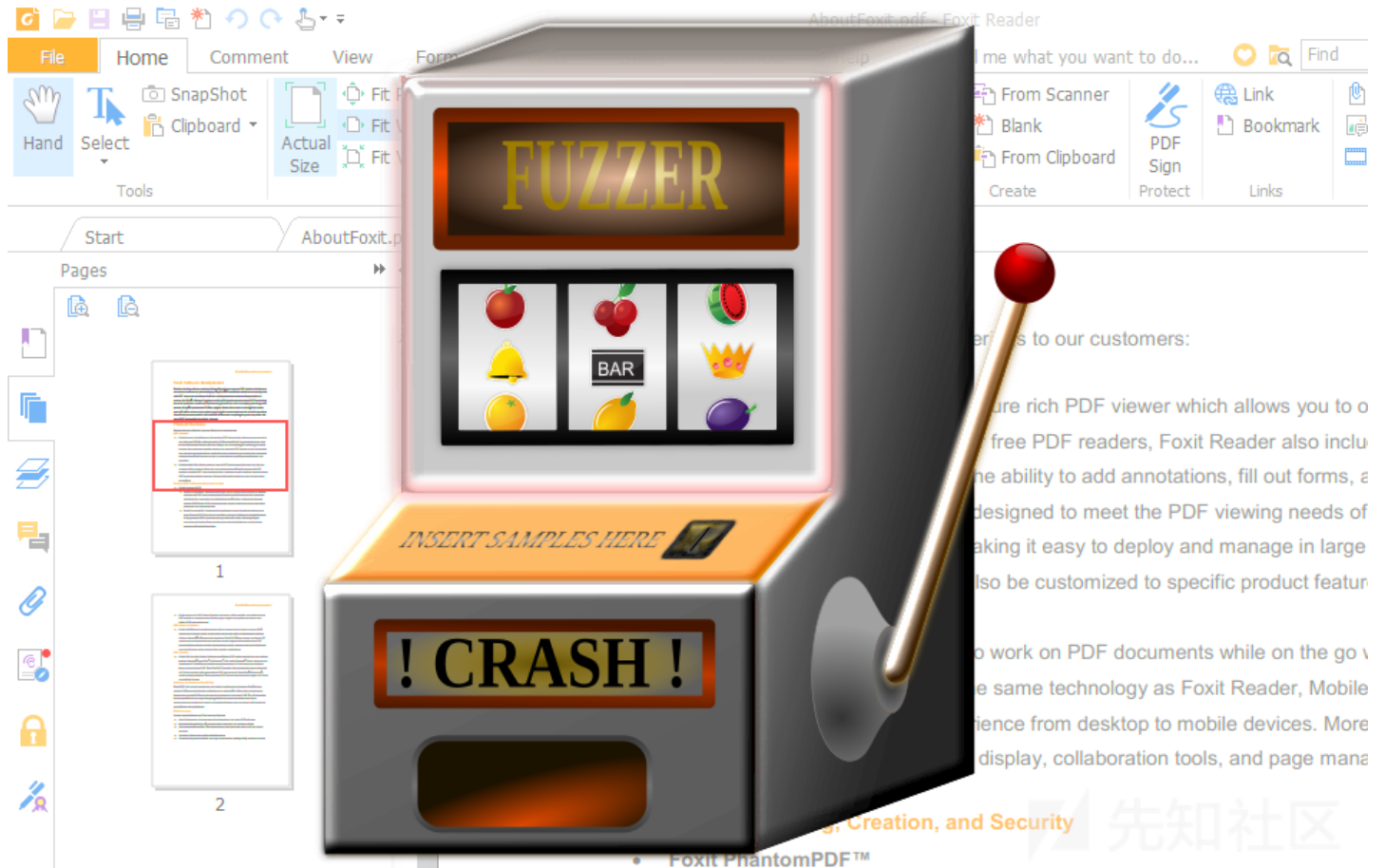


本文为翻译文章，原文链接：<https://www.gosecure.net/blog/2019/07/30/fuzzing-closed-source-pdf-viewers>

此文章介绍了fuzz闭源pdf查看器时出现的典型问题以及可能的解决方案。因此，它着重于两者：输入最小化和程序并未终止

这些方法是作为我的硕士论文的一部分找到并实施的，我在德国达姆施塔特工业大学与Fraunhofer SIT合作撰写了这篇论文。



## Context ( 问题背景 )

模糊PDF查看器的核心思想非常简单：选择一个PDF文件，稍微破坏它并检查它是否会使查看器崩溃。

虽然这听起来非常简单，但要正确有效地完成它却非常困难。PDF文件格式是目前使用最多且最重要的格式之一。因此，PDF查看器的安全问题在过去已被广泛利用并不是一

链接：<https://www.exploit-db.com/exploits/34603>

并且在2018年又被成功攻破

链接：<https://blog.malwarebytes.com/threat-analysis/2018/05/adobe-reader-zero-day-discovered-alongside-windows-vulnerability/>

报告给主要PDF查看器的非常多的问题表明仍然有许多的强化工作需要完成，我想为这两者做出贡献：PDF查看器的安全性和模糊测试社区。

对PDF查看器进行fuzz时通常会出现的问题是：

Fuzzer可以确定没有发生crash吗？

PDF查看器从不表示它已经完成了解析并呈现给定的PDF，应用程序将在何时关闭？

应该选择哪些PDF作为突变模板？

所选PDF应尽可能多地涵盖目标代码。如果源代码不可用，如何有效地测量代码覆盖率？

## 问题1：非终止程序

Fuzzee的（正常）终止向Fuzzer发出信号，表示它已完成处理并且没有发生崩溃。这对于Fuzzer很重要，因为它现在可以开始下一次测试迭代。PDF查看器的问题在于它们

大多数现有的Fuzzers所做的是，他们要么使用硬编码超时，否则如果没有发生崩溃就会杀死应用程序，或者他们不断地轮询目标的CPU周期量并假设当某个参数低于阈值时不管是哪种方法，超时或者阈值都必须精确设定，但或多或少都会猜到：对于Fuzzer来说，这意味着它太早（可能会丢失崩溃）或太迟（浪费时间）杀死应用程序。

方法：使程序中中止！

我们的想法是找到查看器的最后一个基本块，当它被赋予有效输入时执行。这里的假设是，只有当查看器完全解析并呈现给定的PDF时，才会执行此基本块。下一个测试开始

为了找出程序中哪些基本块已在运行时执行，研究人员利用了一个名为程序跟踪(Program Tracing)的概念。我们的想法是让目标生成有关其执行（跟踪）的附加信息，例如内存使用，采用分支或执行的基本块。

由于目标不是创建这些信息，因此必须向其添加说明。此过程称为程序检测(Program Instrumentation)。


在开源环境中，目标程序可以简单地使用其他编译器扩展例如AddressSanitizer(ASAN)进行重新编译，这些扩展将负责在编译时添加检测。显然，这对于闭源PDF查看器来

幸运的是，令人惊奇的框架DynamoRIO不需要任何源代码来应用此工具，因为它在运行时检测程序（动态二进制检测）

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

创建的程序跟踪看起来像这样：

```
Columns: id, base, end, entry, checksum, timestamp, path
0, 0x00400000, 0x00410000, 0x00401000, 0x000153f1, 0x6369646e, C:\Users\IEUser\Desktop\shalsum_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, 0x76e00000, 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, 0x77d00000, 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
```



上图为DynamoRIO的输出

可以看出，跟踪显示了哪个模块的基本块已经执行，并且它保留了基本块的顺序，这使得确定最后一个基本块变得相当容易。因此，要找出目标PDF查看器执行的最后一个基本块，通过向其提供不同但有效的PDF来创建多个跟踪。然后很明显，在靠近迹线末端的某处通常存在一个共同的基本块，这是必须被检测的块。

不幸的是，只有在程序退出后才会将跟踪写入磁盘，因此必须在此处使用较高超时阈值，当然这是一个硬编码数值。

现在找到了最后一个公共基本块，需要对其进行修补，以便终止程序。这可以通过覆盖基本块来实现：

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

这个问题是它需要9 bytes来表示这些指令。如果基本块的大小不是9 bytes，则后续指令将被破坏。

为了解决这个问题，可以在PE文件中添加一个新的可执行部分，其中包含上面的指令。因此，可以通过跳转到新添加的部分来修补基本块：

```
push SectionAddress
ret
```

为了修补目标，可以使用框架LIEF，这使得更改给定的PE文件变得相当容易。

译者注：这里推荐一个相关的CTF binary patch教程，其中有一些对liep的使用说明

<http://p4nda.top/2018/07/02/patch-in-pwn/#%E4%BF%AE%E6%94%B9%E7%A8%8B%E5%BA%8F-eh-frame%E6%AE%B5>

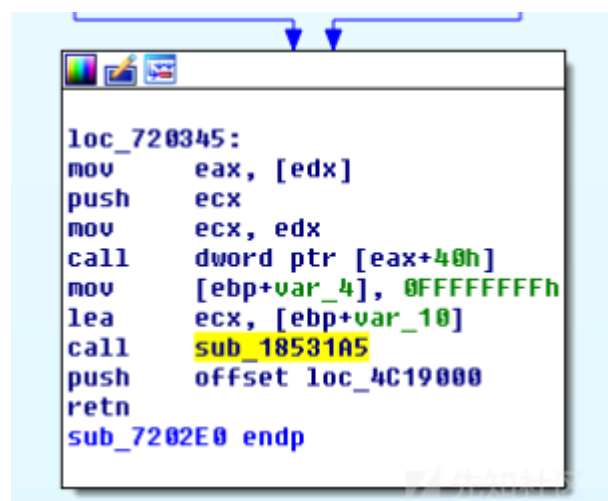
显然，使用断点修补基本块要容易得多，这是一个单字节指令。许多现有的Fuzzers依赖于程序崩溃会终止的事实，但不能用于PDF查看器。我们应用的退出检测(exit instrumentation)是检测崩溃更为容易和准确。

该方法是自动化的，并成功用于多个PDF和图像查看器：

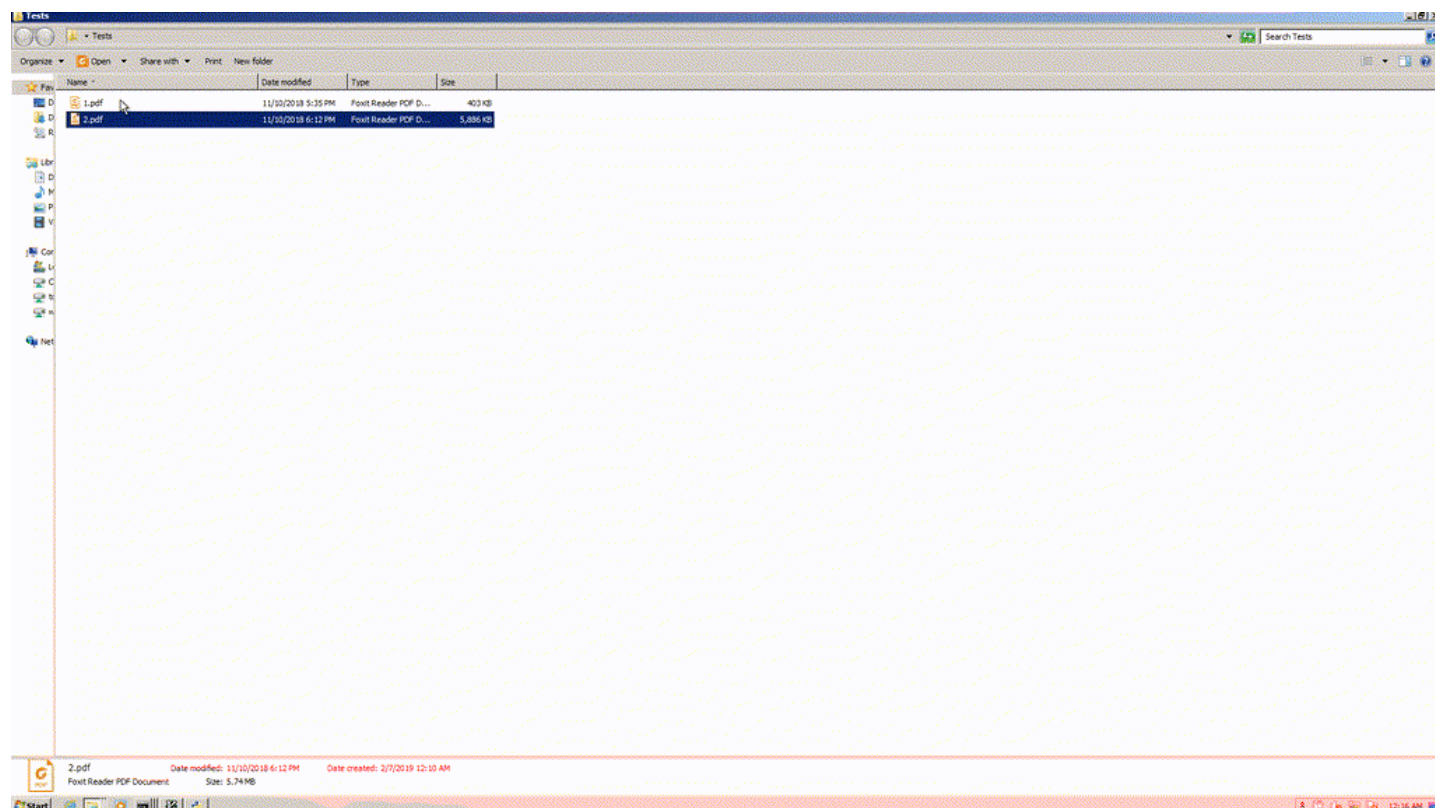
- FoxitReader
- PDFXChangeViewer
- XNView

下图给出了补丁在汇编层面上的体现。请注意，修补后的版本将返回到该新添加的部分。

带终止补丁的基本块



FoxIt Reader的行为



## 问题2：输入最小化

Fuzzing的成功在很大程度上取决于初始输入集（语料库）。因此，必须确保语料库尽可能多地覆盖目标代码，因为它显然增加了在其中发现错误的机会。此外，必须避免语料库中的冗余，以便每个PDF触发目标中的独特行为。

对此的常见方法是称为语料库蒸馏(Corpus Distillation)

这样做的核心思想是首先收集大量有效的输入。  
然后，对于每个输入，测量基本块代码覆盖，并且如果输入仅触发先前输入已经访问过的基本块，则将其从集合中移除。

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

同样，需要创建程序跟踪。由于源代码不可用，动态二进制检测似乎是测量基本块代码覆盖率的唯一机会。  
这里的问题是动态二进制检测似乎会产生不可接受的开销。

为了证明这一点，FoxitReader使用AutoExit方法来patch，并且测量直至终止的时间

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

在这里，动态二进制检测会导致近5秒的开销，耗时太高而无法执行有效的语料库蒸馏。

## 解决方案：自定义调试器

由于动态二进制检测显然太慢而无法执行语料库蒸馏，因此必须找到另一种方法来测量基本的块代码覆盖率。这个想法包含两部分：

1. 静态检测二进制文件
2. 创建一个处理检测的自定义调试器

首先，使用断点(单字节指令0xcc)对目标中的每个基本块进行修补。补丁静态应用于磁盘上的二进制文件。如果执行了任何基本块，它将触发断点事件(int3)，该事件可由监

下图显示了检测的基本块：

```
public _A
proc near
int 3 ; $!
mov ebp, esp
nop
pop ebp
retn
endp ; sp-analysis failed
```

```
===== S U B R O U T I N E =====

public _B
proc near
int 3 ; $!
mov ebp, esp
nop
pop ebp
retn
endp ; sp-analysis failed
```

```
===== S U B R O U T I N E =====

public _C
proc near
int 3 ; $!
mov ebp, esp
nop
pop ebp
retn
endp ; sp-analysis failed
```

由于断点，调试器很容易识别哪些基本块已被执行。

为了评估这种方法的性能，FoxitReader的所有基本块都使用断点进行patch（1778291个基本块）。

在第一次迭代中，FoxitReader花了16秒直到最终终止，这比DynamoRIO慢10秒。但是由于磁盘上的二进制文件中的断点已经被还原，它们将永远不会再触发int 3事件。

因此，可以假设在第一次迭代之后，大多数断点已经被恢复，因此开销应该是合理的。

- 第一次迭代：16秒（48323个断点）
- 第二次迭代：2秒（2212个断点）
- 第三次及之后：~1.5秒（非常少的断点数）

可以看出，在第一次迭代之后，检测导致最小的开销，但是调试器仍然能够确定任何新访问的基本块。

这种方法在主要产品上进行了测试，并在所有这些方面完美运行：

- Adobe Acrobat Reader
- PowerPoint
- FoxitReader

## Fuzzing

通过爬虫在互联网收集了80000个PDF，并将该集合最小化为220个独特的PDF，耗时约1.5天。使用此最小化设置进行Fuzz的结果非常好，并且所有崩溃都被推送到数据库中：

显示模糊测试结果

All crashes					
id	timestamp	exploitability_rule	stack_trace	exploitability_rating	exploitability_desc
43.00	2018-10-02T19:00:40Z	WriteAV	...	Probably exploitable	User mode write access violations that are near NULL are probably exploitable.
42.00	2018-09-29T13:38:31Z	WriteAV	...	Probably exploitable	User mode write access violations that are near NULL are probably exploitable.
41.00	2018-09-27T19:05:18Z	ReadAVNearNull	...	Not likely exploitable	This is a user mode read access violation near null, and is probably not exploitable.
40.00	2018-09-27T11:59:57Z	ReadAVNearNull	...	Not likely exploitable	This is a user mode read access violation near null, and is probably not exploitable.
39.00	2018-09-16T20:20:31Z	Unknown	...	Unknown	Exploitability unknown.
38.00	2018-09-16T09:05:27Z	ReadAVNearNull	...	Not likely exploitable	This is a user mode read access violation near null, and is probably not exploitable.
37.00	2018-09-12T09:42:05Z	Unknown	...	Unknown	Exploitability unknown.
36.00	2018-09-10T14:39:26Z	WriteAV	...	Probably exploitable	User mode write access violations that are near NULL are probably exploitable.
35.00	2018-08-31T21:15:58Z	Unknown	...	Unknown	Exploitability unknown.
34.00	2018-08-30T18:40:48Z	ReadAVNearNull	...	Not likely exploitable	This is a user mode read access violation near null, and is probably not exploitable.
33.00	2018-08-28T11:58:03Z	Unknown	...	Unknown	Exploitability unknown.
32.00	2018-08-27T16:47:17Z	Unknown	...	Unknown	Exploitability unknown.
31.00	2018-08-27T10:01:51Z	Unknown	...	Unknown	Exploitability unknown.
30.00	2018-08-27T04:00:12Z	WriteAV	...	Probably exploitable	User mode write access violations that are near NULL are probably exploitable.
29.00	2018-08-26T19:34:56Z	Unknown	...	Unknown	Exploitability unknown.
28.00	2018-08-26T00:40:44Z	ReadAVNearNull	...	Not likely exploitable	This is a user mode read access violation near null, and is probably not exploitable.
27.00	2018-08-24T23:08:59Z	WriteAV	...	Probably exploitable	User mode write access violations that are near NULL are probably exploitable.

结果

最终，Fuzzer在大约2个月的时间框架内发现了43起独特的崩溃事件，其中三起足以将其报告给Zero Day Initiative。

它们被分配了以下ID：

- ZDI-CAN-7423：Foxit Reader解析越界读导致RCE
- ZDI-CAN-7353：Foxit Reader解析越界读导致信息泄露
- ZDI-CAN-7073：Foxit Reader解析越界读导致信息泄露

点击收藏 | 0 关注 | 1

[上一篇：Apache Solr Injec...](#) [下一篇：渗透测试初期之信息收集](#)

1. 0 条回复

- 动动手指，沙发就是你的了！

[登录](#) 后跟帖

先知社区

[现在登录](#)

热门节点

[技术文章](#)

[社区小黑板](#)

目录

[RSS](#) [关于社区](#) [友情链接](#) [社区小黑板](#)