使用Fuzz技术进行漏洞挖掘

2018年了，想要学习二进制漏洞挖掘如何开始?

[+]by Asprose

目录

01 Fuzz 简介

02 如何编写自己的Fuzz

03 基于覆盖率指导的Fuzz

00 序

什么是二进制漏洞?

二进制漏洞的应用场景?

二进制漏洞的分类?

[+]by Asprose

00 序

有那些比较出名的二进制漏洞应用场景?

[+]by Asprose

00 序

什么是PWN2OWN?

Pwn2Own是全世界最著名、奖金最丰厚的黑客大赛，由美国五角大楼网络安全服务商、ZDI，谷歌、微软、苹果、Adobe等互联网和软件巨头都对比赛提供支持，通过黑客攻击挑战来完善自身产品。

00 序

PWN2OWN的漏洞目标有那些?

vmware

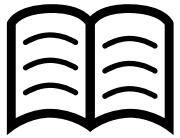
[+]by Asprose

01

Fuzz 简介

[+]by Asprose

01 Fuzz 简介



模糊测试（Fuzzing），是一种通过向目标系统提供非预期的输入并监视异常结果来发现软件漏洞的方法。主要是通过输入大量数据，发现程序中存在的问题。可以通过使程序某些内容溢出出现异常，或者输入的是程序规定的范围内的数据结果出现异常，从而找出程序的bug。

模糊测试的实现是一个非常简单的过程：

1、准备一份输入目标程序中的文件，比如一个txt文档或者一个doc文档。

2、用程序打开文件。

3、观察程序是否正常运行。

4、若正常直接回到步骤1，若崩溃则保存导致目标程序崩溃的文件再回到步骤1

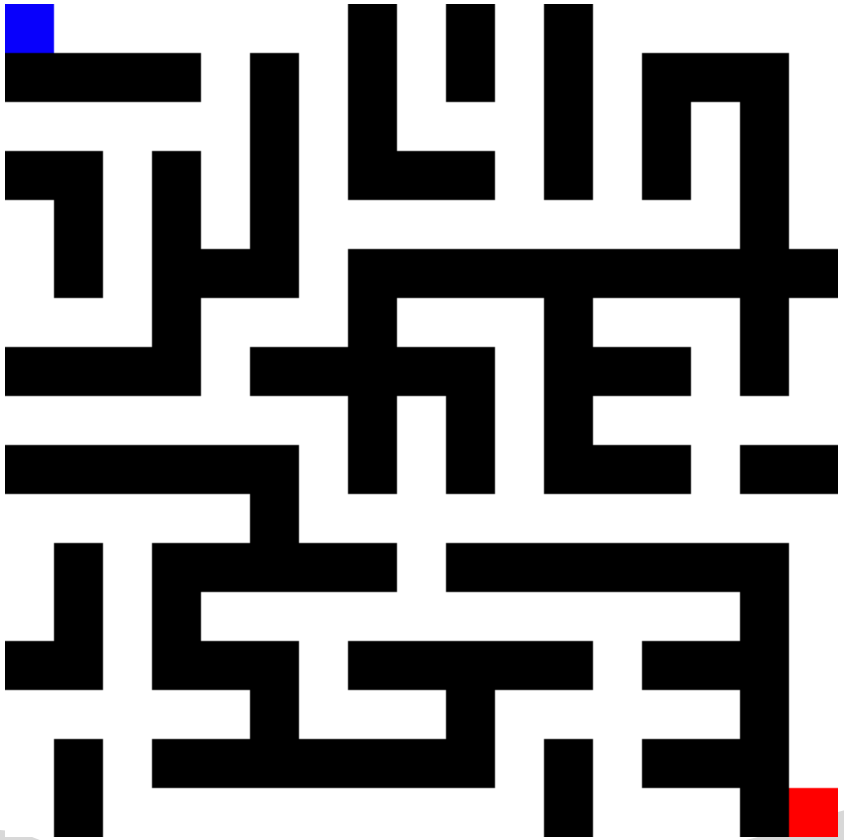
01 Fuzz 简介

漏洞挖掘模型类似走迷宫。迷宫的入口即是传入的数据入口，迷宫中的线路就是数据传递的线路。

如何在迷宫走的更远，更多，发现迷宫的更多的路径是当前漏洞挖掘的一个研究点。毕竟高的路径数，通常代表着更高的出洞概率。

另一方面，不同迷宫入口也代表不同的漏洞攻击点它能带来的攻击面和攻击矢量都是不同的。比像浏览器的漏洞挖掘，入口就颇多，js 解析引擎、各种小文件解析、pdf、flash、视频、音频等等。

不同的攻击面选择带来的还有挖掘的难易，比如js解析引擎的挖掘难度就相当大。

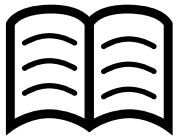


02

如何编写自己的Fuzz

[+]by Asprose

02 如何编写自己的Fuzz



0x01 编写属于自己的Fuzz 的我们需要解决的问题

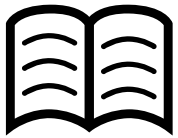
如何提供大量高效的文件，让fuzzer快速得到想要的结果?

如何有效的监测的目标程序的状态?

如何提高Fuzz 的执行速度?

如何去除重复的崩溃?

02 如何编写自己的Fuzz



0x02如何提供大量高效的文件

[+] 气宗

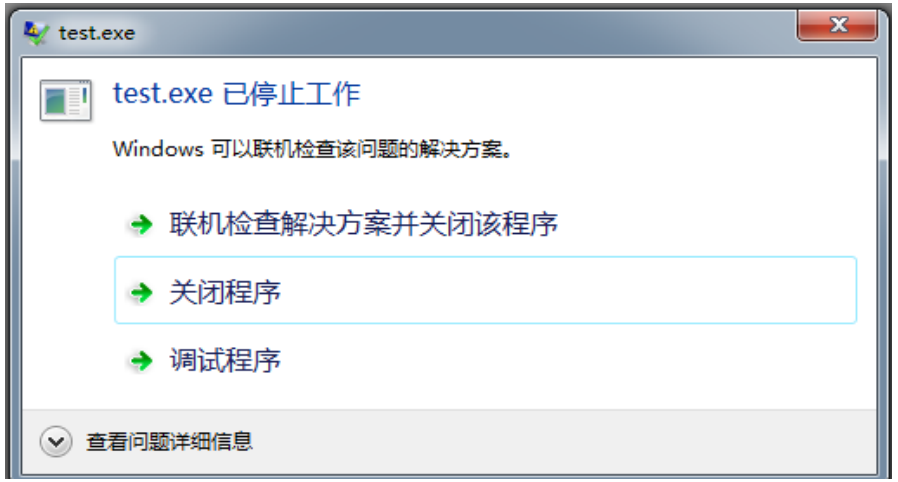
• 假如我们Fuzz 的目标是最简单的文件格式Fuzz,通过阅读目标程序的 官方文档，了解通透该文件格式,编写程序从零开始构造文件。这是一个相当费力费时间的过程，甚至大概率最后浪费大量的时间也没有结果，但是这是正常的。

[+]剑宗

• 凭本事找到或要到一些骨骼清奇的文件，然后对文件的内容对替换修改，得到新的文件。这需要一开始的初始文件具备特殊性和代表性，以及后面的修改操作具有合理性。

02 如何编写自己的Fuzz

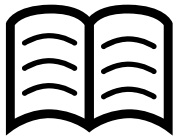
0x03 如何有效的监测的目标程序的状态?



最简单的解决方法，检测windows的崩溃报错程序是否出现来检测崩溃的存在。
看起来好像可行，但是考虑多线程和速度来说就根本不可行。

稍微靠谱的方法，想想为什么在使用OD/GDB调试程序的时候，异常地址访问，各种异常调试器都能捕获到?

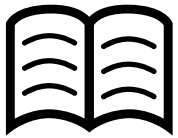
02 如何编写自己的Fuzz



0x03 如何有效的监测的目标程序的状态?

> 通过查找阅读网上开源的win平台调试器的源码，得到一个结论调试器通过windows API *CreateProcess* 函数来启动目标程序并开始调试。CreateProcess有参数可以设定子进程的状态。

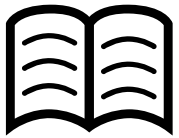
02 如何编写自己的Fuzz



0x03 如何有效的监测的目标程序的状态?

```
ret = CreateProcess(NULL,    // target file name.
(LPSTR)commmand.data(),   // command line options.
NULL,                     // process attributes.
NULL,                     // thread attributes.
FALSE,                    // handles are not inherited.
DEBUG_PROCESS,            // debug the target process and all spawned children.
NULL,                     // use our current environment.
NULL,                     // use our current working directory.
&si,                      // pointer to STARTUPINFO structure.
(LPPROCESS_INFORMATION)&pi);// pointer to PROCESS_INFORMATION structure.
```

02 如何编写自己的Fuzz

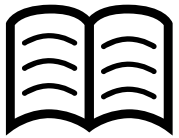


0x03 如何有效的监测的目标程序的状态?

```
switch (dbg.u.Exception.ExceptionRecord.ExceptionCode){
case EXCEPTION_ACCESS_VIOLATION:
        exception = TRUE;
        printf("[*] Access Violation\n");
case EXCEPTION_INT_DIVIDE_BY_ZERO:
        exception = TRUE;
        printf("[*] Divide by Zero\n");
case EXCEPTION_STACK_OVERFLOW:
        exception = TRUE;
        printf("[*] Stack Overflow\n");
}
```

通过对被调试的目标的程序检测它的异常代码，我们可以实时的得到目标程序的状态，这样的崩溃检测高效而且准确，甚至能提供崩溃类型。

02 如何编写自己的Fuzz



0x03 如何有效的监测的目标程序的状态?

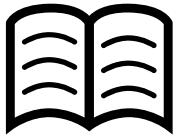
> 承然，在windows 上可以使用这种类似调试器Debug 目标程序的方法来获取目标程序的异常状态，但在linux 下有一些更为简单的思路。比如利用Asan。

Asan：

Asan是Linux下的内存检测工具,ASAN（Address-Sanitizier）早先是LLVM中的特性，后被加入GCC 4.8，在GCC 4.9后加入对ARM平台的支持。因此GCC 4.8以上版本使用ASAN时不需要安装第三方库，通过在编译时指定编译CFLAGS即可打开开关。

```
>gcc test.c –o test CFLAGS="-g -fsanitize=address"
```

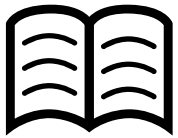
02 如何编写自己的Fuzz



0x03 如何有效的监测的目标程序的状态?

> 当Asan检测到有内存错误时，将直接终止程序，输出错误的内存信息，直接退出程序，并且输出的错误信息里有详细的内存错误报告，以及内存错误的类型。通过利用检测Asan的错误信息报告，我们也可以快速的检测到目标程序是否正常。（当然linux 下使用Asan 的作用就和 windows 下使用 pageheap 一样重要都是未了检测一些不会导致程序崩溃的隐藏内存问题）

02 如何编写自己的Fuzz



0x03 如何有效的监测的目标程序的状态?

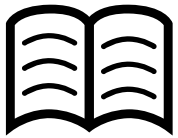
Asan的错误输出信息如下:

```
==7402==ERROR: AddressSanitizer: stack-buffer-overflow on address 0x7fff2971ab88 at pc
 0x400904 bp 0x7fff2971ab40 sp 0x7fff2971ab30READ of size 4 at 0x7fff2971ab88 thread T0

        #0 0x400903 in main /tmp/test.c:3
        #1 0x7fd7e2601f9f in __libc_start_main (/lib64/libc.so.6+0x1ff9f)
        #2 0x400778 (/tmp/a.out+0x400778)
Address 0x7fff2971ab88 is located in stack of thread T0 at offset 40 in frame
        #0 0x400855 in main /tmp/test.c:1     This frame has 1 object(s):     [32, 40) 'a' <==
Memory access at offset 40 overflows this variable
HINT: this may be a false positive if your program uses some custom stack unwind mecha
nism or swapcontext      (longjmp and C++ exceptions *are* supported)

SUMMARY: AddressSanitizer: stack-buffer-overflow /tmp/test.c:3 main
```

02 如何编写自己的Fuzz



0x04 如何提高Fuzz 的执行速度?

采用多线程的方式来启动目标程序，同时执行多个样本。但是这样需要 目标程序切换到单进程模式，比如IE浏览器，若不采用单进程模式，开启再多的浏览器进程，也是在一个总进程里。

IE 设置单进程的方法：

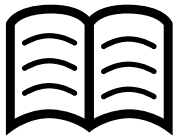
[+]打开regedit

[+]依次展开HKEY_CURRENT_USER\Software\Microsoft\Internet Explorer\Main分支

[+]新建一个DWORD32值，并将其命名为TabProcGrowth，在弹出的对话框中输入“0”或者“1”

[+]修改好键值之后，重启explorer.exe，使注册表值生效。

02 如何编写自己的Fuzz

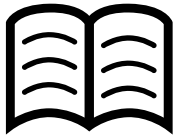


0x04 如何提高Fuzz 的执行速度?

采用内存存储目标程序和样本，在内存里跑fuzz ,fuzz的运行中有大量的时间会涉及到I/O 操作，比如从磁盘里加载目标程序到内存，生成样本文件并写入磁盘，从磁盘读取样本文件，这一系列操作对磁盘速度有着不小的要求，特别是在多线程跑fuzz的时候，会有大量的磁盘读写操作。 所以若将目标程序和样本存储在内存中，则会大大提高I/O速度。

Win上可以使用很多第三方软件实现划一片内存为虚拟磁盘，比如RAMDisk，而且各种第三方软件网上教程也多，便不多述。下面讲讲Linux 下自带的内存文件系统。

02 如何编写自己的Fuzz



0x04 如何提高Fuzz 的执行速度?

Tmpfs:

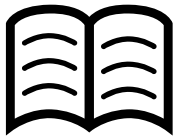
tmpfs是一种虚拟内存文件系统正如这个定义它最大的特点就是它的存储空间在VM VM(virtual memory)里面。

怎样使用tmpfs呢?

```
#mount  -t tmpfs -o size=20m  tmpfs /mnt/tmp
```

上面这条命令分配了上限为20m的VM到/mnt/tmp目录下，用df命令查看一下，确实/mnt/tmp挂载点显示的大小是20m，但是tmpfs一个优点就是它的大小是随着实际存储的容量而变化的，换句话说，假如/mnt/tmp目录下什么也没有，tmpfs并不占用VM。上面的参数20m只是告诉内核这个挂载点最大可用的VM为20m，如果不加上这个参数，tmpfs默认的大小是RM的一半，假如你的物理内存是128M，那么tmpfs默认的大小就是64M。

02 如何编写自己的Fuzz

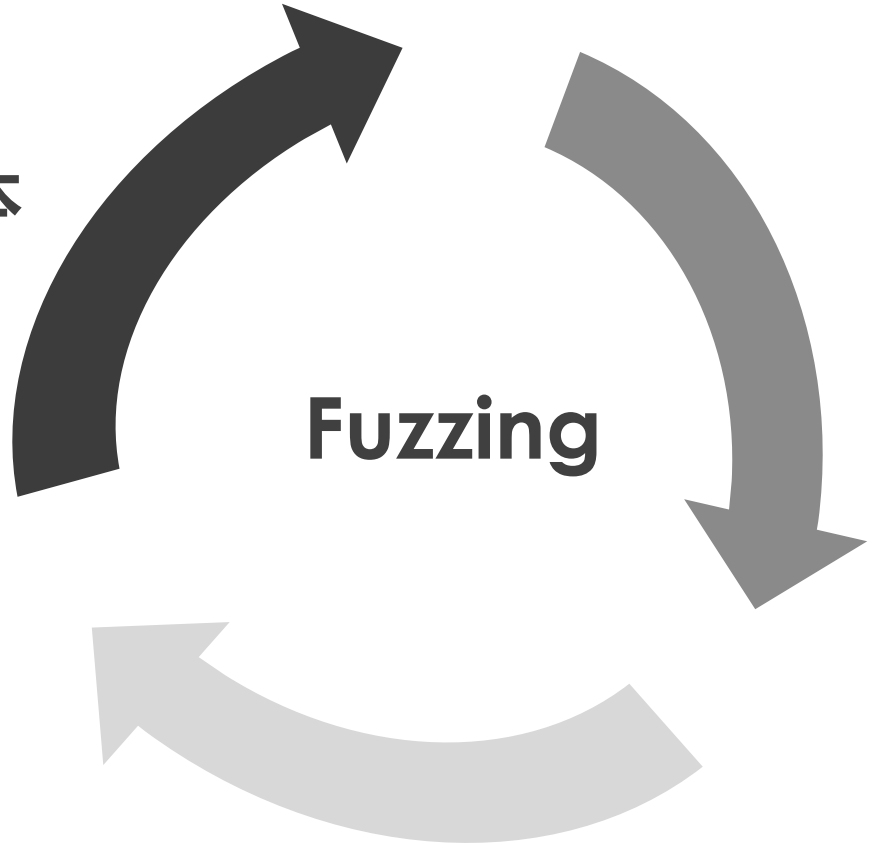


0x05 如何去除重复的崩溃?

当fuzz 发现了大量的崩溃，或者说得到崩溃时，需要自动化进行去重。否则会给后续的崩溃分析得到漏洞这个过程增加巨大的工作量，比如辛辛苦苦的在汇编的海洋里遨游分析了五六个样本后，得到了悲愤的结论，这些样本都是一个漏洞，这种体会不仅让人难受，更加浪费时间和精力，毕竟调试是个体力活。

去重的方式，可以获取崩溃点的汇编代码的地址偏移进行比较得到，是否是同一崩溃点，因为不同的文件可以在同一个代码点崩溃。

02 如何编写自己的Fuzz



根据崩溃结果判断是否存储样本

03

基于覆盖率指导的新式Fuzz

[+]by Asprose

Google Project zero

Project Zero是Google公司于2014年7月15日所公开的一个信息安全团队，此团队专责找出各种软件的安全漏洞

Ivan Fratric

Parisa Tabriz

lcamtuf

j00ru

Natalie Silvanovich

Tabriz

[+]by Asprose

开源 AFL 项目

AFL由lcamtuf所开发，并且开源。在github和lcamtuf的博客上分别能获取到win版本和 linux 版本的afl。
http://lcamtuf.coredump.cx/afl/

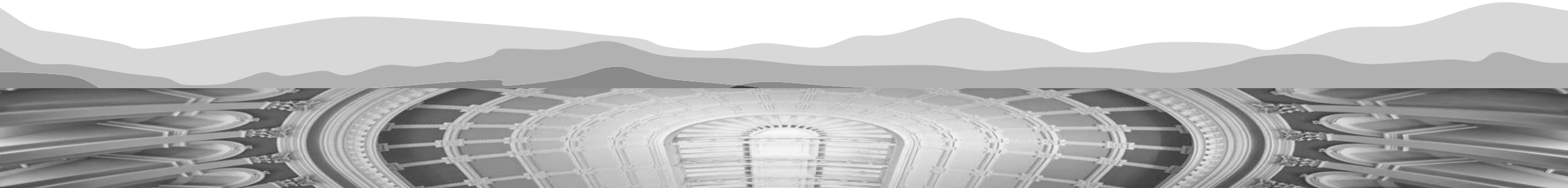
https://github.com/googleprojectzero/winafl

03 基于覆盖率指导的Fuzz

0x01 AFL 简介

American fuzzy lop是一种面向安全的 模糊器，它采用一种新型的编译时间仪器和遗传算法来自动发现触发目标二进制文件中新内部状态路劲的有趣的测试用例。这大大改善了模糊代码的功能覆盖。

与其他模糊器相比，afl-fuzz设计实用：它具有适度的性能开销，使用各种高效的模糊测试策略和努力最小化样本的技巧， 基本上不需要配置，并且无缝地处理复杂的，目标程序Fuzz。

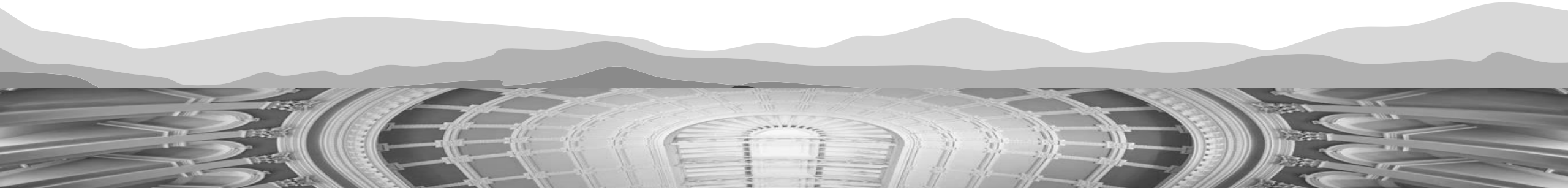


03 基于覆盖率指导的Fuzz

0x01 AFL 简介

American fuzzy lop是一种面向安全的 模糊器，它采用一种新型的编译时间仪器和遗传算法来自动发现触发目标二进制文件中新内部状态路劲的有趣的测试用例。这大大改善了模糊代码的功能覆盖。

与其他模糊器相比，afl-fuzz设计实用：它具有适度的性能开销，使用各种高效的模糊测试策略和努力最小化样本的技巧， 基本上不需要配置，并且无缝地处理复杂的，目标程序Fuzz。



03 基于覆盖率指导的Fuzz

0x02 AFL 工作流程

整体算法可归纳为：

1）将用户提供的初始测试用例加载到队列中

2）从队列中获取下一个输入文件

3）尝试将测试用例修剪到不会改变的最小尺寸测量的程序行为

4）使用平衡且经过充分研究的品种反复改变文件传统的模糊策略

5）如果任何产生的突变导致新的状态转换由仪器记录，添加变异输出作为新的条目队列。

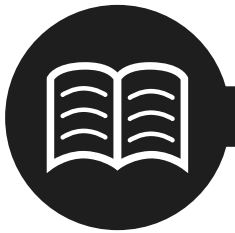
6）转到2。

03 基于覆盖率指导的Fuzz

0x03 AFL的优点

0x00基于覆盖率能有效判断挖掘的进度

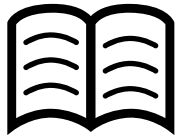
0x01基于覆盖率能有效的判断样本的质量



0x03 AFL 执行速度超级超级快

0x02 AFL 自带某些文件的词典

03 基于覆盖率指导的Fuzz



0x05 AFL 的简单使用

AFL 在linux 对有源码的开源C/C++项目进行漏洞挖掘是最简单情况，当然AFL 也可以对闭源程序Fuzz 比如，qemu模式和WinAFL。

AFL 使用afl-gcc/afl-g++ 把开源项目编译后，实际上已经完成了afl的插桩，通过这些插桩，afl在fuzz的过程中，可以获取到目标程序那些路径执行了，以及整个样本集的能获取到代码覆盖率。

afl 的启动方式为：

```
>./afl-fuzz -i  input  -o  output  -t 30000 -m 1024 --  your_target @@
```

-i 指定初始样本集文件夹的路径

-o指定fuzz 结果的存储文件夹路径

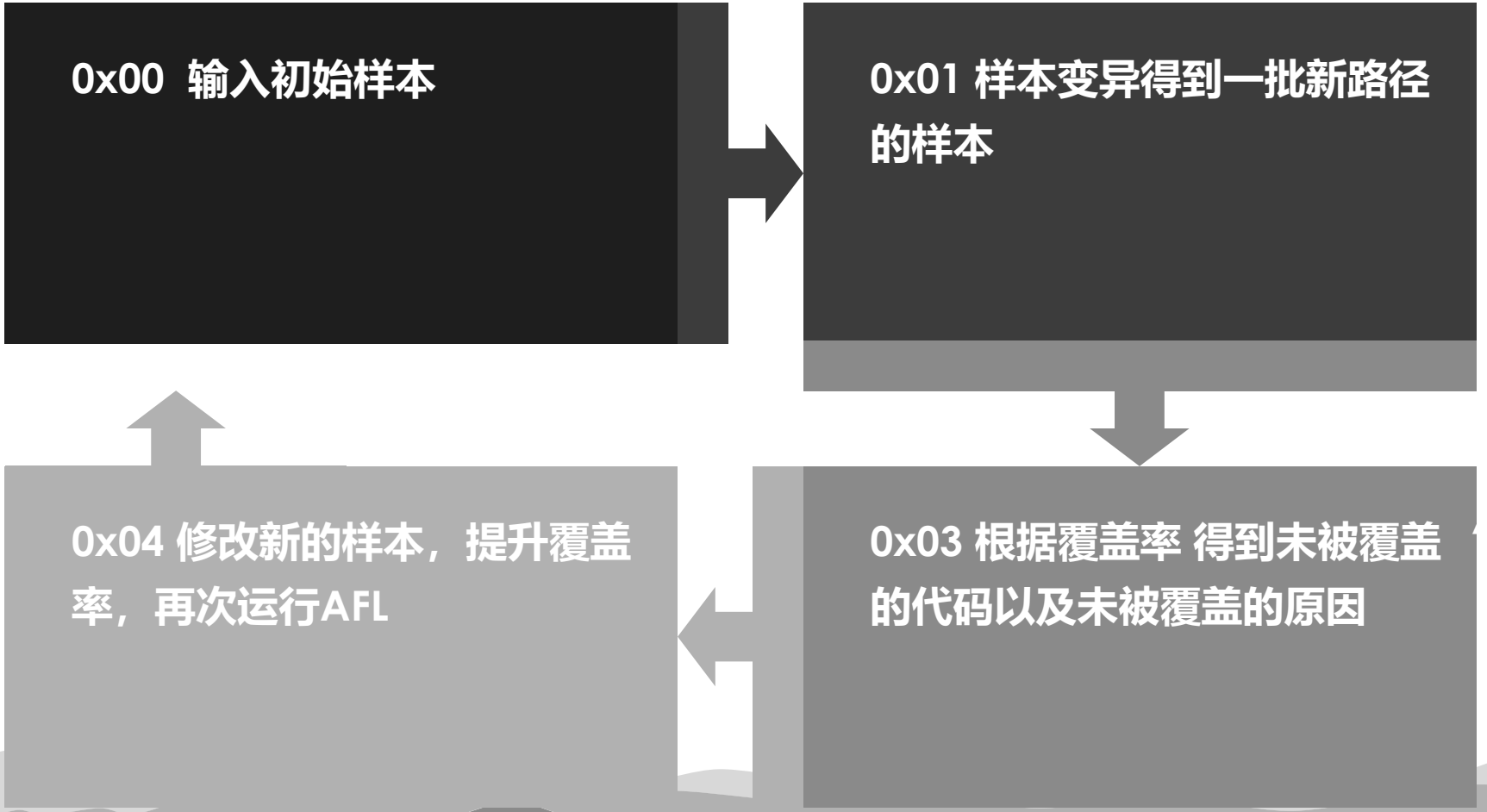
-t 指定fuzz 等待的时间

-m指定afl的内存限制

@@ 则是afl fuzz的时候会把样本路径替换@@

03 基于覆盖率指导的Fuzz

0x04 AFL的漏洞挖掘流程



03 基于覆盖率指导的Fuzz

0x06 AFL漏洞挖掘的tips

感谢聆听！

[+]by Asprose