模糊测试实验说明

December 20, 2022

1 实验背景

模糊测试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安全漏洞的测试方法。

核心工作流程是从测试用例队列中筛选 (*Select*) 出合适的测试用例，并对该用例变异 (*Mutate*) 后将其作为新一轮的测试用例，最后对运行结果进行观测 (*Observe*) 来决定是否将该用例加入队列中。在这个过程中，我们将上述半自动化生成的测试用例作为待测程序的输入，并监测程序异常，如崩溃、断言 (assertion) 失败、内存泄漏……

2 实验任务

本次实验在我们提供的模糊测试框架上完成。框架里给出了模糊测试的核心流程，如图 Figure 1

```
1.  #include "fuzzer.h"                //实现了Init、Execute、Save函数

2.  typedef struct LinkNode {
3.    char fname[256];                //测试用例文件名
4.    struct LinkNode *next;          //队列中的下一个元素
5.  } LinkNode;

6.  typedef struct {
7.    LinkNode *front, *rear;         //队头指针和队尾指针
8.    int cnt;                        //队列元素计数
9.  } LinkQueue;

10. LinkQueue queue;                  //测试用例队列的队头元素
11. LinkNode *target;                 //每次选取的目标测试用例
12. LinkNode *newone;                 //每次变异后的测试用例

13. LinkNode * Select(LinkQueue *);         //待实现的筛选函数
14. LinkNode * Mutate(LinkNode *);          //待实现的变异函数
15. int Observe(void);                      //待实现的观测函数

16. int main()  {
17.   int result;
18.   Init(&queue);                   //初始化测试用例队列
19.   while (1) {
20.     target = Select(&queue);      //筛选目标测试用例
21.     newone = Mutate(target);      //变异目标测试用例
22.     Execute(newone);              //执行变异后的测试用例
23.     result = Observe();           //观测执行结果
24.     if (result&1 == 1 || result&2 == 2)//新覆盖或新状态
25.       Save(&queue, newone);  //加入测试用例队列
26.   }
27. }
```

Figure 1: 模糊测试的核心流程

1. L18 $Init(\&queue)$: 初始化测试用例队列 (queue)，将用户提供的一组合法输入当做种子用例 (seed) 放入队列中
2. L20 $target = Select(\&queue)$: 从队列中挑选出一个测试用例 (target)
3. L21 $newone = Mutate(target)$: 对 target 进行变异，得到新测试用例 newone
4. L22 $Execute(newone)$ 运行 newone，同时在该函数中记录下运行时的信息并存入 coverage(Figure 2) 与 state(Figure 3) 中。

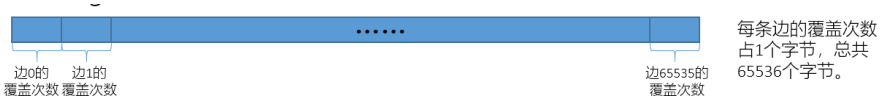

Figure 2: coverage 文件格式

Figure 3: state 文件格式

其中，各个比较符号对应的编码如: $\boxed{<:1}$、$\boxed{>:2}$、$\boxed{==:4}$、$\boxed{\leq:8}$、$\boxed{\geq:16}$、$\boxed{\neq:32}$

5. L23 $result = Observe()$: 对运行结果进行观测，并将观测值存入 result
6. L24,L25 根据 result 的值来决定是否保留测试用例 (在示例的实现中 result&1==1 表示此次运行有新的覆盖，result&2==2 表示此次运行有新的状态)

你的任务 实现 $Select()$，$Mutate()$，$Observe()$ 三个函数 (如果你认为有必要，也可修改 fuzzer.c 中的其他代码片段，但请**不要**修改 libfuzzer.c 中的内容)，来使得你的模糊测试具有更高的效率，效率的量化指标为:

- ◇ 对待测代码的覆盖率
- ◇ 监测到的程序异常数目

3 运行环境说明

下发目录说明

运行时的目录结构如 Figure 4:

Figure 4: 模糊测试目录树

fuzzer 目录 fuzzer 包含模糊测试框架，其中已经编写 Makfile 文件。

program 目录 program 包含待测程序源代码 (位于目录 src)、可执行二进制文件 (opj_compress)。

demo 目录 demo 包含示例程序源代码 (demo.c)、可执行二进制文件 (demo) 以及模糊测试的种子用例 (seed)。

work 目录 work 是程序运行时目录，每次运行记录的 state 与 coverage 文件都在这个目录下。

dep 目录 dep 包含二进制程序 opj_compress 运行所需的相关依赖，在对 program 进行模糊测试之前，**需要在终端中先进入 dep 目录并运行命令** $\boxed{\textbf{./install.sh}}$

execute 脚本 execute 用于启动模糊测试，脚本需要接受一个参数表示待测程序目录。

以演示用例 demo.c 为例，你只需要进入 lab3 所在目录，并在终端输入 $\boxed{\text{./execute demo}}$ (如果是对 program 进行模糊测试，那么输入 $\boxed{\text{./execute program}}$) 即可对演示用例进行模糊测测试。

vulnerability 目录 vulnerability 包含 ppt 中《15.3.2 含有漏洞的待测程序——简易文本编辑器》一节所述内容，你可以运行 `./execute.sh` 来利用程序的缓冲区溢出漏洞在终端下调用日历命令 (/usr/bin/cal)。

运行时信息

在运行时，终端每隔一分钟会输出一行信息，如 Figure 5，其中包含了覆盖的代码边数以及观察到异常行为的次数 (对于异常行为，相同的代码边覆盖只记一次)

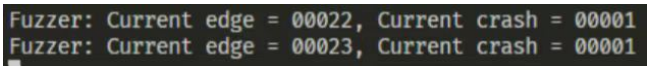

Figure 5: 运行时信息

4 提交说明

提交内容 1. 实验报告，2. 基于给出框架实现的相关代码 (包括但不限于 fuzzer.h，fuzzer.c，libfuzzer.c)

报告要求 小组提交的实验报告应该包含以下内容

- ◇ 小组的分工情况
- ◇ 包括但不限于模糊测试中你所实现的 $Select$、$Mutate$、$Observe$ 模块的说明

实验报告不鼓励长篇大论，说清楚即可

5 评分标准

测试说明 测试时将使用你所实现的 fuzzer 对二进制文件 program 进行长达 24h 的模糊测试，记录下最后的边覆盖数目以及触发异常的次数。

分数构成 本次实验共占 15 分，由模糊测试结果 (70%) 与实验报告 (30%) 组成。模糊测试结果的得分计算如下:

$$score_i = min\{\frac{coverage_i}{min\{X, max(coverage_1, coverage_2, coverage_3..., coverage_n)\}} \times 7.5\} + 3$$

其中 $score_i$ 为第 i 组的得分；$coverage_i$ 表示第 i 组在运行 24h 后代码边覆盖数；X 是一个待定的阈值。

附：漏洞说明

我们对源程序进行了修改，人为地增加了 5 个漏洞。

A 浅层漏洞 ($=3$)

这些漏洞相关的修改在下发的源代码中可见，你可以通过静态的方式去分析 3 个浅层漏洞；建议用分析的结果去引导新测试用例的生成，而不是直接构造一组测试用例。

漏洞 1 该漏洞位于"./src/bin/jp2/convertbmp.c, L366"

漏洞 2 该漏洞位于"./src/bin/jp2/convertbmp.c, L824"

漏洞 3 该漏洞位于"./src/bin/jp2/convertbmp.c, L217"

B 深层漏洞 (≥ 2)

包括程序本身存在的漏洞以及我们人为添加的漏洞

漏洞 4 所处的 BasicBlock 块编号为 `0xd664`

漏洞 5 所处的 BasicBlock 块编号为 `0xdc17`