw/NtSetSecurityObject()
- NtQueryObject(), NtSetSecurityObject(), NtQuerySecurityObject()
- ExDupHandleTable()/ExDestroyHandleTable (process creation/destruction)
- ExCreateHandle(), ExDestroyHandle()
- user32!UserHandleGrantAccess()
Because the Sandbox is restricted we care less about what he is doing, but there are a few interesting things to watch for. Here are some suggestions:

*Note: Most of these are “undocumented”.*

- ZwContinue(): the _NTCONTINUE function that is often hooked by anti-debugging code (not that Chrome does it)
- ZwCreateFile()
- ZwWriteFile()
A Neat thing about Kernel Debuggers

★ The kernel gets **ALL** exceptions first!

★ Like virtually all Windows functionality, Usermode debuggers rely heavily upon LPC messages.
★ “Debugger” processes talk to CSRSS via LPC
★ CSRSS receives all debug events for all processes from the kernel and handles dispatching them debugger processes.
★ When a Kernel debugger is attached, the Kernel never passes these exceptions on to CSRSS’s waiting LPC channel.
★ The most important thing however is that the Kernel gets **all** exceptions first, especially int 3, which is what Chrome sandbox uses to taddle-tell back to the Broker ;-)
★ In Vista/Win7 this is different: see ZwCreateDebugObject()
TANGENT: Detecting Kernel Mode Debuggers from Userspace

Once you know about how Kernel mode debuggers get all exceptions first, the concept is simple:

Use RDTSC single-step detection technique with int3s in-between to detect kernel debugger exception handler timing.

Furthermore, int3s fired at the “wrong time” break things. See for yourself.

If you dig a bit under the hood to understand the process around ZwCreateDebugObject (XP+), how CSRSS passes debug info, and stuff like EPROCESS.DebugPort and \Windows\ApiPort you will probably find better ways to detect Kernel debuggers from userspace
TANGENT: Detecting Kernel Mode Debuggers from Userspace

How it might look in C?

```c
void start(){
    timea = GetTickCount();
}

void stop(){
    timeb = GetTickCount();
}

DWORD checktime(){
    time_delta = timeb - timea;
    printf("%d\n", timea);
    printf("%d\n", timeb);
    return time_delta;
}

int main(int argc, char* argv[]){
    start();
    __asm {
        //exception they have to jump
        int 3
    }
    stop();
    if (BeingDebugged()) {
        MessageBox(NULL, "Don't be debugging me!", "WTF!", MB_OK);
        ExitProcess(0);
    } else {
        if (checktime() > 1){
            MessageBox(NULL, "Hah! I can still see you are debugging me!", "WTF!", MB_OK);
            ExitProcess(0);
        }
    }
    ExitProcess(0);
}
```

How it might look in ASM?

1. rdtsc
2. mov ecx, eax
3. int 3
4. rdtsc
5. sub eax, ecx
6. cmp eax, 0x1000
7. ja kernel_debugger_detected
If you fuzzed sandboxed processes and had “success” you’ve probably seen this (I call it “Chrome Mr. Yuck”):

but when you attach your user-space debugger....nothing. That’s because the Broker catches sandbox exceptions and breakpoints first!
Google being snide about Broker-handled Sandbox exceptions...

On the Chromium website, down in some documentation Google mentions this:

This is no mystery at all when you realize that the Sandbox (the debuggee) is coded to intentionally whine to the Broker by throwing exceptions which the Broker (as the debugger) then “handles”.

GOOGLE DOES NOT USE THE OS’S CRASH REPORTING MECHANISMS (like WER in Windows or Crash Reporter in OSX). It uses it’s own custom one called BreakPad.

Pro-tip: If fuzzing Chrome, be sure to set your ZoneAlarm/LittleSnitch/whatever to disallow Chrome outbound. Or better yet, disable the NIC entirely for that VM ;-)

Thursday, July 15, 2010
Example of Remotely Triggered (client side) overflow (handled)
Tools & Techniques: Introducing The SandKit
The SandKit

A Collection of tools to assist with the investigation and testing of Sandboxes.
(Also intended to give ideas about tools you might want to write yourself.)

- Code Injection Techniques (vanilla dll injection, reflective dll injection, kernel-to-userspace dll injection?)
- CopyMem
- MemDiff
- DumpMem
- HookFix
- Sa7Shell
- BinCompare
- DumpToken Redux
- TokenBrute/HandleBrute
- Sandbox_Poc (Google Chrome source “sub-project”)
  - Download the Chrome source and find it in:
  - /home/chrome-svn/tarball/chromium/src/sandbox/sandbox_poc/
  - It comes with visual studio solution and everything!
Code Injection

Sandkit implements “Vanilla DLL injection” to inject a DLL into a target process.
- This injection technique is the VERY common: OpenProcess()/VirtualAllocEx()/CreateRemoteThread()->LoadLibraryA() technique.

Reflective DLL injection
- for “harder” injection targets such as restricted processes or heavily hooked executables.
- some minimal unhooking would still necessary
- Sandkit may eventually include this.

Kernel-to-userspace Injection?
- Use documented APC Injection/Thread Notifier technique to have kernel injected code run in a usermode Thread’s context
- Combine this with basic Reflective DLL injection technique
- MANY caveats: accounting for PTE changes when injected code executes (hooks still in place), modifying PTE for usermode context, etc.

Thursday, July 15, 2010
Copy memory from one process into another. This tool is the basis for the HookFix application.
MemDiff

- Take a look into memory in two different processes and compare it.
- Log where the two regions of memory begin to differ.
- Simple but time-saving tool for the detection of hooks

```
SandKit>> help
...
<for help, type: help <command>>
EOF dumpmem help hook_fix memdiff pythonshell sa7shell
copy_mem exit hist injectdll ps readmem
SandKit>> help memdiff

Compare two regions of memory in two different processes and report where these regions of memory differ.
Usage:
   memdiff <pid-one> <address> <length in bytes> <pid-two> <address>
SandKit>> ps chrome
Pids of processes with names matching 'chrome':
2288 : chrome.exe
1840 : chrome.exe
SandKit>> memdiff 2288 0x7c885000 20 1840 0x7c885000
Attached to PID: 2288
Attached to PID: 1840
==> Sizes of files are the same (20 bytes), a good start!
***** Files differ at byte: 0x0
***** Files differ at byte: 0x1
***** Files differ at byte: 0x2
***** Files differ at byte: 0x3
```
Similar to the \texttt{.writemem} command in Windbg. Just write raw memory from a process to a file.

```
SandKit>> memdiff 2288 0x7c885000 100 c:\chrome_kernel32_memdump.dmp
Incorrect number of arguments.
SandKit>> help dumpmem
Read memory from a process and write it to a file.
Usage:
dumpmem <pid> <address> <length in bytes> <file to dump to>
```

```
SandKit>> ps chrome
Pids of processes with names matching 'chrome':
2288 : chrome.exe
1040 : chrome.exe
SandKit>> memdump 2288 0x7c885000 100 c:\chrome_kernel32_memdump.dmp
Bad command or filename.
SandKit>> dumpmem 2288 0x7c885000 100 c:\chrome_kernel32_memdump.dmp
Attached to PID: 2288
Wrote 100 bytes to file 'c:\chrome_kernel32_memdump.dmp'.
SandKit>>
```
Write a string or character array directly to the memory of a process.
HookFix

★ HookFix just uses CopyMem to fix the specific hooks put in placed into the Sandbox by the Broker.

★ There is no magic here, we just:

1. Borrow the .text region of a "normal" process with our module loaded (in this case the Broker).
2. Locate the differences between the "normal" and modified .text regions within the Sandbox
3. Save the Sandbox modules .text region first (for restoration).
4. Overwrite the Sandbox module’s .text region

Note: We have to just be careful to not to borrow stuff outside of .text, because there are “process specific” variables in the address space of dlls like ntdll. Such as:

    ntdll!__security_cookie
After using the Sandkit DLL injector, you get a console window!

If you can read this, a Console has been successfully allocated!

Testing STDIN: type some text then hit <ENTER>
You entered: "testing"

Py_Initialize() is @ 1e08f0b0
Py_Finalize() is @ 1e08e860
PyRun_InteractiveLoop() is @ 1e08f340
PyRun_SimpleString() is @ 1e08f2d0
Jumping into the python dll.
This is a test print() from inside python.

Attempting to enter interactive mode of python...

sa7>>help
Type help() for interactive help, or help(object) for help about object.
sa7>>dir()
['__builtins__', '__doc__', '__name__', 'sys']

sa7>>>
Messing around inside the process (notepad.exe) like Message Box popups!
Sa7Shell: How does it work?

Inject the full Python interpreter into a target process, and mess around with it internally!

- This may sound trivial to do with vanilla DLL injection and it (for the most part is).
- However you have to handle special cases like: If your injected DLL does printf(), where does STDOUT go in a GUI app?
- Answer: AllocateConsole() and then my “handle shenanigans”

```c
//Ok, this is a lame trick but it seems to work! From testing, it looks like
//GetConsoleTitle() is a cheap way to detect whether an app even has a
//Console created, it also seems to adequately test whether an app even has
//console capabilities. I tested this by injecting into a bunch of different apps
//and it seems to be reliable.
thang = (LPTSTR)GlobalAlloc(GMEM_ZEROINIT, 2000);
if (GetConsoleTitle(thang,1999) == 0) //Console window does not exist
    //so we have to create one.
    MessageBox(NULL, "No Console Window exists. Creating one.", "!", MB_OK);
else
    MessageBox(NULL, pName, "A Console already exists. ", MB_OK);
if (!AllocConsole()){
    MessageBox(NULL, "Can not AllocConsole()!", "!", MB_OK);
    return TRUE;
} else {
    MessageBox(NULL, "AllocConsole() successful!", "!", MB_OK);
    SetConsoleTitle("Sa7oriShell running in yer processes...");
```
Sa7Shell: Handle Shenanigans

```c
lStdHandle = (long)GetStdHandle(STD_OUTPUT_HANDLE);
if (lStdHandle == (long)INVALID_HANDLE_VALUE)
    MessageBox(NULL, "Could not get STD_OUTPUT_HANDLE", "!", MB_OK);
//The next line causes process to exit with no exceptions when injected
//into remote process.

hConHandle = _open_osfhandle(lStdHandle, _O_TEXT);// _O_TEXT defined in
//#include <fcntl.h> and _open_osfhandle in io.h

if (hConHandle == -1)
    MessageBox(NULL, "Could not open STD_INPUT_HANDLE", "!", MB_OK);

fp = fdopen( hConHandle, "w" );
stdout = *fp;
setvbuf( stdout, NULL, _IONBF, 0 );

// redirect unbuffered STDIN to the console
lStdHandle = (long)GetStdHandle(STD_INPUT_HANDLE);
if (lStdHandle == (long)INVALID_HANDLE_VALUE)
    MessageBox(NULL, "Could not get STD_INPUT_HANDLE", "!", MB_OK);

hConHandle = _open_osfhandle(lStdHandle, _O_TEXT);
if (hConHandle == -1)
    MessageBox(NULL, "Could not open STD_INPUT_HANDLE", "!", MB_OK);

fp = fdopen( hConHandle, "r" );
stdin = *fp;
setvbuf( stdin, NULL, _IONBF, 0 );

// redirect unbuffered STDERR to the console
lStdHandle = (long)GetStdHandle(STD_ERROR_HANDLE);
if (lStdHandle == (long)INVALID_HANDLE_VALUE)
    MessageBox(NULL, "Could not get STD_ERROR_HANDLE", "!", MB_OK);

hConHandle = _open_osfhandle(lStdHandle, _O_TEXT);
if (hConHandle == -1)
    MessageBox(NULL, "Could not open STD_ERROR_HANDLE", "!", MB_OK);

fp = fdopen( hConHandle, "w" );
stderr = *fp;
setvbuf( stderr, NULL, _IONBF, 0 );
```
PythonShell command in Sandkit

Drop directly into a python shell from Sandkit to fiddle:

```bash
C:\WINDOWS\System32\cmd.exe - c:\Python24\python.exe SandKit.py
Z:\data\CHECKOUTS\github\SandKit>c:\Python24\python.exe SandKit.py

...ooo000 Welcome to 000ooo...
SandKit
...ooo000000000000000000000oo...

SandKit>> help pythonshell

Use this to drop BACK to an interactive python shell. This can be used to then enter python code or import python modules as you would with the normal python interactive shell.

SandKit>> pythonshell

*** Welcome to SandKit Interactive Python Console ***
Break out with CTRL-Z.
>>> import bincompare as bc
>>> import litedbg
>>> litedbg.hexdump("Booyah Grandma!")
00000000: 42 6f 6e 65 6c 6c 6f 77 72 6f 6d 20 47 72 61 6e
!Booyah Grandma!

>>> bc.compare("Booyah Grandma!", "BooYah Grandma!", 0)
== Sizes of files differ by 1 bytes.
>>> ^Z
```

Thursday, July 15, 2010
BinCompare (stand-alone)

★ A standalone tool that does the same thing that memdiff does but specifically for files instead of just memory.
★ One of those stupidly simple things that is massively useful.

```bash
navi-two:sandbox_research s7ephen$ ./bincompare.py --help

BinCompare
    Compare two files starting at the first byte.

./bincompare.py <file1> <file2> tolerance

tolerance: the number of first "differences" to ignore.
    if 0, dont stop until end of file.
```

```bash
navi-two:sandbox_research s7ephen$ cat dump_memory_in_range.wds
lm #to find ranges of ntdll and kernel32
.writemem kernel32_broker.dmp 0x7c800000 0x7c8f6000
.writemem ntdll_broker.dmp 0x7c900000 0x7c9af000
navi-two:sandbox_research s7ephen$ ./bincompare.py kernel32_broker.dmp kernel32_sandbox.dmp

==> Sizes of files are the same (1007616 bytes), a good start!

***** Files differ at byte: 0x85000
***** Files differ at byte: 0x85001
***** Files differ at byte: 0x85002
***** Files differ at byte: 0x85003
***** Files differ at byte: 0x85024
***** Files differ at byte: 0x85048
```

Thursday, July 15, 2010
DumpToken Redux

A Dll’d and .h’d version of Matt Conover’s DumpToken tool with additional native API helpers such as NtQueryObject ObjectType Information.

The .h and .dll make it easily reusable in your injectable code.

This screenshot is from code that has been injected into an app using Sandbox_PoC from Google Chrome.
TokenBrute/HandleBrute: A Token/Handle Sniper

- Inspired by a part of Cesar Cerrudo’s (MS04-044) PoC
- A Dll’d and .h’d tool that “snipes” or “steals” tokens granted into a process by brute forcing token handles
- Not magic. Surprisingly simple actually. Iterates 0 to MAX_HANDLES (10,000 on XP) in separate thread.
- Also uses DumpToken Redux to display info if token is found.

Success getting Thread handle...
Starting Token handle search...
Found A TOKEN that let us SetThreatToken<> on it!
Token was at handle: This is a unrestricted token
Token type: impersonation
Impersonation level: identification
Token ID: This is just “identification” but you get the concept ;-)
Where do I get all this stuff?

How can I follow up after this talk?
Where to get it?

★ Sandkit and this presentation is here:

http://s7ephen.github.com/SandKit

★ Get these slides there.

★ Follow on Github for updates. (As I package/sanitize my private tools for public release I will be adding them to the SandKit project.)
In a nutshell:

For Bug Hunters:
Things to look into.

For Sandbox Developers:
Things to look out for.
Auditing sandboxes is entirely a “configuration” audit game.

Applications written without sandboxing in mind have the worst trouble shoe-horning into a sandbox.

Exhaustively check everything from the inside of the Sandbox out. Try to make these test cases integral parts of your build/release process.

Don’t “cheat” and pass tokens/handles/etc into the sandbox! Even for a “quick moment”.

Merely having the sandbox doesn’t secure you. You must should how to configure it (build PolicyFilters, install your own Intercepts even!)
There are really two audits: Audit of the “Sandbox” itself and Audit of the “Sandbox implementation”
- “Sandbox” bugs will be where the Sandbox meets the OS/Kernel or the IPC channels back into the “Broker”. These are harder and higher value ;-)
- “Sandbox implementation” bugs will be where the Sandbox meets the application’s requirements. These are specific to the app.

Applications written without sandboxing from the ground up will have difficulty shoe-horning into a sandbox
- The larger the application, the higher probability something (a legacy library, thread, etc) will require lax token restrictions and SID filters.

If you have code execution inside the sandbox, don’t be afraid to have your code “wait patiently” for the proper execution environment.
Do you need any work like this?

- Software Reverse Engineering?
- Penetration Testing?
- Source Code Auditing?
- Security Architecture Analysis?
- Embedded System Security?
- Security Consultation?
- Cryptography Implementations?
- Blackbox auditing of software/hardware?
- Whitebox auditing of software/hardware?
- Web application penetration testing?

Matasano does all of this!

Contact Me for more Info!

stephen@matasano.com
Special Thanks and Contact Info
SPECIAL THANKS

Stephen C. Lawler
Mathieu “Sandwich” Suiche
Stephania Vu
Google Security Team
My Mt. Whitney Summit Team
(Hai, Quynh, Rick, Nick)

The “customer” who approved and cleared me to give this talk ;-)
THANKS FOR Listening!
I hope this is helpful.

stephen@sa7ori.org
Twitter: s7ephen