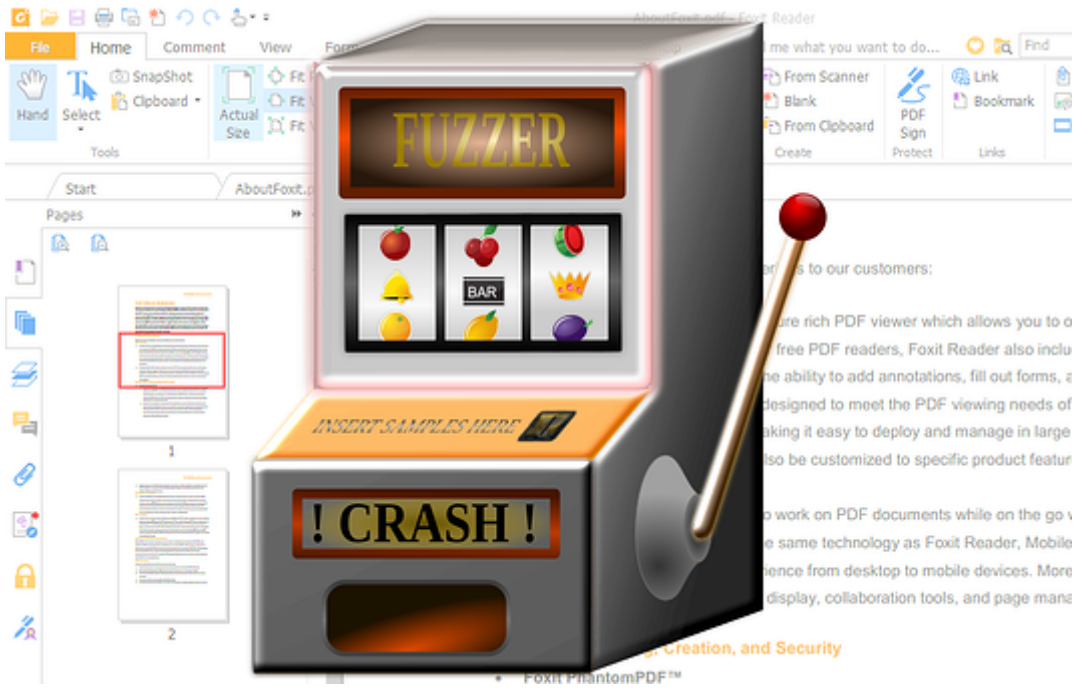
Fuzzing Closed Source PDF Viewers

gosecure.net/blog/2019/07/30/fuzzing-closed-source-pdf-viewers

This blog post covers typical problems which arise when fuzzing closed source PDF viewers and possible approaches to these problems. Hereby it focuses on both: Input-Minimization and Non-Terminating programs.

The approaches were found and implemented as part of my master thesis which I have written at TU Darmstadt, Germany in cooperation with Fraunhofer SIT.



Context

The core idea of fuzzing a PDF viewer is quite simple: take a PDF, corrupt it slightly and check if it crashes the viewer. While this sounds extremely simple, it is quite difficult to do it correctly and efficiently. The PDF file format is one of the most used and thus important formats in these days. Therefore it is not a surprise that PDF viewers have widely been exploited in the past[1] and were still exploited in 2018[2]. Checking out the amount of bugs which are reported to major PDF viewers[3] reveals that there is still a lot of hardening to do and I wanted to contribute to both: the security of PDF viewers and the fuzzing community.

Problems that typically arise when fuzzing PDF viewers are:

- **At which point can the Fuzzer be sure that no crash has occurred?**
A PDF viewer never signals that it is done parsing and rendering the given PDF. At which point can the application be closed?
- **Which PDF(s) should be chosen as templates for mutations?**
The chosen PDF(s) should cover as much of the target's code as possible. How can code coverage be measured efficiently if the source code is not available?

Problem: Non-Terminating programs

The (normal) termination of the Fuzzee signals the Fuzzer that it has finished doing its processing and no crash has occurred. This is important for the Fuzzer because it can now start the next iteration of testing. The problem with PDF viewers is that they obviously never terminate themselves and thus Fuzzers lack a metric to determine when the next test can be started.

What most existing Fuzzers do is that they either make use of a hardcoded timeout after which they kill the application if no crash has occurred, or they constantly poll the amount of CPU cycles of the target and assume that the program can be closed when it drops below a certain threshold. Both: the timeout and the threshold must be determined precisely but are more or less guessed. For the Fuzzer this means that it kills the application either too early (might be missing crashes) or too late (waste of time).

Approach: Make the program terminate!

The idea was to find the last basic block of the viewer which is executed when it is given a valid input. The assumption here is that this basic block is only executed when the viewer is completely done with parsing and rendering the given PDF. As of next, this block must be patched in a way that it terminates the program.

In order to find out which basic blocks of a program have been executed at runtime, researchers make use of a concept called **Program Tracing**. The idea is to make the target generate additional information about its execution (trace), such as memory usage, taken branches or executed basic blocks. Because the target is not creating these information, instructions must be added to it. This process is called "Program Instrumentation". In an open source environment, the target program could have simply been recompiled with additional compiler extensions such as AddressSanitizer (ASAN) which would take care of adding the instrumentation at compile time. Obviously this impossible for closed source PDF viewers.

Luckily the amazing framework "**DynamoRIO**" does not need any source code to apply this instrumentation, as it instruments the program at runtime (Dynamic Binary Instrumentation).

```
drrun.exe -t drcov -dump_text -- Program.exe
```

The created program trace looks something like this:

```
Columns: id, base, end, entry, checksum, timestamp, path
  0, 0x00400000, 0x00411000, 0x00401000, 0x000153f1, 0x6369646e, C:\Users\IEUser\Desktop\sha1sum_orig.exe
  1, 0x61000000, 0x614d0000, 0x6107fa00, 0x003323da, 0x080e560b, C:\Program Files\OpenSSH\bin\cygwin1.dll
  2, 0x674c0000, 0x675bc000, 0x674d72a0, 0x000ff010, 0x4e9b1247, C:\Program Files\OpenSSH\bin\cygiconv-2.dll
  3, 0x6c140000, 0x6c162000, 0x6c154b70, 0x0001ac83, 0x00003d04, C:\Program Files\OpenSSH\bin\cyggcc_s-1.dll
  4, 0x6d170000, 0x6d2d0000, 0x6d218e60, 0x00136967, 0x589416df, C:\Users\IEUser\Desktop\DynamoRIO-Windows-7.0.0-RC1\lib32\release\dynamorio.dll
  5, 0x6f3a0000, 0x6f3ac000, 0x6f3a0000, 0x0000eed4, 0x589416e0, C:\Users\IEUser\Desktop\DynamoRIO-Windows-7.0.0-RC1\ext\lib32\release/drmgr.dll
  6, 0x6f7c0000, 0x6f7d1000, 0x6f7c6190, 0x00012a71, 0x3234322b, C:\Program Files\OpenSSH\bin\cygintl-8.dll
  7, 0x6f850000, 0x6f85b000, 0x6f850000, 0x00008bd0, 0x589416e1, C:\Users\IEUser\Desktop\DynamoRIO-Windows-7.0.0-RC1\ext\lib32\release/drreg.dll
  8, 0x708c0000, 0x708cd000, 0x708c0000, 0x000097f7, 0x589416e2, C:\Users\IEUser\Desktop\DynamoRIO-Windows-7.0.0-RC1\ext\lib32\release/drx.dll
  9, 0x70990000, 0x7099b000, 0x70990000, 0x00006b1e, 0x589416e3, C:\Users\IEUser\Desktop\DynamoRIO-Windows-7.0.0-RC1\ext\lib32\release/drcovlib.dll
 10, 0x70b90000, 0x70b98000, 0x70b90000, 0x0000e4f2, 0x589416e3, C:\Users\IEUser\Desktop\DynamoRIO-Windows-7.0.0-RC1\tools\lib32\release\drcov.dll
 11, 0x75ba0000, 0x75beb000, 0x75ba7d88, 0x0004f7af, 0x5b1aa77b, C:\Windows\system32\KERNELBASE.dll
 12, 0x770a0000, 0x77175000, 0x770ecfb7, 0x000d626c, 0x5b1aa77a, C:\Windows\system32\kernel32.dll
 13, 0x77c60000, 0x77da2000, 0x77c60000, 0x0014ed0c, 0x5b6db285, C:\Windows\SYSTEM32\ntdll.dll
 14, 0x76eb0000, 0x76f5c000, 0x76eba472, 0x000a8f06, 0x4eeaf722, C:\Windows\System32\msvcrt.dll
 15, 0x77360000, 0x77402000, 0x77392213, 0x000af359, 0x5b6db220, C:\Windows\System32\rpcrt4.dll
 16, 0x77000000, 0x77019000, 0x77004975, 0x00021fc1, 0x556362e4, C:\Windows\System32\sechost.dll
 17, 0x77180000, 0x77221000, 0x77194919, 0x000a4dc3, 0x5b6db1d7, C:\Windows\System32\advapi32.dll
 18, 0x76f60000, 0x76ffd000, 0x76f9474c, 0x000a5b31, 0x59946079, C:\Windows\System32\usp10.dll
 19, 0x77df0000, 0x77dfa000, 0x77df136c, 0x00013d10, 0x5b6db215, C:\Windows\System32\lpk.dll
 20, 0x76d90000, 0x76dde000, 0x76d99e49, 0x000550b3, 0x5b71a665, C:\Windows\System32\gdi32.dll
 21, 0x77230000, 0x772f9000, 0x7724d6e1, 0x000ccf24, 0x58249e2b, C:\Windows\System32\user32.dll
 22, 0x76de0000, 0x76ead000, 0x76de168b, 0x000cbe46, 0x59b94a4c, C:\Windows\System32\msctf.dll
 23, 0x77dd0000, 0x77def000, 0x77dd1355, 0x00027a42, 0x4ce7b845, C:\Windows\System32\imm32.dll
BB Table: 18467 bbs
module id, start, size:
module[ 13]: 0x00046be8,  13
module[ 13]: 0x000635d0,  16
module[ 13]: 0x00063529,  18
module[ 13]: 0x0006353b,   4
module[ 13]: 0x000635e0,  11
module[ 13]: 0x000635f1,  12
module[ 13]: 0x000527a4,  69
module[ 13]: 0x000635fd,  20
module[ 13]: 0x00063611,   7
module[ 12]: 0x0004ef9a,  13
module[ 12]: 0x0004efa7,   5
module[  0]: 0x00001000,  21
```

Output from DynamoRIO

As it can be seen, the trace shows which basic blocks of which module have been executed and it retains the order of the basic blocks which makes it fairly easy to determine the last basic block. So to find out the last basic block which is executed by the target PDF viewer, several traces were created by feeding it with different but valid PDFs. It then became obvious that there is usually a common basic block somewhere close to the end of the traces, which is the block which must be instrumented.

Unfortunately, the trace is written to disk only after the program has exited, so that a (high) hardcoded timeout had to be used here.

Now that the last common basic block is found, it needs to be patched so that it terminates the program. This can possibly be implemented by overwriting the basic block with:

```
Xor eax, eax
push eax
Push Address_Of_ExitProcess
ret
```

The issue with this is that it takes 9 bytes to represent these instructions. If the basic block is not 9 bytes in size subsequent instructions would be corrupted. To overcome this issue, a new executable section can be added to the PE file which contains the instruction from above. Thus the basic block can be patched with a jump to the newly added section:

```
push SectionAddress
ret
```

To patch the target, the framework **LIEF**[4] can be used which makes it fairly easy to change a given PE file.

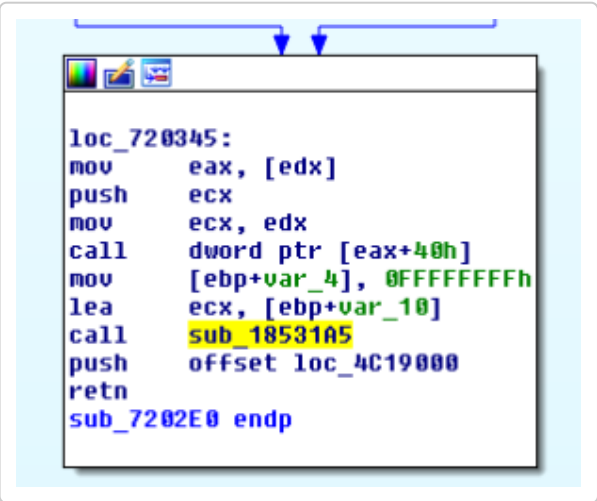
Obviously, it had been much easier to patch the basic block with a breakpoint, which is a one byte instruction. Many existing Fuzzers rely on the fact that a program terminates and thus cannot be used for targeting PDF viewers. The applied exit instrumentation makes their use possible.

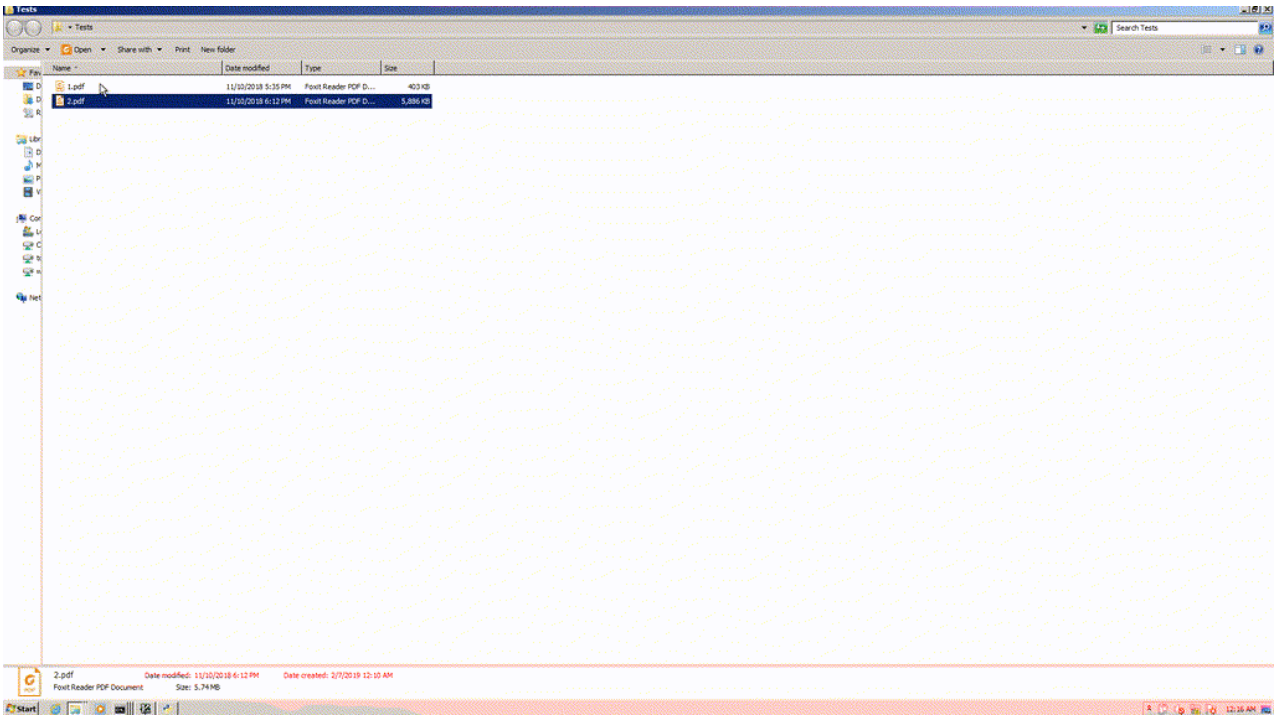
The approach was automated and successfully used on several PDF and image viewers:

1. FoxitReader
2. PDFXChangeViewer
3. XNView

The screenshots below give an idea of how the patch looks like on assembly level. Note that the patched version returns to the newly added section.

Basic block with termination patch





FoxIt Reader behavior once patched with terminating instructions

Input-Minimization

The success of Fuzzing depends heavily on the initial set of inputs (corpus). Thus, it must be assured that the corpus covers as much code of the target as possible, because it obviously increases the chance to find bugs in it. Also, redundancy in the corpus must be avoided, so that each PDF triggers unique behaviour in the target.

A common approach for this is a concept called: **Corpus Distillation**. The core idea of this is to first collect a huge amount of valid inputs. Then for each input the basic block code coverage is measured and if the input triggers only basic blocks which have already been visited by previous inputs, it is removed from the set.

```
corpus = []
inputs = [I1, I2, .... In]
for input in inputs:
   new_blocks = execute(program, input)
   if new_blocks:
      corpus.append(input)
```

Again, program traces need to be created. Since the source code is not available, Dynamic Binary Instrumentation appears to be the only chance to measure basic block code coverage. The problem here is that Dynamic Binary Instrumentation appears to create an unacceptable overhead.

To demonstrate this, FoxitReader was patched with the AutoExit approach and the time until FoxitReader terminated itself was measured.

1. **Vanilla**: 1,5 seconds
2. **DynamoRIO**: 6,4 seconds

Here, Dynamic Binary Instrumentation causes an overhead of almost 5 seconds, which is too high to perform efficient Corpus Distillation.

Approach

Since Dynamic Binary Instrumentation is obviously too slow to perform Corpus Distillation, another approach had to be found to measure basic block code coverage. The idea here consists of two parts:

1. Instrument the binary statically
2. Create a custom debugger which handles the instrumentation

First, each basic block in the target is patched with a breakpoint which is a one byte instruction (**0xcc**). The patch is applied statically in the binary on disk. If any basic block is executed, it triggers a breakpoint event (**int3**) which can be fetched by the supervising debugger. The debugger fetches the **int3** event, and overwrites the breakpoint in both: the binary on disk and in the address-space with its original byte. Finally the Instruction pointer of the target is decremented by one and its execution is resumed.

The screenshot below shows the instrumented basic blocks:

Instrumentation with breakpoints

Thanks to the breakpoints it is easy for
the debugger to identify which basic blocks have been executed.

To evaluate the performance of this approach, all basic blocks of FoxitReader were patched with a breakpoint (1778291 basic blocks).

In the first iteration it took FoxitReader **16 seconds** until it finally terminated. This is **10 seconds slower than DynamoRIO**.
But since the breakpoints are undone in the binary on disk, they will never again trigger an int3 event. Thus, it can be assumed that after the first couple of iterations, most breakpoints will already be reverted and therefore the overhead should be reasonable.

1. **First Iteration:** 16 Seconds (48323 breakpoints)
2. **Second Iteration:** 2 Seconds (2212 breakpoints)
3. **Third and following:** ~1,5 seconds (low number of breakpoints)

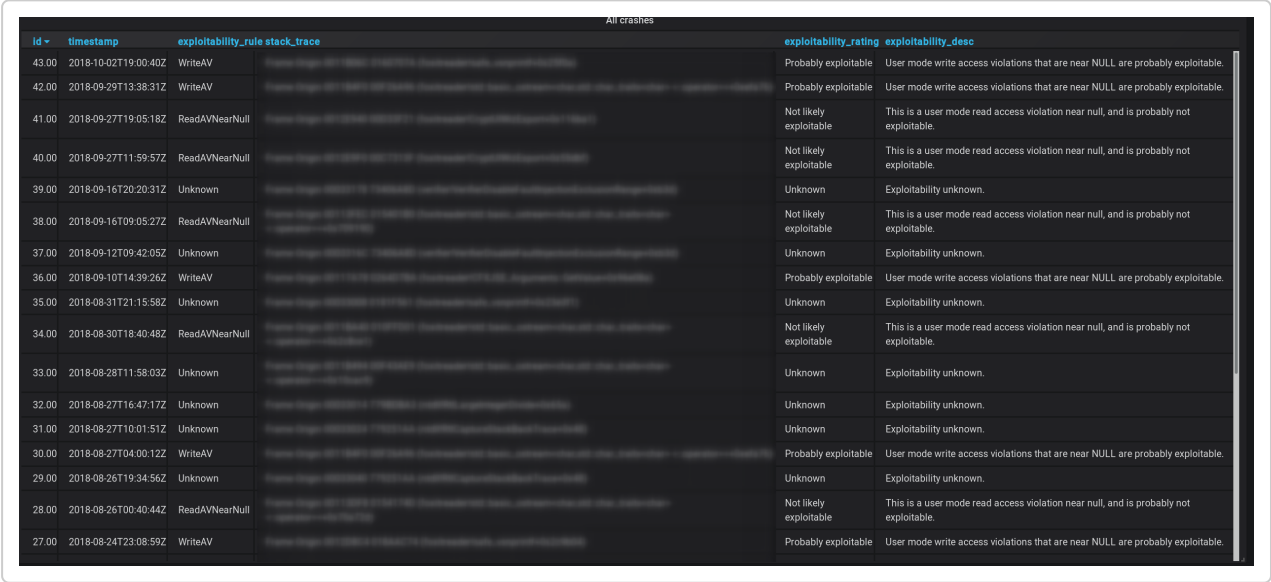
As it can be seen, after the first iterations, the instrumentation caused a minimal overhead, but the debugger is still able to determine any newly visited basic blocks.

This approach was tested on major products and it worked flawlessly on all of these:

1. Adobe Acrobat Reader
2. PowerPoint
3. FoxitReader

Fuzzing

80,000 PDFs were gathered by crawling the internet and the set was minimized to 220 unique PDFs which took around 1,5 days.
With this minimized set, dumb fuzzing was performed. The results were surprisingly good, and all crashes were pushed in a database:



Dashboard displaying fuzzing results

Results

In the end the Fuzzer found 43 unique crashes in a time frame of roughly 2 months, of which three were critical enough to report them to Zero Day Initiative. They were assigned the following ids:

1. ZDI-CAN-7423: Foxit Reader PDF Parsing Out-Of-Bounds Read Remote Code Execution Vulnerability
2. ZDI-CAN-7353: Foxit Reader PDF Parsing Out-Of-Bounds Read Information Disclosure Vulnerability
3. ZDI-CAN-7073: Foxit Reader PDF Parsing Out-Of-Bounds Read Information Disclosure Vulnerability

Sources

1. One public exploit for Adobe Acrobat and Adobe Reader (Bugtraq 42998) https://www.exploit-db.com/exploits/34603
2. Another exploit affecting Adobe Reader (CVE-2018-4990) https://blog.malwarebytes.com/threat-analysis/2018/05/adobe-reader-zero-day-discovered-alongside-windows-vulnerability/
3. Zero Day Initiave latest advisories published https://www.zerodayinitiative.com/advisories/published/
4. LIEF framework used to patch Portable Executable (PE) file: https://github.com/lief-project/LIEF