Use-After-Use-After-Free: Exploit UAF by Generating Your Own

Guanxing Wen

Background

The classic Flash exploit for many past years, regardless of the type of the bug, was mainly about corruption of the length field of Vector objects. For instance, a heap overflow exploit sprays Vectors and creates memory holes by freeing some Vectors. Vulnerable buffer is created to occupy one of the memory holes, corrupting the length field of a Vector object by triggering an overflow. A Vector with a large length is then utilized to search the process memory for ROP gadgets. Then ROP chain is triggered by a fake virtual function table.

For "use-after-free" bugs, the exploitation process is largely similar to that of heap overflow exploits. With a proper heap memory layout, vulnerable objects falling in one of the memory holes is released because of the "free" in "use-after-free" and occupied with controlled Vector data. After the member function of a dangling pointer is invoked, a fully controllable memory write will corrupt one of the length fields. Other aspects are similar to that of heap overflow exploits.

Mitigation

Since the length field is located at the beginning of the Vector buffer. It is convenient to be exploited through various Flash and even browser vulnerabilities. Length fields have been exploited more than three years now, since Adobe only patched the vulnerability itself instead of mitigating against the exploitation method. Eventually in December 2015, the popularity of Hack Teams' Flash exploit played a major role in Adobe's adoption of relevant mitigation. With the help of Google Project Zero, Adobe added mitigations to the Vector. With the mitigation, length field moves to the metadata part with only a verification cookie in the old position. The length field and the cookie now reside in different memory blocks, rendering simultaneous corruptions of both impractical.

One month after implementing the Vector mitigation, Adobe added verification fields to ByteArray, which is another popular array-like structure inside Flash. While the verification and length fields are still staying in the same block, although only rarely, powerful 0-day exploits including those used by P2O were able to read the verification field first and calculate the secret randomized value by applying XOR to the length and the verification field, and then corrupting them simultaneously. However, this behavior is highly dependent on the quality of the exploited vulnerability. For example, for a type-confusion bug, the ability to read and corrupt the verification cookie essentially implies that the bug itself is capable of accessing any part of the memory. Even without the corruption of ByteArray, the quality of the bug is sufficient to guarantee successful exploitations.

In addition to the length verification, isolated heap was also introduced into Flash. Its mechanism was first explained in Google Project Zero's blog. Basically, Object and Data are separated and allocated in different memory partitions. Even if a "use-after-free" bug exists, it is impossible to control the content of the freed object with Vector or ByteArray content as before. Nevertheless, it is still possible to occupy the freed objects' memory with class objects in the same partition, which is the type of exploit that we discuss below.

In summary, Adobe's mitigations had disarmed the classic and popular Flash exploit. In addition, there is yet another mitigation technique called Memory Protector, which was introduced into Flash version 22 this June. Since it does not affect our method nor the vulnerability that we are interested in, it is not discussed further here.

Use-After-Use-After-Free

What are the necessary conditions to exploit a “use-after-free” bug successfully? I think a read primitive helps a lot, which is commonly shared between previous Flash exploits. All modules are ASLR nowadays, hence with only a read primitive, one can find ROP gadgets and wrappers to set the executive bit of the shellcode. Specially for “use-after-free” type of exploits, we are able to fake the virtual function table by just occupation. A memory write is not a necessity.

Currently, exploits gain read primitives by tampering with the length field of array-like objects with large numbers, which contrasts with my approach to tamper with the start addresses. Also, Vectors and ByteArrays are not the only candidates. Other ActionScript classes also contain array-like structures inside. As a proof of concept, I will start with the simplest array structure, the String Object.

If a class holds a String inside, its structure is similar to that as depicted in Figure 1.

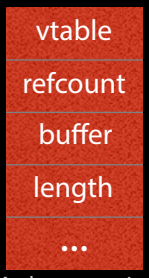

Figure 1. A class contains a String.

The String here is not an ActionScript “String”, otherwise it will be just an atom reference pointer here, instead of length and start address. Some class objects only store the String inside with no further operations, so they prefer a self-defined, light weighted String structure instead of ActionScript “String”.

Then, with the help of a “use-after-free” bug, we are able to release the object’s memory and occupy it with controlled content, where the start address is set to any desired value. Presently, we are able to access the target memory through the String object. After operating on the target memory, we assume that the object can be released again. With repeated occupations and releases, the String becomes a read primitive. This process is the main component of our exploitation method. Since there are many cycles of releases and occupations. Our exploitation method is named “use-after-use-after-free”.

The process described above is idealistic. Most frequently, we require one additional preparation step. During the initial steps of the “use-after-free” bug exploit, after the vulnerable class is released, we occupy it with our selected class object that is in the same size of the vulnerable class and holds a String structure. Because of the heap isolation, these two classes must reside in the same memory partition. When we invoke the virtual function with the dangling pointer, we invoke the function with the same offset but resides in the selected class’s virtual function table. The key point is that such type-confusion status is not expected by any of the functions and may help to release the selected class memory, resulting in two dangling pointers pointed to the same memory block. This step prepares us to perform the read primitive process.

Case Study

I discovered CVE-2016-1097 and reported it to Adobe in May. My approach employed a “use-after-free” bug of the “PSDK" class. “PSDK” belongs to the “mediacore" package, which was firstly seen in Flash version 19. I discovered this package by decompiling the player global file and cross-comparing it with its old version (Figure 2).

The package is undocumented, but there is some related information that can be found from Adobe Primetime Player SDK. This SDK is used to develop cross-platform TV based application. For the PC side,

the SDK is mainly a group of ActionScript files. My hypothesis is that, the “mediacore" package is a native implementation of this SDK inside Adobe Flash Player.

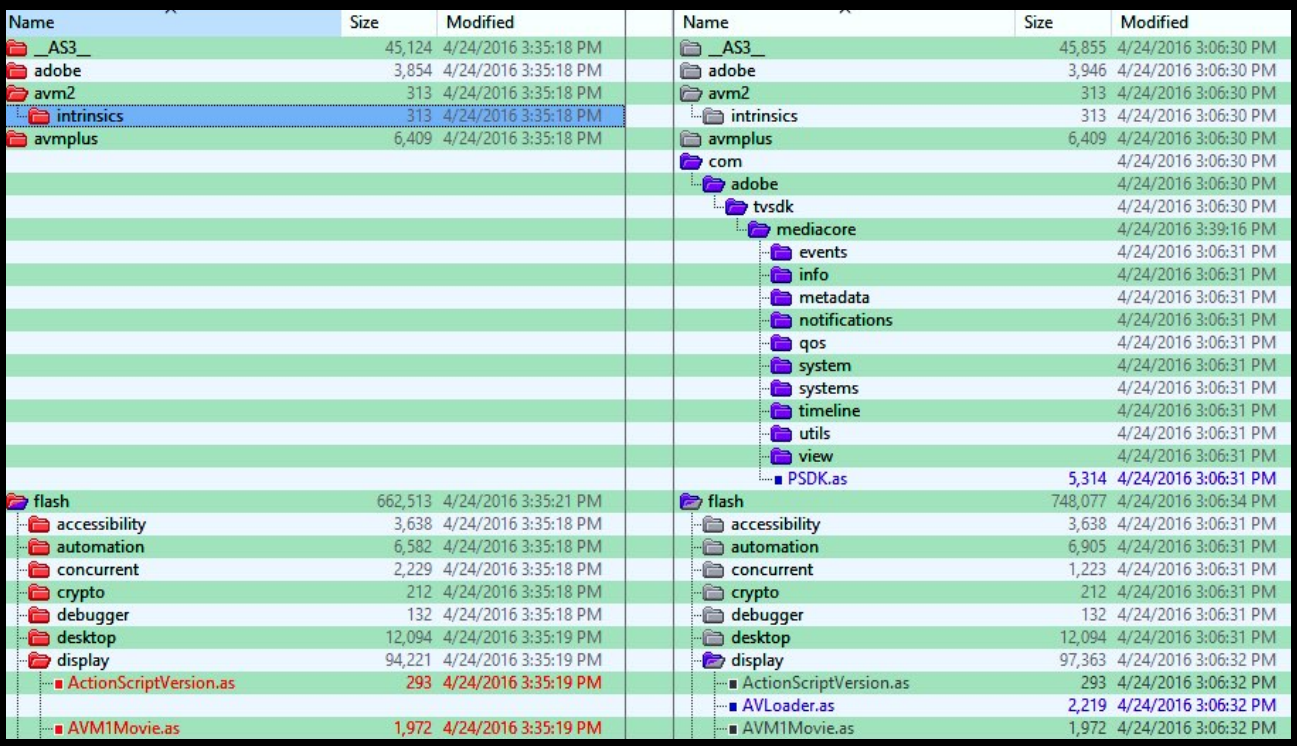

Figure 2. Cross-comparing of AS3 classes.

Apparently, this package has not been thoroughly tested as presented in Flash. I found many memory corruptions of this package in August of 2015. After reporting those issues to Adobe, they removed the entire package for half a year and reintroduced it in Flash version 21. The “use-after-free” bug we discuss below is found right after the reintroduction of “PSDK”. Below is a code snippet to trigger the “use-after-free” bug:

```
function poc()
{
    var ps:PSDK = PSDK.pSDK;
    ps.release();
    ps.createdispatcher();
}
```

The only way to get an instance of "PSDK" is through its static member property “PSDK". Apparently "PSDK" is constructed by the AVM (ActionScript Virtual Machine). Naturally, it should be destructed and cleaned by the AVM as well. The "PSDK" unexpectedly contains a method called “release()” that can be invoked directly from ActionScript level to explicitly release a class object’s memory. This is rarely the case of Flash objects, which normally depend on their reference counts of to become zero, while the garbage collector (MMgc), scans all objects periodically and opportunistically releases the class’s memory. Since the "PSDK" class is undocumented, we are uncertain if this manual release operation is necessary. However, after the class is released, its atom reference still points to the class memory. Hence we are facing the old text-book “use-after-free”.

32-bit exploit

"PSDK" requires 0x20 bytes in memory, all the ActionScript-level member functions are indexed in the first virtual function table, the destructor is indexed in the second one. The destructor can only be invoked by MMgc to release its class memory.

In order to occupy the memory of "PSDK" class, we have to choose a class from the same partition and with the same size as the "PSDK" class. I manually checked all the classes under the "mediacore" package. which are the most probable candidates. Classes under the same package are highly likely to be implemented in a consistent way, especially with regards to memory management.

Figure 3. Type-confusion function call

I settled on the "Track" class, which contains two string fields and is the same size as "PSDK". Newly constructed "Track" will then occupy freed memory of "PSDK". If we invoke "createDispatcher()" with the dangling pointer, it is the virtual functions of "Track" being indexed (Figure 3). This type-confusion function call decrements the second DWORD and releases current memory if the DWORD becomes zero. The second DWORD of current class is the length of the first String field. If we initialize "Track" with a

single character String, the length field shall be 1. After decrementing it, current memory will be released once again after “use-after-free”.
At the moment, we have a dangling pointer in the type of “Track”. To retrieve a read primitive, we need a mechanism to occupy and release current memory with controlled data freely.
“Metadata.setByteArray()” is a handy function that allow us to finish the task. “setByteArray()” duplicates input bytes inside “Metadata”. Since “Metadata” is also under the “mediacore" package, it shares the same heap partition with “Track”.
“setByteArray()” starts by allocating a temporary memory space to preprocess input bytes. This temporary space occupies the memory space of “Track”. Afterwards, a new buffer will be allocated to store the input bytes inside "Metadata". What makes this function desirable is that the temporary memory is subsequently released automatically at the end of “setByteArray()”. Hence our “Track”’s memory, which is filled with controllable content, will still be freed.
Recall that, after the "PSDK" class was released, “Track” took its position, then type-confusion function call released “Track” again.
With the help of “setByteArray()”, we can easily occupy and corrupt the start address of the String object. Each time we could set the 0x10 offset (start address of the second String field of “Track”) of the ByteArray to be the target address, then “setByteArray()” will push the data over “Track”. Subsequently, accessing “track.language” returns the content of target memory.

```
bytes.postion = 0x10;
bytes.writeUnsignedInt(0xadd7e555);
mt.setByteArray("address", bytes);
res = track.language;
value = (res.charCodeAt(3)<<24);
value|= (res.charCodeAt(2)<<16);
value|= (res.charCodeAt(1)<<8);
value|= (res.charCodeAt(0));
```

This above code can be looped as needed to read arbitrary memory and permits us to read any place inside the process memory.
We can simply read a stable location sprayed with “this” (current class pointer). With the current class pointer, we can find self-defined variables like ByteArray or Vector (used to store shellcode and fake virtual function table) starting from the 0x54 offset. With virtual function tables of any found variables, we can find the module base address of Flash and begin searching the ROP chain. Such technique is well-explained in Hack Team’s Flash exploit. It is an efficient process with less brute force searching compared to other known techniques.
After every buffer is filled and linked, “setByteArray()” is invoked one more time to corrupt the second virtual function table. We corrupt the second virtual function table instead of the first one because the release of the temporary memory space brought by “setByteArray()” has a minor side effect of modifying the first byte of current memory. Therefore, we cannot fully control the first virtual function table. After all the ActionScript code in our exploit is executed, MMgc will attempt to release the memory of "PSDK" by invoking the destructor inside the second virtual function table that holds the ROP chain address.

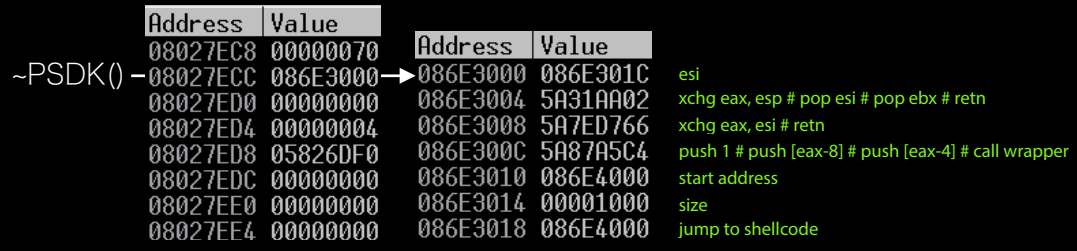

Figure 4. Trigger the ROP via second virtual function table of "PSDK".

ROP chain simply invokes a wrapper inside Flash (Figure 4). The wrapper code will call VirtualProtect to set the executive bit of the shellcode memory then jump over. Searching pattern for this wrapper remains unchanged since CVE-2014-0515 to Flash version 21. Beginning with Flash version 22, searching pattern is push [eax - 0x0C], push [eax - 0x08], compared to the original push [eax - 0x08], push [eax - 0x04]. This change is necessary due to the Memory Protector, which is a new mitigation to delay the release of objects' memory spaces. An extra field is added to this wrapper function. We will discuss this mitigation later in the conclusion part.

64-bit Exploit

Principally, memory layouts of Flash objects are different between the 64-bit and the 32-bit versions. The pointers in the 64-bit version increase in size from 4 bytes to 8 bytes. As a result, "PSDK" and "Track" are no longer the same size. We resort to choose "MediaResource" class to occupy the memory of "PSDK". "MediaResource" is also under "mediacore" package and contains a String object inside. See code snippets below.

```
ps = PSDK.pSDK;
ps.release();
ms = new MediaResource("jack", 0x54336677, null);
try{
    ps.createDefaultContentFactory();
}catch(e:Error){}
```

This type-confusion function call treat offset 0xF0 of current memory as a reference count (Figure 5), decrements it and release "MediaResource" if it is zero.

```
__int64 __fastcall sub_305374C0(__int64 a1)
{
  void *v1; // rax@3
  unsigned int v3; // [sp+20h] [bp-
  __int64 v4; // [sp+50h] [bp+8h]@1

  v4 = a1;
  v3 = sub_30739190((volatile signed __int32 *)(a1 + 0xF0));
  if ( !v3 )
  {
    if ( v4 )
    {
      LODWORD(v1) = (*(int (__fastcall **)(__int64, _QWORD))(*(_QWORD *)v4 + 0x10i64))(v4, 0i64);
      j_free(v1);
    }
    else
    {
      j_free(0i64);
    }
  }
  return v3;
}

__int64 __fastcall sub_30739190(volatile signed __int32 *a1)
{
  return (unsigned int)_InterlockedDecrement(a1);
}
```

Figure 5. Type-confusion function call under 64-bit.

While 0xF0 offset is out of bounds of the current class, we cannot control that part of memory with member property of "MediaResource" as we did with "Track" under 32-bit versions. However, we can still control that part of memory by occupying it with "setByteArray()" before "PSDK" is released.

```
var bytes:ByteArray = new ByteArray();
bytes.endian = "littleEndian";
bytes.position = 0x30;
```

```
bytes.writeInt(1);
mt.setByteArray("jack", bytes);
```

There are two reasons why we can simply invoke “setByteArray()” once to occupy the 0xF0 memory. The first reason is that "PSDK", “Track” and other classes under the “mediacore “package are using a different heap partition compared to other traditional Flash objects. Traditional Flash objects are completely managed by MMgc on self-maintained Flash heaps. Whereas, "PSDK" and “Track” are managed directly via “malloc()” and “free()” on the default/CRT heap. It is possible that this is a new trend in Adobe Flash, moving self-managed VirtualAlloc objects to the default heap. This new memory partition is rather pure in composition. After "PSDK" is initialized, the memory after "PSDK" are not interfered by traditional Flash objects. When “setByteArray()” is invoked, it is the first time the memory after "PSDK" is allocated. The second reason is that “setByteArray()” allocates two pieces of memory, one is temporary and the other one storing inside "Metadata". In combination with the memory of “PSDK”, the memory blocks associated with “setByteArray()” cover the 0xF0 memory.

After “MediaResource” is released, a read primitive is gained. However, this time we cannot start by reading a sprayed Vector’s memory, since heap sprays of normal Flash objects are high 32-bit randomized.

Fortunately, I found that spraying malloced Flash objects under Windows 7 does not lead to sufficiently high entropies of heap randomization (hight 32-bit remains zero). To spray malloced Flash objects, I choose another class, “AdAsset “under “mediacore", which works like an array and is capable of storing different kinds of objects. If we create “AdAsset” many times, classes inside them will duplicate themselves and cover a stable memory location. One of the classes inside “AdAsset” is “MediaResource”, which also contains an integer value that can be used as a flag.

```
gc_arr = new Array();
ad = new AdClick("","","");
ms = new MediaResource("jack",0x54336677,null);
mt = new MetaData();
for(var i=0;i<0x80000;i++)
{
    gc_arr[i]=new AdAsset("",1,ms,ad,mt);
}
```

From the sprayed memory we can locate our flag and thus “MediaResource” with adjacent "Metadata". "Metadata" can be a replacement of ByteArray or Vector to store the shellcode, the ROP chain and the fake virtual function table. Every set of bytes stored inside can be indexed via fixed offsets. Notice that the offset for the second step is calculated via a hash function. The buffer we need can be stored in different locations via different key names. The hash value can only be 0-7. We want to avoid a hash collision, or else there will be a list structure inside "Metadata", increasing the difficulty for us to locate the buffer. As we do not need many buffers, we choose proper key names to ensure unique hash values. With the known address buffer, the fake virtual function table trick operates similarly to that as applied to 32-bit versions.

Windows 10 Exploit

Currently, all experiments are conducted under Windows 7. I do not intend to build a fully workable exploit under Windows 10. Instead, I will provide a quick peek to illustrate some key differences.

As usual, "PSDK" is allocated via “malloc()” on the LFH (Low Fragment Heap). However, the LFH has evolved from Windows 7 to Windows 10. Heap randomization is added into LFH in Windows 10, which makes heap occupation in Windows 10 compared to Windows 7 less predictable. For Windows 7, after a memory block is freed, it is tabulated at the end of the memory block free-list. When next allocation

occurs, the most recently freed block will be the next one allocated. For Windows 10, there is a bitmap controlling the allocation, but we can still use the free-list model to understand its operations. When a memory block is freed, it is added to the memory block free-list. If for example, there are "n" blocks on the free-list, when the next allocation occurs, LFH will choose one of these blocks randomly and hence we cannot determine if the previously freed block is occupied or not with one allocation.

I deduced via black-box experimentations with heaps that the occupation can still work with multiple attempts, but "n" allocations do not yield full occupation. After "n-1" normal allocations, LFH will allocate a new free-list and randomly choose one of its blocks to allocate for "n"-th try, in order to increase the randomness. However, the extent of randomization is limited, as after two to three cycles of free-list allocations, LFH realizes that there is still one free block on the first free-list and makes an attempt to allocate it, thus giving us eventual occupation. My deductions corroborate well with reverse-engineered results by others of "ntdll.dll".

After "PSDK" is released, we can construct "Track" hundreds of times to ensure that it eventually occupies the memory. Because of the Flash object layout under Windows 10, we invoke "createAdPolicySelector()".

```
ps.release();
for(i=0;i<0x100;i++)
    track = new Track("j","lan",true,true);
ps.createAdPolicySelector(1,mp);
```

This type-confusion function call is simpler to control than Windows 7, as we do not need to control any part of memory. We only need to pass the parameter "1" and "Track" will be released (Figure 6). Then we are in the read primitive situation once again. For the read primitive part, each cycle of "setByteArray()" need to be looped multiple times as we did with the "PSDK" occupation. This is the extent of my research on this type exploit under Windows 10.

```
LPVOID __thiscall sub_10080240(LPVOID lpMem, char a2)
{
  LPVOID v2; // esi@1

  v2 = lpMem;
  *(_DWORD *)lpMem = off_10FC6C80;
  if ( a2 & 1 )
    sub_10A294D5(lpMem);
  return v2;
}
```

```
if ( lpMem )
{
  if ( !HeapFree(hHeap, 0, lpMem) )
  {
    v1 = (_DWORD *)sub_10DAB7D1();
    v2 = GetLastError();
    *v1 = sub_10DAB758(v2);
  }
}
```

Figure 6. Type-confusion function call under Windows 10.

It seems that Windows 10 presents a tough environment for us to exploit, but our "use-after-use-after-free" exploitation technique may still find its way to a read primitive with a proper approach.

Conclusion

After my submission of the discovered bug to Adobe, they chose to manually remove the associated atom reference at first in a manner that can be bypassed. Hence producing yet another bug

CVE-2016-4248. If we declare two references and invoke “release()” with one of them and trigger a “use” with the other, then the “use-after-free” bug still exists.

The modification to the triggering code involves changing one line. All the remaining exploitation code is absolutely the same. As it was Flash version 22 by then, Adobe had introduced the Memory Protector into Flash, presumably a mitigation technique learnt from Microsoft, which is used to delay the release of class objects. However it does not affect the “use-after-free” exploitation of “PSDK”, perhaps because it is a mitigation technique only focused on class objects allocated by MMgc. I conducted no further analysis was conducted on this mitigation technique, as it does not interfere with our exploitation procedures.

Subsequently, Adobe completely abandoned the “release()” function for the next patch.

Our “use-after-use-after-free” is a relatively common way to exploit an “use-after-free” bugs. The idea behind this method is adaptable to other platforms. I have seen such occupation/release and type-confusion exploits targeting an iOS kernel. Although the first step to released the selected class via a type-confusion call may seem complicated, I have demonstrated that it is possible under three different Flash versions, across both 32-bit and 64-bit versions under Windows 7 and a 32-bit version of Windows 10.

As I have implemented, the occupation and release cycle deals with a String object and thus give us a read primitive. Extending this idea in the future, we may find other array structures to replace String and hopefully gain the ability to write primitives, which are helpful to overcome mitigations under Windows 10.