

Fuzzing Adobe Reader for exploitable vulns (fun != profit)



kciredor.com/fuzzing-adobe-reader-for-exploitable-vulns-fun-not-profit.html

April 25,
2018

Binaries vs websites

It has been half a year since my last blog post covering an IDOR in a website API. About time to write about something new and hopefully interesting! Having switched my focus from websites to binaries a new world opened up to me.

The reason I switched is my passion for low-level engineering. Reading through disassemblies, walking along with code being executed in a debugger, memory corruption, etc. Reverse engineering has always been a passion of mine and binary exploitation seems to get pretty close. I got completely hooked during @corelanc0d3r's Exploit Development Bootcamp, after which I treated myself to the Advanced class as well.

Adobe Reader

Step one is finding your first target. The reason I chose Adobe Reader is primarily that it's a well-known application, offering reasonable bounties for example through submission to the ZDI. Also, my assumption was that it would be easier to find bugs in a PDF reader than in a browser like Chrome. I would say, to my knowledge, that Adobe Reader, Office and the well-known internet browsers are the top 5 well known and hardest application targets to find exploitable vulnerabilities in. Which makes them interesting.

Before exploitation comes fuzzing

Wow did I underestimate this one! I told myself it would take quite some time to build a reliable exploit once I found a bug in Adobe Reader. There are so many mitigations to work through once you have an exploitable crash. Amongst others: Data Execution Protection (DEP: prevents your code from being executed), Address Space Layout Randomization (ASLR: where in memory is my code anyway?), Sandboxing (you need to escape this one, it limits what your code can do). It's hard to end up with reliable code execution.

But before you can start building an exploit you need to trigger a bug or multiple bugs. Perhaps one to leak a DLL address to bypass ASLR and another one which overwrites an exception handler address and triggers a crash. So how do you find these bugs? The answer is fuzzing. And in case of a hard target like Adobe Reader, this can take forever.

Well, for me it did. Or at least months.

Diving into fuzzing you'll find out it's a world on its own. There are many concepts and tools. Endless possibilities to try and force crashes on targets. The basic idea is generating as many inputs (e.g. input files) as you can and running them as fast as you can against your target. If you are lucky, one or more inputs will crash the target.

With harder targets besides luck, you'll need better ideas than throwing around random inputs.

Many ways to Rome

Fuzzing a complex target like Adobe Reader requires you to get to know the target really well. Open up some of the executables / DLL's in a disassembler and see what you can make up of the symbols (if any) and references. Perhaps you'll find out Adobe Reader uses libtiff, which learns us that might be a way in. Give this recon process plenty of time.

Now let's see how we can attack our target.

Generating corpus on open source PDF readers

Researcher @yarp604 explained to me it's hard to get coverage on a target like Adobe Reader, mostly because it's slow and hard to orchestrate. His suggestion was to create a large enough PDF corpus on open source PDF readers and throw that corpus at Adobe Reader. And so I did.

Amongst the most popular fuzzers is American Fuzzy Lop (afl-fuzz) by @lcamtuf. Instead of just continuously throwing random input files at a target, afl actually learns what the input file format looks like by covering the code paths (edges to be exact) of the target application. Also, it leverages Linux forking to run hundreds or even thousands of executions of your target per second per processor core.

In order to fuzz open source readers on Linux you should get yourself a couple of vm's with a distro you like. Install afl on it and download the open source reader tarball. Now unpack it into /dev/shm, which is shared memory. We try to compile everything statically and run everything from memory, in order to gain more executions per second. Also, you'll need to compile the open source reader using one of afl's compilers. Furthermore we should patch out any output file writes for more speed.

I have set up fuzzing for 3 different targets in parallel: xpdf, mupdf and ghostscript. Let me share with you an example on xpdf.

- Unpack xpdf v4.x sourcecode into /dev/shm so we are working in RAM.
- Your target needs to accept an input file and should fully parse the input PDF. I picked pdftoppm.c.
- Edit pdftoppm.c and patch out the output file write. I simply placed a continue inside the for loop right before the writes starting with the line: if (!strcmp(ppmRoot, "-")).

- Compile xpdf using afl's gcc or clang, for me clang ended up running xpdf faster. Harden it and use Address Sanitizer to find more interesting inputs:

```
mkdir build && cd build && CC=afl-clang-fast CXX=afl-clang-fast++ cmake -
DCMAKE_BUILD_TYPE=Release .. && USE_ASAN=1 AFL_HARDEN=1 make
```

- Place your input files in /dev/shm/in-raw. You should start with a nice input corpus, for instance, get some files from <https://github.com/mozilla/pdf.js/tree/master/test/pdfs>
- You need to isolate unique inputs, no reason to fuzz inputs that offer the same code paths. We do this by asking afl to minimize the input corpus for us:

```
afl-cmin -i in-raw -o in -- xpdf-4.00/build/xpdf/pdftoppm -mono -r 1
@@ -
```

Now you could also minimize each file using afl-tmin if you feel like it. If you don't then because of the way afl works at least you should probably do something like:

```
find in/ -type f -size +10k -
delete
```

Now run your fuzzer from within screen or tmux and run as many fuzzers as you have cores available! You can verify you're on the right track using afl-gotcpu.

```
afl-fuzz -i in -o out -x /usr/share/afl/dictionaries/pdf.dict -M master -- xpdf-4.00/build/xpdf/pdftoppm -
mono -r 1 @@ -
```

- Notice the command line I'm using with pdftoppm: -mono -r 1 really speeds things up. Furthermore I use the afl provided dictionary which helps afl learn the PDF format much faster. The @@ will be replaced by afl with the generated inputs. Replace -M master with -S slave1, -S slave2 etc for each core.
- Don't forget to setup a cronjob that backups your corpus from /dev/shm to persistent storage. A simple crash of your vm will otherwise make you start over!
- Note: you may be able to leverage afl's persistent mode which will make the fuzzing even faster by looping the actual rendering functions. Look into it, the docs are very good.

On a 6 core Xeon vm I was able to get about 1200 executions per second on xpdf, 4000 on mupdf and 20 on ghostscript (lol). You can get a nice summary view using afl-whatsup.

Throwing the open source PDF corpus at Adobe Reader

Running these 3 fuzzers for about 6 weeks got me a corpus of 500k pdf files. Now there's a couple of things we need to take care of before we can do a proper test run. First of all, you'll have 3 servers with /dev/shm/out/ populated with subdirectories queue, crashes, hangs (each per master/slave). Copy the out directories over to your local machine or vm and use some Unix foo to join them all together into a single directory with unique filenames. Something like this should do the trick:

```
find . -type f -name 'id*' -size +0 | gawk 'BEGIN{ a=1 }{ printf "cp \"%s\" ../%08d.pdf\n", $0, a++ }' | bash
```

Now create a Windows batch file that will loop through all your files and launch Adobe Reader, for example:

```
for %%f in (y:\*.pdf) do  
AcroRd32 %%f
```

You can run this from your Windows vm. You'll also need to code a popup killer script using AutoIt which will close popups like "This PDF is corrupted". And since you cannot add a timeout to this setup, you'll have to manually close the Reader window every single time... Right.

How can we improve on this? Simply use BugId by @berendjanwever. It will output html reports for you and you can add timeouts. The batch file would now look like this and you're all set:

```
for %%f in (y:\*.pdf) do bugid acrobatdc --nApplicationMaxRunTime=15 --  
"%%f"
```

Notice that BugId will suggest enabling full pageheap for your target, and you should when testing/fuzzing targets. It will offer to set it for you. You can also use gflags for this purpose to set it yourself. Having pageheap enabled will make you find more bugs. Don't forget to disable it again when building your exploits.

Again I suggest you deploy the whole thing on a cloud vm so you can close your laptop screen while it's running in the background. And while you're at it why not run the exact same setup for FoxitReader as well in parallel?

For me, this 500k corpus did not do much with Adobe Reader (as in no crashes at all, a couple of hangs). It did crash FoxitReader though, but those crashes were commonly agreed upon not to be exploitable. Bugs are not necessarily vulnerabilities.

(Win)AFL fuzzing Adobe Reader itself

The open source PDF corpus did not contain any inputs that triggered a vuln in Adobe

Reader or FoxitReader.

Next!

Wouldn't it be cool to use afl on Adobe Reader itself? It might be slow, but a crash would then be a true crash on our final target. Adobe Reader runs on Windows and macOS. I'm a Linux user so I started out with a local Windows 8 VM with Adobe Reader DC installed. No need to install Windows 10 I figured because that might just over complicate things with even more mitigations up front. You'll know by now that afl does not support Windows out of the box, but @ifsecure actually ported it by leveraging DynamoRIO and released WinAFL.

WinAFL will require you to specify a function name (given the target has symbols) or function offset, which you should find by reverse engineering your target application. The function should open your input, process it, close your input. WinAFL will run your target application and loop this function, again and again, each time restoring the state of memory as if it were the first run, replacing the input file meanwhile. This is a clever way to reach a high number of executions per second.

So I tried WinAFL on an easy target first: unrar.exe, a small CLI application. This seems to work great. Load unrar.exe in a disassembler and find your offset and number of args. Populate a small input corpus and run WinAFL. For example:

```
c:\research\win afl\bin64\ afl-fuzz.exe -t 1000+ -i c:\research\unrar\in -o c:\research\unrar\out -D
C:\research\dr\bin64 -- -fuzz_ iterations 5000 -target_ module unrar.exe -nargs 3 -target_ offset 0xE864 -
coverage_ module unrar.exe -- unrar.exe p @@
```

For the more complex apps like Adobe Reader and FoxitReader, I wasn't able to get things going this easily.

Attempt 1: Harness AcroRd32.dll

When you disassemble Adobe Reader (I focussed on AcroRd32.dll) you'll find some references to PDDoc. This made me think: how about coding a harness which does a LoadLibrary on AcroRd32.dll and using GetProcAddress call the methods that open and process a PDF document.

That didn't work out: the number of args did not match the number of args I could find in a couple of online Adobe SDK resources. Furthermore, I had too much trouble figuring out all the crazy data structures you need to put in place.

Trial and error kept resulting in error. My harness.cpp is now in the trash.

Attempt 2: Harness JP2KLib.dll

This dll is responsible for rendering JPEG2000 files, so what if we can focus on just that part of the PDF processing? Well, again, I hit the wall in trying to figure out the way JP2KLib.dll works. Making use of API call spying software did not help either (no symbols, just huge structures).

Next!

Attempt 3: Find an offset for WinAFL to target the real thing

I put in quite some time and effort to find a fuzzable function in AcroRd32.exe and .dll.

But even if you could find one that meets all WinAFL's requirements of such a method, by the time you try to fuzz it with WinAFL you'll find out it cannot cover code running in child processes. And Adobe Reader spawns a child process that does the heavy lifting.

You can confirm this yourself by loading Reader in WinDbg and on the initial break apply .childdbg 1. On subsequent breaks, you can check out loaded modules, switch between the processes, etc.

Not sure even if child coverage would be possible you'll be able to provide WinAFL with a proper target function. It needs to open your input, render it, and close it immediately to free resources so WinAFL can put a new input in place and repeat. I don't think such a function is built into Adobe Reader :-). Perhaps... coding a custom plugin? Then have the fuzzer cover AcroRd32.dll but target your plugin function. I did not investigate this path further.

Harnessing the PDF Libraries

Browsing Adobe's and Foxit's websites you'll find out there are actually SDK's you can download and use. This gave me the idea you can write your own harness that simply renders a given input PDF in memory using the engine exposed by the SDK. Odds are this is the exact same engine as the actual Reader uses to do the rendering.

First of all you need to get a copy of the Adobe and/or Foxit SDK's. Note these SDK's are available as evaluation packages.

Here's the C++ code for the Adobe PDFL harness I made and used to fuzz the rendering engine:

```
// *****  
//  
// Harness.cpp leverages the Adobe PDFL libs to render (in-mem) every page of a given pdf.  
// Features a public fuzz method which can be used as target method in WinAFL  
//  
// Based on the open source RenderPage DataLogic PDFL example from github:  
// https://github.com/datalogics/adobe-pdf-library-samples/tree/master/CPlusPlus/Sample_Source/  
//
```

```

// Confirm render timing: powershell -Command "Measure-Command {./harness.exe test.pdf | Out-
Default}"
//
// (c) kciredor 2018
//
// *****

#include "APDFLDoc.h"
#include "InitializeLibrary.h"

#include "RenderPage.h"

#define RESOLUTION 110.0
#define COLORSPACE "DeviceRGB"
#define FILTER "DCTDecode"
#define BPC 8

extern "C" __declspec(dllexport) void fuzz(char* fn);

APDFLib libInit;

void fuzz(char* fn) {
    std::cout << "Rendering " << fn << " - Res " << RESOLUTION << ", Colorspace " << COLORSPACE
<< ", Filter " << FILTER << ", BPC " << BPC << std::endl;

    DURING
        APDFLDoc pdfDoc(fn, true);

        for (int i = 0; i < pdfDoc.numPages(); i++) {
            PDPage pdPage = pdfDoc.getPage(i);
            RenderPage drawPage(pdPage, COLORSPACE, FILTER, BPC, RESOLUTION);

            PDPageRelease(pdPage);
        }

        PDDocRelease(pdfDoc.getPDDoc());
    HANDLER
        std::cout << "Caught PDFL error" << std::endl;
    END_HANDLER
}

int main(int argc, char** argv) {
    if (! libInit.isValid()) {
        std::cout << "PDFL init failed with code " << libInit.getInitError() << std::endl;

        return false;
    }

    if (argc != 2) {
        std::cout << "Requires input.pdf parameter" << std::endl;

        return false;
    }
}

```

```
fuzz(argv[1]);  
  
return true;  
}
```

Now compile it, put it next to the PDFL dll's, populate a small seed corpus (try 'small.pdf') and start fuzzing:

```
c:\research\winaf\bin32\afl-fuzz.exe -t 10000+ -x c:\research\winaf\testcases\extras\pdf.dict -i  
c:\research\reader\in -o c:\research\reader\out -D C:\research\dr\bin32 ^  
-- -fuzz_iterations 5000 -target_module harness.exe -nargs 1 -target_method fuzz -covtype edge ^  
-coverage_module DL150PDFL.dll ^  
-- harness.exe @@
```

Put this setup on a cloud vm and let it run for a while. Triage the crashes it finds daily against the real Acrobat Reader. Unfortunately for me this path did not work out the way I expected: PDFL crashes != actual Reader crashes.

The exact same thing you can do for Foxit's library. Actually got some real FoxitReader crashes here, but not exploitable again. Coding a harness for this one is just as easy and I'll leave it up to you to have some fun with it.

Better understanding the PDF file format: libtiff

Like I mentioned earlier, you might find out Adobe Reader uses libtiff when you disassemble the main executable or main dll. Same thing with libpng. So another idea would be to fuzz libtiff and wrap the crashing tiff's into a PDF. You can fuzz libtiff using afl on Linux and get some nice speeds (12000/sec on 6 Xeon cores), this time I applied afl's persistent mode as well.

If you want to wrap a tiff inside a PDF without changing the tiff a single bit, you can put it inside an XFA as an ImageField with base64 embedded image content. XFA is one of two ways to embed a form into a PDF. You want to automate this process, but you can start with a clean PDF shaped like this by using the Adobe AEM Forms Designer. Apparently it's now considered old and released for free (serial# on adobe.com), but the download link is quite hard to find.

So after fuzzing libtiff for a couple of days and automatically wrapping all crashes into XFA's into PDF's and triaging the resulting PDF's I did not get any worthwhile crashes. On the other hand, some of the PDF's did manage to crash Adobe Reader 9.3, exploitable even! But that's old stuff, I'm targeting the latest DC version. It appears Adobe hardens libtiff quite a bit, it's not simply a copy of the latest open source version that they use.

Coding my own custom fuzzer

Almost out of ideas. I remember Corelan hinting you have to write your own fuzzers or

everyone will find the same bugs. The art of fuzzing (PDF link at the end of this write-up) suggests the same.

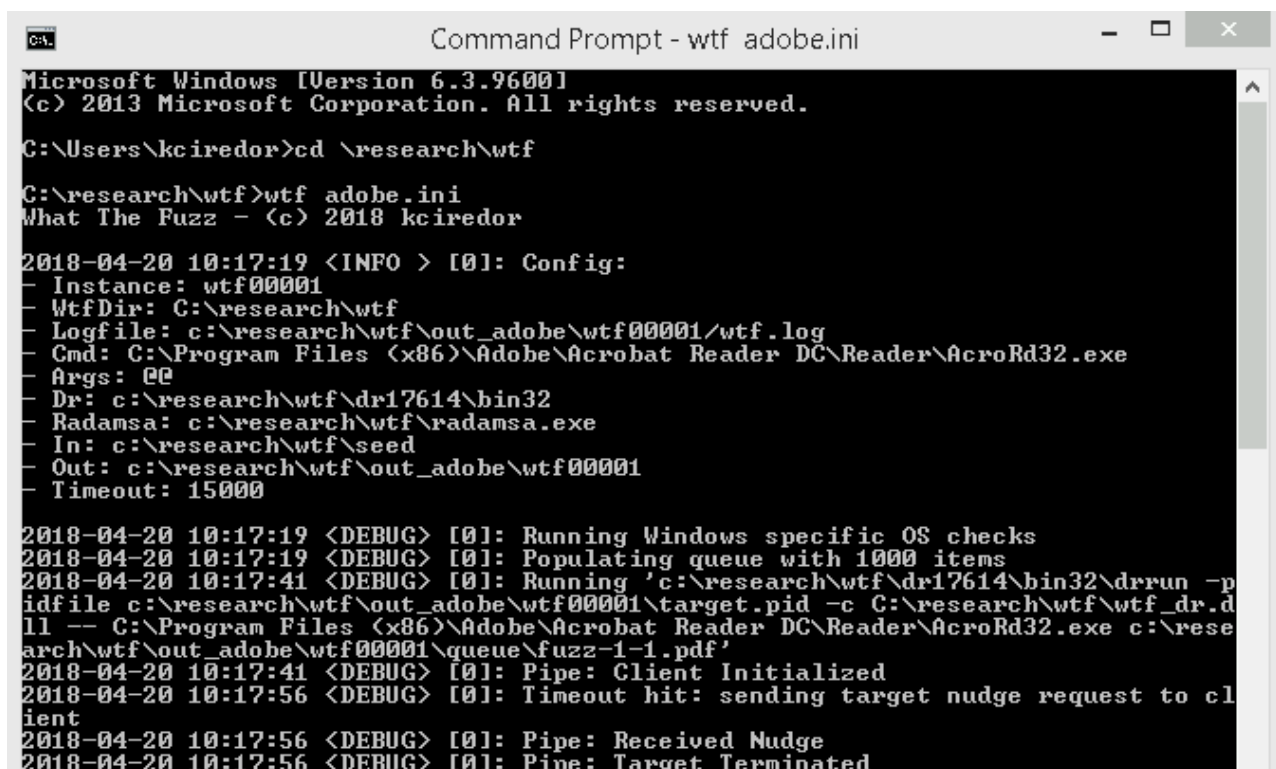
Oh well. Let's do it. :-)

How do you start coding your own fuzzer? I would say there's a certain amount of basic requirements:

- Run a target application
- Feed the target your inputs
- Handle timeouts
- Catch crashes
- Fuzz inputs

I started by reading through WinAFL and afl sourcecode to get an understanding of how this works. Soon enough I decided on writing both the fuzzer and a DynamoRIO client in C and use the client for getting code coverage. DR also allows you to catch exceptions (which could be crashes). Halfway there I figured out I like the speed of coding in Go better so now my fuzzing and orchestration logic is written in Go and the DynamoRIO part (the client) is written in C.

My fuzzer is called What The Fuzz and is able to create an input queue based on a given seed and handle all the basic requirements I mentioned before. It runs on both Linux and Windows and finds crashes on xpdf in seconds. WTF even found crashes on FoxitReader within an hour or so, but unfortunately not exploitable so far.



```
Microsoft Windows [Version 6.3.9600]
(c) 2013 Microsoft Corporation. All rights reserved.

C:\Users\kciredor>cd \research\wtf

C:\research\wtf>wtf adobe.ini
What The Fuzz - (c) 2018 kciredor

2018-04-20 10:17:19 <INFO> [0]: Config:
- Instance: wtf00001
- WtfDir: C:\research\wtf
- Logfile: c:\research\wtf\out_adobe\wtf00001\wtf.log
- Cmd: C:\Program Files (x86)\Adobe\Acrobat Reader DC\Reader\AcroRd32.exe
- Args: @@
- Dr: c:\research\wtf\dr17614\bin32
- Radamsa: c:\research\wtf\radamsa.exe
- In: c:\research\wtf\seed
- Out: c:\research\wtf\out_adobe\wtf00001
- Timeout: 15000

2018-04-20 10:17:19 <DEBUG> [0]: Running Windows specific OS checks
2018-04-20 10:17:19 <DEBUG> [0]: Populating queue with 1000 items
2018-04-20 10:17:41 <DEBUG> [0]: Running 'c:\research\wtf\dr17614\bin32\dr-run -p
idfile c:\research\wtf\out_adobe\wtf00001\target.pid -c C:\research\wtf\wtf_dr.d
ll -- C:\Program Files (x86)\Adobe\Acrobat Reader DC\Reader\AcroRd32.exe c:\rese
arch\wtf\out_adobe\wtf00001\queue\fuzz-1-1.pdf'
2018-04-20 10:17:41 <DEBUG> [0]: Pipe: Client Initialized
2018-04-20 10:17:56 <DEBUG> [0]: Timeout hit: sending target nudge request to cl
ient
2018-04-20 10:17:56 <DEBUG> [0]: Pipe: Received Nudge
2018-04-20 10:17:56 <DEBUG> [0]: Pipe: Target Terminated
```

For now, it's a chaos monkey leveraging Radamsa but I'm planning on actually coding the 'code coverage guided fuzzing' soon. Like @rantyben says in an epic presentation: start fuzzing with what you have, cold cores don't add any value, meanwhile improve functionality.

About vm deployment / automation

Sure you can run fuzzers on your own laptop, but it will ruin your SSD and you can't turn off your laptop. And it does not scale. You need remotely deployed vm's, the more the better.

You could use for instance a Windows 2008 Server image on a cloud provider and install your tooling.

Another possibility is to create your own vm image using KVM + virt-manager and make sure it's compatible with your preferred cloud provider (think about drivers like virtio). Deploy the image and you're all set having your fully customized OS and tooling of choice, ready to fuzz!

Now set up some cronjobs or whatever kind of scheduled job you fancy. Backup your corpus every night. Sometimes cloud vm's crash and you need to prepare for that.

Fun does not always mean profit

When you've continued reading this far you'll have found out that I did not find any bugs in Adobe Reader itself, despite many attempts. Found a ton of bugs in lots of other software though. Every single time I thought: this is a smart idea, this should probably get me some bugs!

Because there's no way to build an exploit if you don't have the vulnerabilities, this is where it ends for now. Need an energy recharge to start looking into PDF's again ;-)

So why did I release this research? Because I did not find that much research that explains in-depth on how (not) to fuzz this kind of targets. Often times you'll get hints and from there you are on your own. Perhaps someone else can take it from here given my experiences so far. It will hopefully save you months of trial and error and inspire you to find more clever ways, or simply do what I did but throw many more VM's at it and cross your fingers! Let me know if you get lucky.

Resources

Definitely, check out the following resources on (PDF) fuzzing if you want to learn more:

What's next

Perhaps I'll continue researching new angles with Adobe Reader. Perhaps I'll switch to vm escapes. Malware analyses sounds fun. Hacking IoT sounds fun. Odds are I'll be fuzzing some more and it might now be the time to follow a @richinseattle fuzzing training to add new insights and keep on learning.

Any suggestions would be very welcome, please hit me up on [Twitter @kciredor](#) or leave a comment below this post.

Let's keep on hacking!

Cheers, kciredor