50 CVEs in 50 Days: Fuzzing Adobe Reader

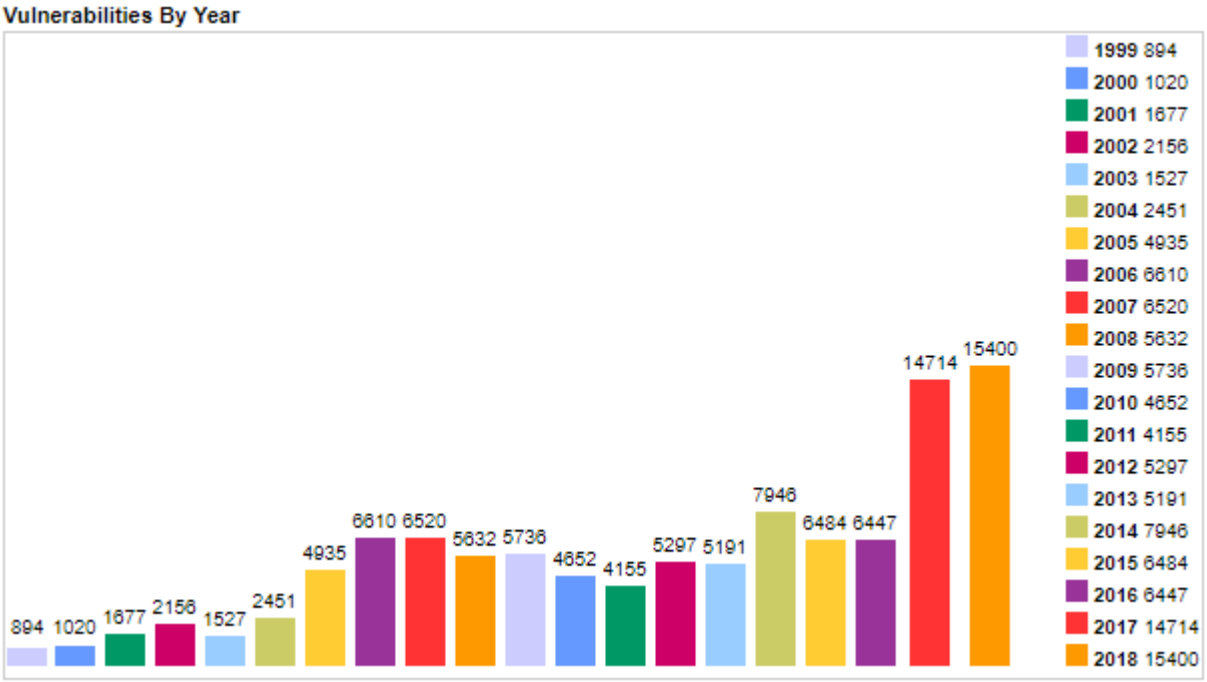
research.checkpoint.com/50-adobe-cves-in-50-days

December 12, 2018

December 12, 2018
Research By: Yoav Alon, Netanel Ben-Simon

Introduction

The year 2017 was an inflection point in the vulnerability landscape. The number of new vulnerabilities reported that year was around 14,000, which is over twice the number from the year before (see table below). The probable reason for this is the increased popularity of automatic vulnerability finding tools, also known as “fuzzers”.



The mere existence of fuzzers is not breaking news; they've been around for over two decades. The news is that fuzzers have grown up. They've become more capable, more accessible, and overall more mature. Still, using fuzzers has somewhat of a reputation as a "dark art", a lot of researchers don't bother with fuzzers because they are perceived as a hassle to use.

Given all the above, we found it natural to ask: Yes, more researchers are using fuzzers to find more vulnerabilities — but are *all* the researchers using fuzzers to find *all* the vulnerabilities? How many low-hanging fruits are still out there, just waiting for the first person to press the big shiny button that says 'FUZZ'?

To find out, we constructed the most vanilla experiment we could think of. We took one of the most common Windows fuzzing frameworks, WinAFL, and aimed it at Adobe Reader, which is one of the most popular software products in the world. We set a time-frame of 50 days for the entire endeavor – reverse-engineering the code, looking for potential vulnerable libraries, writing harnesses and, finally, running the fuzzer itself.

The results left us flabbergasted. In those 50 days, we were able to find over 50 new vulnerabilities in Adobe Reader. On average, that's 1 vulnerability per day — not quite the usual pace for this kind of research.

In this paper, we lay out the entire story of this research. We share a novel methodology we used to increase the scope of our search, improvements that we were able to make to WinAFL, and, finally, the insights we gained along the way.

What is WinAFL?

AFL is a coverage guided genetic fuzzer, which has a rock solid implementation and clever heuristics that have proven to be very (!) successful in finding real bugs in real software.

WinAFL is a fork of AFL for Windows, created and maintained by Ivan Fratric (Google Project Zero). The Windows version uses a different style of instrumentation which enables us to target closed source binaries.

We recommend reading the AFL technical paper, which goes into detail on how AFL works. It also points out the tool's shortcomings and helps you debug when things go wrong.

We found WinAFL to be extremely effective in finding file format bugs, especially in compressed binary formats (images / videos / archives).

Attacking Acrobat Reader DC

The easiest place to start is the main executable, AcroRd32.exe. This is a (relatively) thin wrapper around AcroRd32.dll, which is approximately 30MB in size. AcroRd32.dll has a lot of code, some of which contains parsers for PDF objects, but a lot of it is GUI code (not the place you usually want to look for bugs).

We know that WinAFL is better at binary formats, so we decided to focus our efforts and attack a specific parser. The challenge is to locate a parser and write a harness for it. We'll explain what exactly is a harness is a bit later on.

We want a binary format parser with minimal dependencies that we can load without also loading the entire Reader process.

We explored the DLLs in Acrobat's folder and found that JP2KLib.dll fits all categories:

Property	Value
CompanyName	Adobe Systems Incorporated
FileDescription	Adobe JPEG2000 Core Library
FileVersion	1.2.2.39492
InternalName	JP2KLib
LegalCopyright	© 2009 Adobe Systems Incorporated. All Rights Reserved.
LegalTrademarks	Adobe ®
OriginalFilename	JP2KLib.dll
ProductName	JP2KLib 2017/10/28-02:37:53
ProductVersion	84.268563
BuildDate	2017/10/28-02:37:53
BuildVersion	84.268563
BuildType	RELEASE
BuildID	39492
JP2K_IPID	<AdobeIP#0000435>

JP2KLib.dll is a parser for the JPEG2000 format, which is a complex binary format (753 KB), and has exported functions which are pretty descriptive.

Ordinal	Function RVA	Name Ordinal	Name RVA	Name
(nFunctions)	Dword	Word	Dword	szAnsi
00000046	00064BB9	0045	000B165A	JP2KImageDataSetSizeOfImageDataType
00000047	00064BCD	0046	000B167E	JP2KImageDataSetWidth
00000048	00064BE6	0047	000B1694	JP2KImageDecodeImageRegion
00000049	00064C60	0048	000B16AF	JP2KImageDecodeImageWithoutSeeking
0000004A	00064CD1	0049	000B16D2	JP2KImageDecodeTileInterleaved
0000004B	00064D4B	004A	000B16F1	JP2KImageDecodeTileInterleavedIncremental
0000004C	00064DC8	004B	000B171B	JP2KImageDecodeTilePlanar
0000004D	00064E3F	004C	000B1735	JP2KImageDestroy
0000004E	00064EB4	004D	000B1746	JP2KImageEncodeImage
0000004F	00064F25	004E	000B175B	JP2KImageEstimatePeakDecodeMemoryReq...

The research was conducted on the following versions:
Acrobat Reader DC 2018.011.20038 and earlier
JP2KLib.dll version 1.2.2.39492

What Is a Target Function?

A target function is a term that WinAFL uses to describe the function that is used as the entry point to the fuzzing process. The function is called in a loop for *fuzz_iterations* times, each time mutating the input file on disk. The function must:

- Open the input file, read the file, parse the input and close the file.
- Return normally – without throwing a *C++* exception or calling *TerminateProcess*

Finding such a function in nature is pretty uncommon. When targeting a complex software, we usually need to write a harness.

What Is a Harness?

A harness is a small program that triggers the functionality we want to fuzz. The harness includes a function which will be used as our target function. Here's an example for a minimal harness for gdiplus from the WinAFL repository:

```
int main(int argc, char** argv)
{
    if(argc < 2) {
        printf("Usage: %s <image file>\n", argv[0]);
        return 0;
    }

    GdiplusStartupInput gdiplusStartupInput;
    ULONG_PTR gdiplusToken;
    GdiplusStartup(&gdiplusToken, &gdiplusStartupInput, NULL);

    Image *image = NULL, *thumbnail=NULL;

    // open and parse the input path
    image = new Image(charToWChar(argv[1]));
    if(image && (Ok == image->GetLastStatus())) {
        // ...
    }

    // cleanup
    if(image) delete image;
    if(thumbnail) delete thumbnail;

    GdiplusShutdown(gdiplusToken);

    return 0;
}
```

The first argument to *main* is a path. Within the function, we call the *Image::Image* parser, which is the API we wanted to fuzz. Notice that in error cases, we don't terminate the process and at the end we free all resources.

This process is relatively easy for documented API. We can copy a sample code or write a simple program using the documentation. But where's the fun in that?

We chose to target Adobe Reader, which is a closed sourced binary. The process of writing a harness for this type of target looks something like this:

1. Find the functionality we want to fuzz.
2. Reverse-engineer it a little bit.
3. Write a program that calls the reversed API.
4. Repeat until we have a fully functional harness.

In the following section, we describe in detail how we reverse-engineered JP2KLib, and wrote a working harness for it. We also share a few tips. Readers who are only interested in our fuzzing methodology can skip to the next section.

Writing a Harness for JP2KLib.dll

Before starting to reverse-engineer JP2KLib.dll, we checked whether the library is open source or has public symbols. This is a big time saver and is more common than you might think. But in our case, we weren't so lucky.

As we wanted our harness to be as similar as possible to how Adobe Reader uses JP2KLib, the first thing we had to do is to find a PDF file that triggers the behavior we wanted to fuzz. This enables us to easily locate the relevant parts of the program.

In our case, we have a large corpus of PDFs for testing our products. We *grep*ped the string "/JPXDecode", which is the PDF filter for JPEG2000, and used the smallest example that came up. We could also have googled for a sample file or used Acrobat Pro / Phantom PDF to generate a test case.

Pro Tip 1: *The reader has a sandbox, which is sometimes annoying for debugging/triaging, but this can be disabled – https://forums.adobe.com/thread/2110951*

Pro Tip 2: *We turned on PageHeap to assist in the reverse-engineering efforts, as it helps in tracking allocation place and size.*

We extracted the jp2 file from our sample, so we could use it for our harness without the PDF wrapper. This will be used as our testing input for the harness.

Now that we had a minimal working example, we placed a breakpoint on the load event of JP2KLib.dll using *"sxe ld jp2klib"*. When the breakpoint hit, we placed a breakpoint command on all exported functions of JP2KLib. The breakpoint command logs the call stack, the first few arguments, and the return value:

bm /a jp2klib! ".echo callstack; k L5; .echo parameters:; dc esp L8; .echo return value: ; pt; "*

We loaded the sample PDF and got the following output:

```
callstack
 # ChildEBP RetAddr
WARNING: Stack unwind information not available. Following frames may be wrong.
00 008fa228 600a6dec JP2KLib!JP2KLibInitEx
01 008fa324 6009d489 AcroRd32!AX_PDXlateToHostEx+0x26204b
02 008fa340 5fc8f6b3 AcroRd32!AX_PDXlateToHostEx+0x2586e8
03 008fa384 5fc8ed83 AcroRd32!PDAlternatesGetCosObj+0x77223
04 008fa454 5fc8e111 AcroRd32!PDAlternatesGetCosObj+0x768f3
parameters:
008fa22c  600a6dec 47e82fe0 43aacfac 600a6fc7  .m.`./.G...C.o.`
008fa23c  efe6e483 43f6af70 43aacfac 00000000  ....p..C...C....
return value:
eax=00000000 ebx=00000000 ecx=00000000 edx=01000002 esi=47e82fe0 edi=43f6af70
eip=4de148af esp=008fa22c ebp=008fa324 iopl=0         nv up ei pl zr na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000246
JP2KLib!JP2KLibInitEx+0x1d:
4de148af c3              ret
```

JP2KLibInitEx is the first function called after loading JP2KLib. We noticed that JP2KLibInitEx takes only one argument. Let's examine it:

```
0:000> dps 47e82fe0
47e82fe0  600a6cd6 AcroRd32!AX_PDXlateToHostEx+0x261f35
47e82fe4  600a6d2f AcroRd32!AX_PDXlateToHostEx+0x261f8e
47e82fe8  5fb155c0 AcroRd32!CTJPEGLibInit+0x19120
47e82fec  600a6d10 AcroRd32!AX_PDXlateToHostEx+0x261f6f
47e82ff0  5fd702e2 AcroRd32!CTJPEGDecoderRelease+0xa992
47e82ff4  5faf37a2 AcroRd32!AcroWinMainSandbox+0x1981e
47e82ff8  600a6d5f AcroRd32!AX_PDXlateToHostEx+0x261fbe
47e82ffc  600a6d78 AcroRd32!AX_PDXlateToHostEx+0x261fd7
47e83000  ????????
47e83004  ????????
```

We can see that it's a struct of size 0x20 and it contains pointers to functions in AcroRd32.dll. When we encounter an unknown function, we don't rush into reversing it as we don't know if it's going to be used by the target code. Instead, we point each address to a unique empty function we call "nopX" (where X is a number).

We now have enough information to start writing our harness skeleton:

1. Get input file from command line arguments.
2. Load JP2KLib.dll.
3. Get a pointer to JP2KLibInitEx and call it with a struct of 8 nop functions.

```
int main(int argc, char ** argv)
{
    if (argc < 2) {
            printf("Usage %s: <input file>\n", argv[0]);
            return 1;
    }

    vtable_t vtbl;

    vtbl.funcs[0] = nop0;
    vtbl.funcs[1] = nop1;
    vtbl.funcs[2] = nop2;
    vtbl.funcs[3] = nop3;
    vtbl.funcs[4] = nop4;
    vtbl.funcs[5] = nop5;
    vtbl.funcs[6] = nop6;
    vtbl.funcs[7] = nop7;

    HMODULE hJp2klib = LoadLibraryA("JP2KLib.dll");
    if (hJp2klib == NULL) {
            dbg_printf("failed to load library, gle = %d\n", GetLastError());
            exit(1);
    }

    LOAD_FUNC(hJp2klib, JP2KLibInitEx);

    int ret = JP2KLibInitEx_func(&vtbl);
    dbg_printf("JP2KLibInitEx: ret = %d\n", ret);

    return 0;
}
```

We use LOAD_FUNC as a convenience macro. We also have a NOP(x) macro for creating nop functions.

```
// Macro to help to loading functions
#define LOAD_FUNC(h, n)                                  \
  n##_func = (n##_func_t)GetProcAddress(h, #n);          \
  if (!n##_func) {                                       \
    dbg_printf("failed to load function " #n "\n"); \
    exit(1);                                             \
  }

// Macro help creating unique nop functions
#define NOP(x)                                                        \
  int nop##x() {                                                      \
    dbg_printf("==> nop%d called, %p\n", ##x, _ReturnAddress()); \
    return (DWORD)x;                                                  \
  }
```

We compile, run it with sample.jp2 – and it works!

Let's continue ("g"). We then move to the next function JP2KGetMemObjEx which doesn't take any arguments, so we call it and save the return value.

The next function JP2KDecOptCreate also doesn't take any arguments, so we call it and save the return value. However, we notice that JP2KDecOptCreate internally calls nop4 and nop7, which means we need to implement them.

Our next move is to understand what "nop4" does. We placed a breakpoint on the original function pointer to "nop4" *AcroRd32!CTJPEGDecoderRelease+0xa992* and continued execution:

```
0:000> bu AcroRd32!CTJPEGDecoderRelease+0xa992; g;
Breakpoint 0 hit
eax=008fa2b8 ebx=00000000 ecx=09894f9c edx=01000002 esi=47e82fe0 edi=43f6af70
eip=5fd702e2 esp=008fa234 ebp=008fa324 iopl=0         nv up ei pl nz na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000206
AcroRd32!CTJPEGDecoderRelease+0xa992:
5fd702e2 55              push    ebp
0:000> u
AcroRd32!CTJPEGDecoderRelease+0xa992:
5fd702e2 55              push    ebp
5fd702e3 8bec            mov     ebp,esp
5fd702e5 5d              pop     ebp
5fd702e6 e92ed4d6ff      jmp     AcroRd32!AcroWinMainSandbox+0x3795 (5fadd719)
```

Which took us to:

```
0:000> u AcroRd32!AcroWinMainSandbox+0x3795
AcroRd32!AcroWinMainSandbox+0x3795:
5fadd719 55              push    ebp
5fadd71a 8bec            mov     ebp,esp
5fadd71c a194e20261      mov     eax,dword ptr [AcroRd32!defaultCTJPEGMemoryManager+0x4a9e70 (6102e294)]
5fadd721 5d              pop     ebp
5fadd722 ff600c          jmp     dword ptr [eax+0Ch]
```

And after few steps:

```
0:000> p
eax=6102e2bc ebx=00000000 ecx=09894f9c edx=01000002 esi=47e82fe0 edi=43f6af70
eip=5fadd722 esp=008fa234 ebp=008fa324 iopl=0         nv up ei pl nz na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000206
AcroRd32!AcroWinMainSandbox+0x379e:
5fadd722 ff600c          jmp     dword ptr [eax+0Ch]  ds:002b:6102e2c8={MSVCR120!malloc (7137ed10)}
```

So it turns out that nop4 is a thin wrapper around malloc. We implemented it in our harness and replaced it with “nop4”. We repeated this procedure again for nop7 and found out it was *memset*!. We looked around and saw that nop5 and nop6 and were *free* and *memcpy* respectively.

The next function, JP2KDecOptInitToDefaults, was called with one argument. This was the return value from JP2KDecOptCreate, so we passed the value to it.

The next function, JP2KImageCreate, takes no arguments, so we called it and saved the return value.

Currently, our harness looks something like this:

```
void* mem_obj = JP2KGetMemObjEx_func();
dbg_printf("JP2KGetMemObjEx: ret = %p\n", mem_obj);

image_t dec_opt = JP2KDecOptCreate_func();
dbg_printf("dec_opt = %p\n", dec_opt);

ret = JP2KDecOptInitToDefaults_func(dec_opt);
dbg_printf("JP2KDecOptInitToDefaults: ret = %d\n", ret);

image_t image = JP2KImageCreate_func();
dbg_printf("image = %p\n", image);
```

The next function is JP2KImageInitDecoderEx, which takes **5!** arguments.

We matched 3 out of 5 parameters to return values from: JP2KImageCreate, JP2KDecOptCreate and JP2KGetMemObjEx

We noticed that the 3rd parameter points to a vtable. We did the same trick as before – created a struct with the same size that points to “nop” functions.
The 2nd parameter points to another struct, only this time it doesn’t seem to contain function pointers. We decided to send the const value 0xbaaddaab.

At this point the code looks like this:

```
    procs.func4 = nop4;
    procs.func5 = nop5;
    procs.func6 = nop6;
    procs.func7 = nop7;
    procs.func8 = nop8;
    procs.func9 = nop9;
    procs.func10 = nop10;
    procs.func11 = nop11;
    procs.func12 = nop12;
    procs.func13 = nop13;

    file_obj_t fobj;
    init_file_obj(&fobj, input_file);

    dbg_printf("file obj = %p\n", &fobj);

    ret = JP2KImageInitDecoderEx_func(image, &fobj, (vtable_t *)&procs, dec_opt, mem_obj);
    dbg_printf("JP2KImageInitDecoderEx = %d\n", ret);
    if (ret != NULL) {
        dbg_printf("failed to decode\n");
    }
```

We ran our harness and quickly arrived at nop10. We set a breakpoint in Adobe Reader on the corresponding function and got to the following call stack:

```
0:000> k
 # ChildEBP RetAddr
WARNING: Stack unwind information not available. Following frames may be wrong.
00 008f9f48 4ddd90dd AcroRd32!AX_PDXlateToHostEx+0x2614ca
01 008fa06c 4de16482 JP2KLib!JP2KCodeStm::IsSeekable+0x9
02 008fa0c4 600a68c6 JP2KLib!JP2KImageInitDecoderEx+0x24
03 008fa14c 600a8260 AcroRd32!AX_PDXlateToHostEx+0x261b25
04 008fa1ac 6009d40e AcroRd32!AX_PDXlateToHostEx+0x2634bf
05 008fa1b8 600a620a AcroRd32!AX_PDXlateToHostEx+0x25866d
06 008fa1cc 5fc8f6c6 AcroRd32!AX_PDXlateToHostEx+0x261469
```

Looking at JP2KCodeStm::IsSeekable in IDA:

```
; bool __thiscall JP2KCodeStm::IsSeekable(JP2KCodeStm *__hidden this)
                public public: bool __thiscall JP2KCodeStm::IsSeekable(void)
public: bool __thiscall JP2KCodeStm::IsSeekable(void) proc near
                                        ; CODE XREF: sub_1000AEF3+65↑p
                                        ; sub_1000B9BB+9↑j ...
                mov     eax, [ecx+24h]
                push    dword ptr [ecx+18h]
                call    dword ptr [eax+18h]
                pop     ecx
                retn
public: bool __thiscall JP2KCodeStm::IsSeekable(void) endp
```

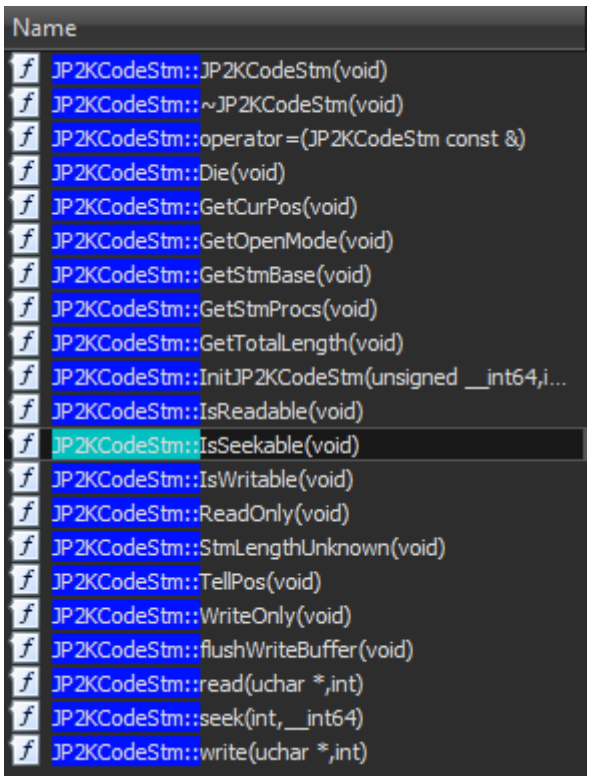
Looking at WinDbg, we can see that JP2KCodeStm at offset 0x24 contains our vtable and offset 0x18 contains 0xbaaddaab. We can see that JP2KCodeStm::IsSeekable calls a function from our vtable passing 0xbaaddaab as the first parameter, so it's basically a thin wrapper around our vtable function #7.

In general, every parser is a bit different but usually they consume an input stream which could be in a familiar file interface (like FILE / ifstream). More often than not, it's some sort of a custom type that abstracts the underlying input stream (network / file / memory). So when we saw how JP2KCodeStm was used, we knew what we were looking at.

Back to our case, 0xbaaddaab is the stream object and the vtable functions operate on the stream object.

We moved to IDA and looked at all the other JP2KCodeStm::XXX functions.

They were all very similar, so we went ahead and created our own file object, and implemented all the necessary methods. The resulting code looks like:




```
procs.nop_return_a0 = (void *)nop_ret_a0;
procs.die = (void *)file_obj_die;
procs.read = (void *)file_obj_read;
procs.write = (void *)file_obj_write;
procs.seek = (void *)file_obj_seek;
procs.tell_pos = (void *)file_obj_tell;
procs.is_seekable = (void *)file_obj_is_seekable;
procs.is_readable = (void *)file_obj_is_readable;
procs.is_writable = (void *)file_obj_is_writeable;
procs.funcs_ptr = (void *)&procs;

file_obj_t fobj;
init_file_obj(&fobj, input_file);

dbg_printf("file obj = %p\n", &fobj);

ret = JP2KImageInitDecoderEx_func(image, &fobj, (vtable_t *)&procs, dec_opt, mem_obj);
dbg_printf("JP2KImageInitDecoderEx = %d\n", ret);
if (ret != NULL) {
    dbg_printf("failed to decode\n");
}
```

We made sure we checked the return value from JP2KImageInitDecoderEx and bailed in case of error. In our case, JP2KImageInitDecoderEx returns 0 on success. It took us a few tries to implement the stream functions correctly, but we finally got our desired return value.

The next function, JP2KImageDataCreate, takes no arguments and its return value is passed to the following function JP2KImageGetMaxRes. We called them both and moved on.

We got to the JP2KImageDecodeTileInterleaved function which takes **7!** Arguments, of which 3 are return values from JP2KImageCreate, JP2KImageGetMaxRes, and JP2KImageDataCreate.

The 2nd and 6th parameters were found to be null after xrefing and looking inside AcroRd32 in IDA.

We remained with the 4th and 5th arguments. We concluded that they depend on the color depth (8/16), so we decided to fuzz with constant depth.

Finally we got:

```
image_data = JP2KImageDataCreate_func();

mex_res = JP2KImageGetMaxRes_func(image);

ret = JP2KImageDecodeTileInterleaved_func(image, 0/*null in ida*/, mex_res, 8, 0xff, 0/*allways null*/, image_data);
dbg_printf("JP2KImageDecodeTileInterleaved_func = %d\n", ret);
```

At last, we called the functions JP2KImageDataDestroy, JP2KImageDestroy, and JP2KDecOptDestroy to release the objects we created and avoid memory leaks. This is critical for WinAFL when *fuzz_iterations* are high.

Done! We have a working harness!

In one final tweak, we separated the initialization code – loading JP2KLib and finding the functions from the parsing code. This improves performance, as we don't have to pay for initialization in every fuzz iteration. We called the new function "fuzzme". We will also export "fuzzme" (you can export functions in an exe file) as it's easier than finding the relevant offset in the binary.

Anecdote: When testing our harness in WinAFL, we found out that WinAFL generates files with duplicate magic. After we dug a bit, we found that Adobe used different SEEK constants than the ones defined in libc, causing us to mix SEEK_SET and SEEK_CUR.

Fuzzing Methodology

1. Basic tests for the harness
    1. Stability
    2. Paths
    3. Timeouts
2. Fuzzing Setup
3. Initial corpus
4. Initial line coverage

5. Fuzzing loop
    1. Fuzz
    2. Check coverage / crashes
    3. cmin & repeat
6. Triage

Basic Tests for the Harness

Before starting a big fuzzing session, we do a few sanity tests to make sure we're not just heating servers. The first thing we check is that the fuzzer is reaching new paths with our harness, meaning that the *total path* count is steadily rising.

If the path count is zero or almost zero, there are a few pitfalls we can check for:

- The target function was inlined by the compiler which causes WinAFL to miss the entry to the target function and results in WinAFL termination with program abort.
- This could also happen if the number of arguments (-nargs) is not correct or that the calling convention is not the default.
- Timeouts – Sometimes the timeout is too low and causes the fuzzer to kill the harness too quickly. The solution is to raise the timeout.

We let the fuzzer run for a few minutes and then checked the stability of the fuzzer. If the stability is low (under 80%), we try to debug the issue. The stability of the harness is important, as it affects the accuracy and performance of the fuzzer.

Common pitfalls:

- Check for random elements. For example, some hash table implementations use random to prevent collision, but this is really bad for coverage accuracy. We just patch the random seed to a constant value.
- Sometimes the software has a cache for certain global objects. We usually just do a nop run before calling the *target function* to reduce this effect
- For 32-bit targets on a Windows 10 64-bit machine, the stack alignment is not always ~8 bytes. This means that sometimes *memcpy* and other AVX optimized code will act differently and that does affect coverage. One solution is to add code in the harness to align the stack.

If all the above fail, we use DynamoRIO to do instruction tracing for the harness and diff the output.

Fuzzing Setup

Our setup consists of a VM with 8-16 cores and 32 GB of RAM, running Windows 10 x64.

We fuzz on a RAM disk drive using ImDisk toolkit. We discovered that with fast targets, writing test cases to disk is a performance bottleneck.

We disable Windows Defender because it hurts performance and because some of the test cases generated by WinAFL were discovered by Windows Defender as a known exploit (“Exploit:Win32/CVE-2010-2889”).

We disable the Windows Indexing Service for performance.

We disable Windows Update because it interferes with the fuzzing efforts (restarts the machine and replaces fuzzed DLLs).

We enable page heap for the harness process because it has proven to find bugs that we wouldn’t detect otherwise.

We use the edge as the coverage type instead of the default basic block as it proved better in finding bugs, despite the fact that it’s slower than basic block mode.

This is an example command for running our adobe_jp2k harness:

afl-fuzz.exe -i R:\jp2k\in -o R:\jp2k\out -t 20000+ -D c:\DynamoRIO-Windows-7.0.0-RC1\bin32 -S Slav02 — -fuzz_iterations 10000 -coverage_module JP2KLib.dll -target_module adobe_jp2k.exe -target_method fuzzme -nargs 1 -covtype edge — adobe_jp2k.exe @@

Initial Corpus

Once we have a working harness, we create an initial corpus for it, usually from:

- Online corpuses (afl corpus, openjpeg-data)
- Test suites from open source projects
- Crawling google / duckduckgo
- Corpuses from our older fuzzing projects

Corpus Minimization

Using a big corpus of files that produce the same coverage hurts the performance of the fuzzer. AFL handles this by minimizing the corpus using afl-cmin. WinAFL has a port for the tool called winafl-cmin.py.

We take all the files that we gathered and run them through winafl-cmin.py, which results in a minimal corpus.

We run winafl-cmin at least two times to see if we get the same set of files. If we got two different sets, it usually means that there's non-determinism in our harness. This is something we try to investigate, using afl-showmap or other tools.

Once we finished minimizing successfully, we save the set of files as our initial corpus.

Initial Line Coverage

Now that we have a minimal corpus, we want to take a look at our line coverage. Line coverage means which assembly instructions we actually executed. To get line coverage, we use DynamoRIO: *"[dynamoriodir]\bin32\drrun.exe -t drcov — harness.exe testcase"* for each test case. Next, we load the results to IDA using Lighthouse:

Coverage %	Function Name	Address	Blocks Hit	Instructions Hit	Function Size	Complexity
91.81%	sub_100457EB	0x100457EB	287 / 298	1759 / 1916	5529	206
92.59%	sub_1006775A	0x1006775A	7 / 7	25 / 27	77	4
93.33%	sub_10045736	0x10045736	4 / 4	28 / 30	75	2
94.19%	sub_100421EF	0x100421EF	9 / 9	81 / 86	231	4
94.44%	__callnewh	0x10086C50	4 / 4	17 / 18	38	2
94.87%	sub_1003F6EE	0x1003F6EE	19 / 20	74 / 78	198	11
95.24%	__unwind_handler4	0x10077700	3 / 3	20 / 21	70	2
95.37%	sub_10067332	0x10067332	16 / 16	103 / 108	279	10
96.23%	sub_10067449	0x10067449	8 / 9	51 / 53	126	5
96.43%	__FindAndUnlinkFra…	0x1006E35B	9 / 9	27 / 28	80	3
97.73%	_strncmp	0x10070000	11 / 11	43 / 44	116	9
98.31%	___addlocaleref	0x1007A681	17 / 17	58 / 59	149	10
100.00%	___crtSetThreadSta…	0x1007A2AE	3 / 3	12 / 12	27	1
100.00%	sub_1006101A	0x1006101A	1 / 1	4 / 4	13	1
100.00%	sub_10043D47	0x10043D47	1 / 1	2 / 2	4	1
100.00%	sub_100798B0	0x100798B0	1 / 1	2 / 2	8	1
100.00%	sub_100448B5	0x100448B5	6 / 6	19 / 19	39	4
100.00%	JP2KCodeStm::IsSee…	0x100290D4	1 / 1	5 / 5	11	1
100.00%	unknown_libname_17	0x1007C8F9	1 / 1	8 / 8	15	1
100.00%	nullsub_6	0x100640FC	1 / 1	1 / 1	3	1
100.00%	unknown_libname_18	0x1007C908	1 / 1	8 / 8	15	1
100.00%	sub_10067123	0x10067123	3 / 3	13 / 13	28	1
100.00%	sub_10076127	0x10076127	1 / 1	2 / 2	6	1
100.00%	sub_10040AE4	0x10040AE4	18 / 18	91 / 91	224	11
100.00%	JP2KCodeStm::read(…	0x10029161	3 / 3	20 / 20	44	2
100.00%	__lock_file2	0x1007616C	3 / 3	19 / 19	48	1
100.00%	sub_1000B97D	0x1000B97D	6 / 6	25 / 25	62	4
100.00%	sub_1005C185	0x1005C185	1 / 1	6 / 6	13	1

Composer | A - 7.44% - BATCH_A | Hide 0% Coverage:

We note the initial line coverage, as it helps us to evaluate how effective the fuzzing session was.

Fuzzing Cycle

The next step is pretty straight forward:

1. Run the fuzzers.
2. Check coverage and crashes.
3. Investigate coverage, cmin and repeat.

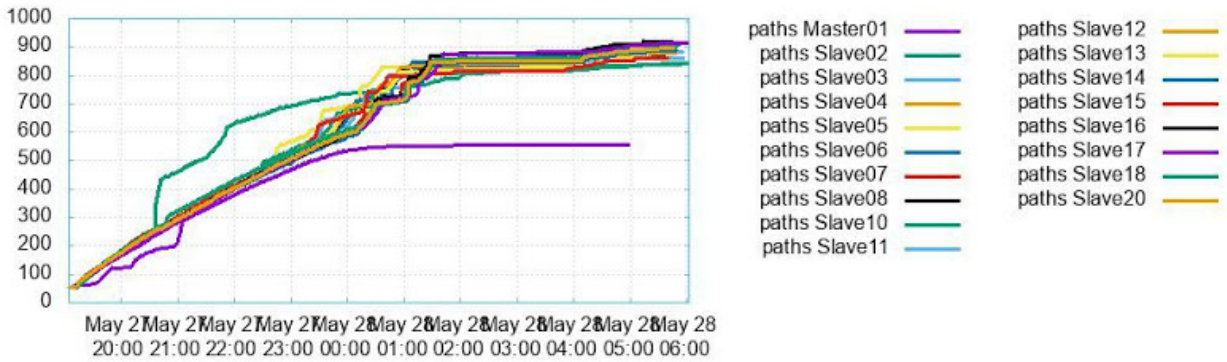
Running the fuzzers, does not require anything special. Just run the fuzzers in the configuration we listed above.

We have a bot with the following features:

1. Status of all fuzzers (using winafl-whatsapp.py).
2. Graph of paths over time for each fuzzer (using winafl-plot.py).
3. Crash triage and generate report (we will talk about this in the next section).
4. Restart dead fuzzers.

We can't stress enough how important it is to automate those tasks. Otherwise, fuzzing is tedious and error-prone.

We check the status of the fuzzers every couple of hours and the paths over time. If we see that the graph plateaus, we try to investigate the coverage.



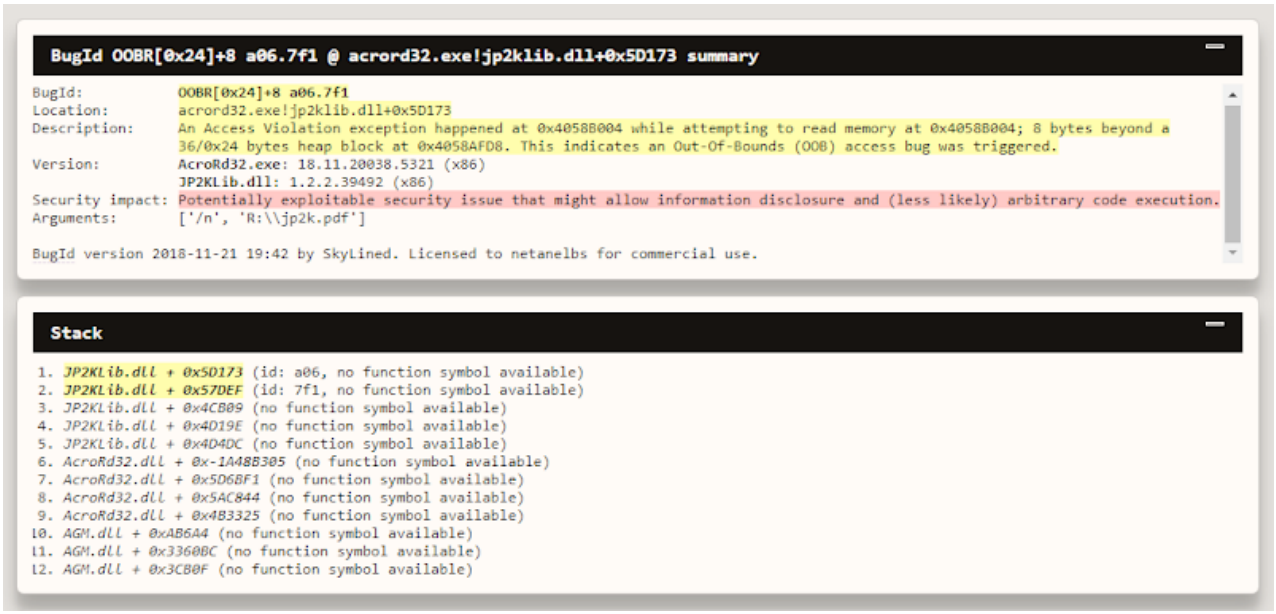
We copy all the queues of all the fuzzers and run them through cmin and look at the results in IDA. We look for functions that are relatively large and have very small coverage. We try to understand what functionality is related to this function and proactively find samples that will trigger this functionality. In JP2K, this wasn't very helpful but in other targets, especially text formats, this phase is a big win!

This stage is very important; in one case, we added a single sample and after a few hours of fuzzing it contributed 1.5% line coverage and we discovered 3 new security bugs.

We then repeated the cycle until we ran out of time or we didn't see any coverage improvement. This usually means we must either change the target or try to improve the harness.

Triage

Once we have a set of test cases that causes a crash, we manually examined the crashes and each crashing input. We quickly changed strategy, as we had a lot of duplicates. We started using BugId to automatically find duplicates and minimize the set to only unique crashes. We used our bot for that.



What We Found

This strategy eventually allowed us to find 53 critical bugs in Adobe Reader and Adobe Pro.

We repeated this process for different parsers such as images, stream decoders and xslt module, which resulted in the following list of CVEs:

CVE-2018-4985, CVE-2018-5063, CVE-2018-5064, CVE-2018-5065, CVE-2018-5068, CVE-2018-5069, CVE-2018-5070, CVE-2018-12754, CVE-2018-12755, CVE-2018-12764, CVE-2018-12765, CVE-2018-12766, CVE-2018-12767, CVE-2018-12768, CVE-2018-12848, CVE-2018-12849, CVE-2018-12850, CVE-2018-12840, CVE-2018-15956, CVE-2018-15955, CVE-2018-15954,CVE-2018-15953, CVE-2018-15952, CVE-2018-15938, CVE-2018-15937, CVE-2018-15936, CVE-2018-15935, CVE-2018-15934, CVE-2018-15933, CVE-2018-15932 , CVE-2018-15931, CVE-2018-15930 , CVE-2018-15929, CVE-2018-15928, CVE-2018-15927, CVE-2018-12875, CVE-2018-12874 , CVE-2018-12873, CVE-2018-12872,CVE-2018-12871, CVE-2018-12870, CVE-2018-12869, CVE-2018-12867 , CVE-2018-12866, CVE-2018-12865 , CVE-2018-12864 , CVE-2018-12863, CVE-2018-12862, CVE-2018-12861, CVE-2018-12860, CVE-2018-12859, CVE-2018-12857, CVE-2018-12839, CVE-2018-8464

One of the bugs we found in jp2k was actually reported to Adobe just a short while before we found it, as it seems it was already discovered being exploited in the wild.

Of course, Adobe Reader is sandboxed, and Reader Protected Mode greatly increases the complexity of turning an exploitable crash inside the sandbox into system compromise, which usually requires another PE exploit, as was used in the mentioned in-the-wild exploit.

We love WinAFL and hope to see it used more.

While using WinAFL, we encountered a number of bugs / missing features. We added support for those new features and upstreamed the patches. These include adding support for App verifier in Windows 10, CPU affinity for workers, fixed a few bugs and added some GUI features.

You can view the commits here:

Netanel's commits – https://github.com/googleprojectzero/winafl/commits?author=netanel01

Yoava's commits – https://github.com/googleprojectzero/winafl/commits?author=yoava333