FUZZING THE WINDOWS KERNEL

Yong Chuan Koh

26th August 2016
Yong Chuan, Koh (@yongchuank)

- Security Consultant and Researcher
- @ MWR Infosecurity (SG) since 2014
- Interests:
  - Reverse Engineering
  - Vulnerability Research
  - Malware Analysis
- Previous Research
  - “Understanding the Microsoft Office 2013 Protected-View Sandbox”
OUTLINE

• Introduction
• Framework Architecture And Components
• Framework Algorithms
• Framework Setup And Configuration
• Results And Case Study
• Conclusion And Future Work
FUZZING THE WINDOWS KERNEL

INTRODUCTION
INTRODUCTION

++

Sandbox

• Sandboxing 101
  – Wikipedia: “…a sandbox is a security mechanism for separating running programs...A sandbox typically provides a tightly controlled set of resources for guest programs to run in, ...A sandbox is implemented by executing the software in a restricted operating system environment, thus controlling the resources (...) that a process may use…”

• Sandbox aims to contain exploits by limiting damage to system
++

Sandbox Escapes

- Maturity of sandbox adoption in popular applications...
  - 2006: Internet Explorer 7 Protected-Mode
  - 2010: Chrome Browser Sandbox
  - 2010: Adobe Reader X Protected Mode
  - 2012: Internet Explorer 10 EnhancedProtected-Mode
Pwn2Own Winning Entries
- 2016: 6 new Kernel vulnerabilities / 7 attempts on Windows targets
- 2015: 4 new Kernel vulnerabilities / 7 attempts on Windows targets
- 2014: 1 new Kernel vulnerabilities / 8 attempts on Windows targets
- 2013: 1 new Kernel vulnerabilities / 8 attempts on Windows targets

Increased Kernel patches from 2014–2015 (~4X)

INTRODUCTION

++

Ok, Kernel is pretty huge...

- Which Kernel component?
  - ~600+ drivers in %WINDIR%\System32
  - Loaded by default, reachable in sandbox
  - Complicated code
  - “Spidey sense”...

- WIN32K.SYS driver
  - 2997280 bytes
  - Complicated
  - Lots of disclosed vulnerabilities already
  - “How bad design decisions created the least secure driver on Windows” by Thomas Garnier
INTRODUCTION

WIN32K.SYS Kernel-Mode Driver

• “Windows Kernel Internals: Win32k.sys” by Dave Probert

• Graphical User Interface (GUI) infrastructure of Windows
  – Window Manager (USER)
  – Graphic Device Interface (GDI)
  – Dx thunks to dxg.sys (DirectX)

• W32UMode.lib DLLs
  – USER32.DLL, IMM32.DLL
  – GDI32.DLL, MSIMG32.DLL
  – D3D8THK.DLL

Dave Probert: http://pasotech.altervista.org/windows_internals/Win32KSYS.pdf
INTRODUCTION

++

Goals

• Windows Kernel Fuzzing Framework
  – Easily scalable
  – Reproducible BSOD
  – Modular and adaptable

• Friendly internal competition
  – “Windows Kernel Fuzzing” by Nils
  – “Platform Agnostic Kernel Fuzzing” by James Loureiro and Georgi Gesheev
  – Different implementation find different vulnerabilities

• Learning about Windows Kernel security
Fuzzing the Windows Kernel

- FRAMEWORK ARCHITECTURE AND COMPONENTS
++

### Architecture

- **Catalog**
  - System Calls
  - Library Calls
  - Fuzz Values

- **Execution**
  - TestCases

- **Databases**
  - Handle Database
  - Log Database
  - Dump Database

- **Modes**
  - FUZZ MODE
  - REPRO-MIN MODE
++ FRAMEWORK ARCHITECTURE AND COMPONENTS

Architecture

Catalog
- System Calls (Ordered)
- Library Calls (Ordered)
- Fuzz Values

Handle Database
Log Database
Dump Database

Maps
Provides

TestCases

FUZZ MODE
Reproduce

REPRO-MIN MODE
Minimize

“Reproduce + Minimize”
++

Architecture - Implementation

• Implemented in Python
  – Familiarity and ease ✓
  – Extensive usage of ctypes library for C-compatibility ✓
  – Re-define numerous C function prototypes and structures ✗

• Alternative: C/C++
  – Development and debugging ✗
  – Existing C function prototypes and structures ✓
  – Efficient fuzzing performance ✓
Components – Catalog

- Determine interaction with target Kernel component
  - In this case, fuzzing Win32k.sys with relevant library and system calls
  - Easily repurposed for different Kernel components
- Quality of catalog determines
  - Type of vulnerability class
  - Code coverage
Components – Catalog

• Collection of Library and System call definitions
  – Argument types and values
  – Return values
  – Custom logging syntax rules to bridge Fuzz Mode and Repro–Min Mode

• Purpose of Library calls
  – Wrapper for System calls
  – Introduce more randomness

• Sources for Library and System call definitions
  – MSDN, Headers, ReactOS (thanks!), Google–fu, reverse–engineering
Components - Catalog Syntax Rules

- Categorize argument and return types
  - HEX, STRING, HANDLE, STRUCTURE

- Syntax Rule: HEX
  - Integers represented in hexadecimals
  - Signed vs unsigned
  - Byte vs Word vs Dword vs Qword

- Syntax Rule: STRING
  - Pointers to sequence of bytes
  - Arrays, Strings, Pointers to integers, etc
Components – Catalog Syntax Rules

• Syntax Rule: HANDLE
  – Special User–land references to Kernel–land objects
  – Different values between Fuzz Mode and Repro–Min Mode runs
  – Database to store handles to types (Fuzz Mode)
  – Database to provide handles to types (Fuzz Mode)
  – Database to map handle values to creation (Repro–Min Mode)
Components – Catalog Syntax Rules

• Syntax Rule: STRUCTURE
  – Combination of HEX, STRING and HANDLE
  – Represented as STRING in itself
  – Can also contain HEX, STRING and HANDLE in fields
++

**Components – Catalog Example 1**

```c
HBITMAP CreateCompatibleBitmap (  
    _In_ HDC hdc,  
    _In_ int nWidth,  
    _In_ int nHeight
);
```

```python
class GDI32_CreateCompatibleBitmap (TestCase):
    def generateArguments (self):
        self.hdc = self.handlearg ("HDC")
        self.nWidth = self.hexarg (self.GetFuzzValue ("Hex"))
        self.nHeight = self.hexarg (self.GetFuzzValue ("Hex"))

        self.args.append (self.hdc)
        self.args.append (self.nWidth)
        self.args.append (self.nHeight)

def runTestCase (self):
    self.handle = gdi32.CreateCompatibleBitmap (self.args[0], self.args[1], self.args[2])
    self.addhandle ("HBITMAP", self.handle)
```

- **Catalog Definition**
  - Get HDC from HANDLE Database
  - Get fuzz HEX values
  - Add HITMAP to HANDLE Database

Reference from MSDN
FRAMEWORK ARCHITECTURE AND COMPONENTS

++

Components – Catalog Example 2

```python
class ACCEL(ctypes.Structure, TestCase):
    _fields_ = [('fVirt', BYTE), ('key', WORD), ('cmd', WORD)]

def __init__(self, *args, **kwargs):
    setattr(self, 'fVirt', self.GetFuzzValue('Hex'))
    setattr(self, 'key', self.GetFuzzValue('Hex'))
    setattr(self, 'cmd', self.GetFuzzValue('Hex'))
```

**Structure Definition**

```python
class USER32_CreateAcceleratorTableA (TestCase):
    def generateArguments (self):
        self.cEntries = self.hexarg (self.GetFuzzValue('Hex'))
        self.lpaccel = self.structarg (ACCEL)

        self.args.append (self.lpaccel)
        self.args.append (self.cEntries)

    def runTestCase (self):
        self.handle = user32.CreateAcceleratorTableW (self.args[0], self.args[1])
        self.addhandle ('HACCEL', self.handle)
```

**Catalog Definition**

- Get fuzz HEX values
- Get STRUCTURE pointer
- Add HACCEL to HANDLE Database
Components – TestCases

- Instances of Library or System calls
  - Catalog definition + Fuzz values + Valid handles
  - Fuzz Mode: randomly selected
  - Repro-Min Mode: ordered according to logs
Components - Databases

- Handle Database
  - Stores valid handles created during run
  - Provides valid handles created during run
  - Maps handle values to creation conditions
- Log Database
  - Stores ordered sequence of testcases, fuzz values and handle values
- Dump Database
  - Stores, sorts and triages BSOD.dmps
  - FAILURE_ID_HASH_STRING and TIMESTAMP
Components – Logging (Fuzz Mode)

- Ordered sequence of testcases
- Arguments (fuzz values and handle values) of testcases
- Return values of testcases
- Log format
  - [thread_name] [module_name] [function_name] [function_arguments]

- Pitfall: Excessive logging!
  - 8MB to 2GB
  - Log offsets on binary template for suitable STRING type
  - Log only handle values on library/system call return
Components - Logging (Fuzz Mode) Example

......
t0:runTestCase:USER32_SetUserObjectInformationW(...,S[template_bin(0x0:0x40)],H[0x10])
t0:runTestCase:SC_NtGdiSetFontEnumeration(H[0x6])
t0:runTestCase:SC_NtGdiEndDoc(HANDLE[0x1011051])
t0:runTestCase:SC_NtGdiExtCreatePen(...,H[0x7],H[0x3],S["\xac\xb1\xfag"],...)
t0:runTestCase:handle => 0x1B00016

t0:runTestCase:USER32_OpenInputDesktop(H[0x1],H[0x1],H[0x1FF])
t0:runTestCase:handle => 0x0

t0:runTestCase:GDI32_CancelDC(HANDLE[0x121184C])
t0:runTestCase:SC_NtGdiSelectPen(HANDLE[0x2401073E],HANDLE[0x1B00016])
t0:runTestCase:handle => 0x1B00017

t0:runTestCase:USER32_CreateWindowStationW(H[0x0],H[0x0],H[0x37F],H[0x0])
t0:runTestCase:handle => 0x195C
......
++

Components – Logging (Repro-Min Mode)

- Tokenize log according to format and catalog syntax
  - [thread_name] [module_name] [function_name] [function_arguments]
  - HEX, STRING, HANDLE, STRUCTURE
Components – Logging (Repro-Min Mode) Example

Assign testcase to corresponding thread...

```
......
t0: runTestCase: SC_NtGdiExtCreatePen(...) 
    H[0xFFFFFFFF], H[0xD7], H[0x3], S["\x\x\x\x", ...]

......
t0: runTestCase: handle => 0x1B00016

......
t0: runTestCase: SC_NtGdiSelectPen(HANDLE[0x2401073E], HANDLE[0x1B00016])

......
t0: runTestCase: handle => 0x1B00017
```

**Fuzz Mode Log**

**Repro-Min Mode Log**
FRAMEWORK ARCHITECTURE AND COMPONENTS

++

Components – Logging (Repro-Min Mode) Example

• Assign testcase context...

```
......
t0:runTestCae:SC_NtGdiExtCreatePen(…,H[0xFFFFFFFF],H[0xD7],H[0x3],S['xac\x1b\xfag'],…)
t0:runTestCae:handle => 0x1B00016
t0:runTestCae:SC_NtGdiSelectPen(HANDLE[0x2401073E],HANDLE[0x1B00016])
t0:runTestCae:handle => 0x1B00017
......
```

Fuzz Mode Log

```
......
t0:runTestCae:
```

Repro-Min Mode Log
Components – Logging (Repro-Min Mode) Example

- Get catalog testcase in ordered sequence...

```
......
t0:runTestCase: SC_NtGdiExtCreatePen(…,H[0xFFFFFFFF],H[0xD7],H[0x3],S["xac\1b\xfag"],…)
t0:runTestCase: handle => 0x1B00016
```

```
t0:runTestCase: SC_NtGdiSelectPen(HANDLE[0x2401073E],HANDLE[0x1B00016])
t0:runTestCase: handle => 0x1B00017
```

```
......
t0:runTestCase: SC_NtGdiExtCreatePen
```

---

- Repro-Min Mode Log

- Fuzz Mode Log
FRAMEWORK ARCHITECTURE AND COMPONENTS

++

Components – Logging (Repro-Min Mode) Example

- Run testcase with corresponding arguments...

```
...  
t0:runTestCase:SC_NtGdiExtCreatePen(...,H[0xFFFFFFFF],H[0x7D],H[0x3],S["xac\x1b\xfag"],...)
t0:runTestCase:handle => 0x1B00016
```

```
...  
t0:runTestCase:SC_NtGdiSelectPen(HANDLE[0x2401073E],HANDLE[0x1B00016])
t0:runTestCase:handle => 0x1B00017
```

```
...  
t0:runTestCase:SC_NtGdiExtCreatePen(...,H[0xFFFFFFFF],H[0x7D],H[0x3],S["xac\x1b\xfag"],...)
```

Fuzz Mode Log

Repro-Min Mode Log
++

Components - Logging (Repro-Min Mode) Example

• Map handles in database...

```
......
t0:runTestCase: SC_NtGdiExtCreatePen(…, H[0xFFFFFFFF], H[0xD7], H[0x3], S["\xac\x1b\xfag"], …)
t0:runTestCase: handle => 0x1B00016
```

Repro-Min Mode Log

```
......
t0:runTestCase: SC_NtGdiExtCreatePen(…, H[0xFFFFFFFF], H[0xD7], H[0x3], S["\xac\x1b\xfag"], …)
t0:runTestCase: handle => 0xAABBCCDD
```

Fuzz Mode Log

```
......
t0:runTestCase: SC_NtGdiExtCreatePen(…, H[0xFFFFFFFF], H[0xD7], H[0x3], S["\xac\x1b\xfag"], …)
t0:runTestCase: handle => 0x1B00016
```

Handle-mapping

```
......
```

0x1B00016

0xAABBCCDD
FRAMEWORK ARCHITECTURE AND COMPONENTS

++

++

Components – Logging (Repro-Min Mode) Example

• Map handles in database...

```
......
t0:runTestCase: SC_NtGdiExtCreatePen(...,H[0xFFFFFFFF],H[0xD7],H[0x3],S["\xac\xb1\xfag"],...)
t0:runTestCase: handle => 0x1B00016
```

```
......
t0:runTestCase: SC_NtGdiSelectPen(HANDLE[0x2401073E],HANDLE[0x1B00016])
t0:runTestCase: handle => 0x1B00017
......
```

```
......
t0:runTestCase: SC_NtGdiExtCreatePen(...,H[0xFFFFFFFF],H[0xD7],H[0x3],S["\xac\xb1\xfag"],...)
t0:runTestCase: handle => 0xAABBCCDD
```

```
......
t0:runTestCase: SC_NtGdiSelectPen(HANDLE[0x2401073E],HANDLE[0xAABBCCDD])
```

Fuzz Mode Log

Repro-Min Mode Log

Handle-mapping

0x1B00016
0xAABBCCDD

……
Fuzzing the Windows Kernel

- FRAMEWORK ALGORITHMS
++

Fuzz Mode

1. Select library/system call from catalog
   a. Specific selection of testcases that create handles ("Trinity fuzzter")
   b. Random selection of testcases
2. Generate testcase arguments
3. Log testcase arguments
4. Run testcase
5. Log result
6. Repeat step 1

```python
def runTestCase(self, testcase):
    f = testcase
    f.generateArguments()
    arguments = f.serializeArguments()
    testcases_log.info("%s(%s)"%(test_name, arguments))
    f.runTest()
    if hasattr(f, "handle") : testcases_log.info("handle => 0x%X"%f.handle))
```
Repro-Min Mode

- No. of lines in typical logs: 15000 to 250000
- “setup group of testcases” vs “fuzzing group of testcases”

1. Generate set of “fuzzing group of testcases” (N)
2. Divide N into blocks (N/M)
3. Remove one block of testcases
4. Remove unreferenced “setup group of testcases”
5. Run all remaining blocks and check BSOD
6. Repeat step 2 until N/M=1
Fuzzing the Windows Kernel

- FRAMEWORK SETUP AND CONFIGURATION
FRAMEWORK SETUP AND CONFIGURATION

++

Setup And Configuration – Host

• Most basic hardware!
  – Spare machine laying around. Definitely can do better……😊

• Intel Xeon X3450, QuadCore @2.67 GHz
• 16 GB RAM
• Windows Server 2008 (x64) Standard
FRAMEWORK SETUP AND CONFIGURATION

++

Setup And Configuration - Guest

- 1 CPU, 2 GB RAM
- Windows 10 (x86) Pro
- Enable special pool for WIN32K.SYS
  - “verifier.exe /flags 0x1 /driver win32k.sys”
- Set BSOD MiniDump for disk-space saving
- Mapped drive to Host for MiniDumps and Logs
- Set normal Windows reboot
  - “bcdedit /set bootstatuspolicy IgnoreAllFailures”
Setup And Configuration – Scaling Up

• Guest is designed to be as self-contained as possible
• Effectively scaling up means spinning more Virtual-Machines
  – Use cloud
  – Buy more hardware ($$$)
• Need a Server–Client model to store MiniDumps and Logs centrally
Fuzzing the Windows Kernel

- RESULTS AND CASE STUDY
## Results

<table>
<thead>
<tr>
<th>Test Period</th>
<th>Jan 2016 – Mar 2016</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total BSOD</td>
<td>10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Error Description</th>
<th>Category</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>DRIVER_PAGE_FAULT_IN_FREED_SPECIAL_POOL (D5)</td>
<td>Use–After–Free</td>
<td>3</td>
</tr>
<tr>
<td>PAGE_FAULT_IN_NONPAGED_AREA (50)</td>
<td>Invalid Read</td>
<td>1</td>
</tr>
<tr>
<td>KMODE_EXCEPTION_NOT_HANDLED (1E)</td>
<td>Null Dereference</td>
<td>4</td>
</tr>
<tr>
<td>IRQL_NOT_LESS_OR_EQUAL (0A)</td>
<td>Miscellaneous</td>
<td>1</td>
</tr>
<tr>
<td>APC_INDEX_MISMATCH (01)</td>
<td>Miscellaneous</td>
<td>1</td>
</tr>
</tbody>
</table>
RESULTS AND CASE STUDY

Case Study – MiniDump.dmp

Use After-Free BSOD... 

For now, rem B853AD88 addr

BSOD due to library call gdi32.PlgBlt()

... reading from freed pool in DEVLOCKBLTOBJ destructor

Reference from MSDN
++

Case Study – Repro-Min

• Patched in MS16-062 (May 2016)
  – Bug Collision with one of these...
    • CVE-2016-0171 (Nils [bytegeist])
    • CVE-2016-0173 (Nils [bytegeist])
    • CVE-2016-0174 (Liang Yin [Tencent])
    • CVE-2016-0196 (Dhanesh [FireEye]; Vulcan Team [Qihoo 360])

• Reproduced and minimized after ~120 iterations
  – 14888 lines to 9 lines

• Analysis for this case study is performed on
  – win32kfull.sys v10.0.10586.71
  – win32kbase.sys v10.0.10586.20
RESULTS AND CASE STUDY

Case Study – Repro-Min

t0:runTestCase:GDI32_CreateICA(…)
t0:runTestCase:handle => 0x52109D0
t0:runTestCase:SC_NtGdiCreateMetafileDC(HANDLE[0x52109D0])
t0:runTestCase:handle => 0x22109D3
t0:runTestCase:SC_NtGdiCreateCompatibleBitmap(HANDLE[0x22109D3],…)
t0:runTestCase:handle => 0x60509D5
t0:runTestCase:GDI32_CreateICA(…)
t0:runTestCase:handle => 0x60109D4
t0:runTestCase:GDI32_CreateCompatibleDC(HANDLE[0x60109D4])
t0:runTestCase:handle => 0x40109DB
t0:runTestCase:GDI32_CreateICA(…)
t0:runTestCase:handle => 0x22109E7
t0:runTestCase:SC_NtGdiSelectBitmap(HANDLE[0x40109DB],HANDLE[0x60509D5])
t0:runTestCase:handle => 0x185000F
t0:runTestCase:SC_NtGdiDeleteObjectApp(HANDLE[0x60509D5])
t0:runTestCase:GDI32_PlgBlt(HANDLE[0x22109E7],…,HANDLE[0x40109DB],…,HANDLE[0x185000F],…)

Green: Handles related to hdcDest
Red : Handles related to hdcSrc
Blue : Handles related to hbmMask
RESULTS AND CASE STUDY

++

Case Study - Analysis

DEVLOCKBLTOBJ::~DEVLOCKBLTOBJ() is called from GrePlgBlt()...

GrePlgBlt()

DEVLOCKBLTOBJ object is also referenced in this code block...”call DEVLOCKBLTOBJ::bLock()”

Enlarge of code-block...

DEVLOCKBLTOBJ::~DEVLOCKBLTOBJ() is called from GrePlgBlt()...
RESULTS AND CASE STUDY

++

Case Study – Analysis

Observations:
1. DEVLOCKBLTOBJ is a local variable
2. Referenced in DEVLOCKBLTOBJ::bLock() without prior initialization
3. DCOBJ of hdcDest and hdcSrc are passed as 1st and 2nd arguments respectively
++

CASE STUDY

Case Study - Analysis

Observations from call-stack:
1. Pool B853AD88 is freed during deletion of SURFACE object (recall: this is the addr that was read and caused the BSOD)
2. SURFACE object is referenced from SURFREF object during hbmSelectBitmap()
3. We now know where in DEVLOCKBLTOBJ::bLock() would lead to ExFreePoolWithTag()
CASE STUDY

--

Case Study - Analysis

**Results and Case Study**

[Set logging breakpoint]:
1. `kd> bp nt!ExFreePoolWithTag`. `printf \"[ExFreePoolWithTag] P %08X, Tag %08X---|\"., poi(esp+4),......`

---

SURFACE-related pool is read from `[DCOBJ(hdcSrc)+1FC]`

Bitmap is selected from DCOBJ(hdcSrc)

[Remove logging breakpoint]:
1. `kd> bc`
RESULTS AND CASE STUDY

++

Case Study – Analysis Summary

• A DCOBJ object is instantiated from PlgBlt (..., hdcSrc, ...)

• The DCOBJ object is passed as 2nd argument in DEVLOCKBLTOBJ::bLock (..., DCOBJ_hdcSrc, ...)

• At BSOD faulting address, a de-reference is read from a freed pool; *[B853AD88h+14h]*

• Freed pool is used in 2 ways in DEVLOCKBLTOBJ::bLock()
  1. Copied from [DCOBJ+1FCh] to [DEVLOCKBLTOBJ+1Ch]
  2. Freed in SURFACE::bSelectSurface (..., B853AD88h, ...) during hbmSelectBitmap()

• Eventually, this SURFACE-related freed pool is referenced in DEVLOCKBLTOBJ::~DEVLOCKBLTOBJ() destructor, resulting in a Use-After-Free vulnerability

• Misc: DEVLOCKBLTOBJ is used in a ::bLock() -> ::~DEVLOCKBLTOBJ() manner
  – ::bLock() initializes and locks DEVLOCKBLTOBJ at the same time
Fuzzing the Windows Kernel

• CONCLUSION AND FUTURE WORK
Conclusion

• WIN32K.SYS as an attractive target for sandbox escapes

• Discussed about framework...
  – Architecture and Components
  – Algorithms
  – Setup and Configuration

• Effectiveness
  – Results from ~8 weeks of fuzzing
  – Demonstrated how this fuzzing could create a source HDC that would free a SURFACE-related pool during hbmSelectBitmap()
Future Work

• Server–Client (Distributed) Model

• WIN32k.SYS User–Mode Callbacks
  – “Kernel Attacks through User–Mode Callbacks” by Tarjei Mandt
  – “Analyzing local privilege escalations in win32k” by Thomas Garnier
CONCLUSION AND FUTURE WORK

++

Future Work

- Expand catalog for other .sys (then again WIN32K.SYS for sandbox escapes may not last long…)
  - Chrome’s DisallowWin32kSystemCalls

Win32k.sys lockdown:

- >= Win8
- ProcessSystemCallDisablePolicy, which allows selective disabling of system calls available from the target process.
- Renderer processes now have this set to DisallowWin32kSystemCalls which means that calls from user mode that are serviced by win32k.sys are no longer permitted. This significantly reduces the kernel attack surface available from a renderer. See [here](#) for more details.

- WIN32K.SYS syscall filter in Edge
Acknowledgements

- Nils
- James Loureiro
- Georgi Geshev
Thank You!

- Questions?