浅析Peach Fuzz

安全研究部 刘永军

关键词：Peach Fuzz

摘要：本文简要介绍了 Fuzz 工具 Peach 的使用，并通过文件格式 Fuzz 举例阐述了 Peach Pit 文件的编写。

1 引言

Fuzz（模糊测试）是一种通过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方法。模糊测试在很大程度上是一种强制性的技术，简单并且有效，但测试存在盲目性。

典型地模糊测试过程是通过自动的或半自动的方法，反复驱动目标软件运行并为其提供构造的输入数据，同时监控软件运行的异常结果。

Fuzz 被认为是一种简单有效的黑盒测试，随着 Smart Fuzz 的发展，RCE（逆向代码工程）需求的增加，其特征更符合一种灰盒测试。

Peach 是一个优秀的开源 Fuzz 框架。

2 Peach 简介

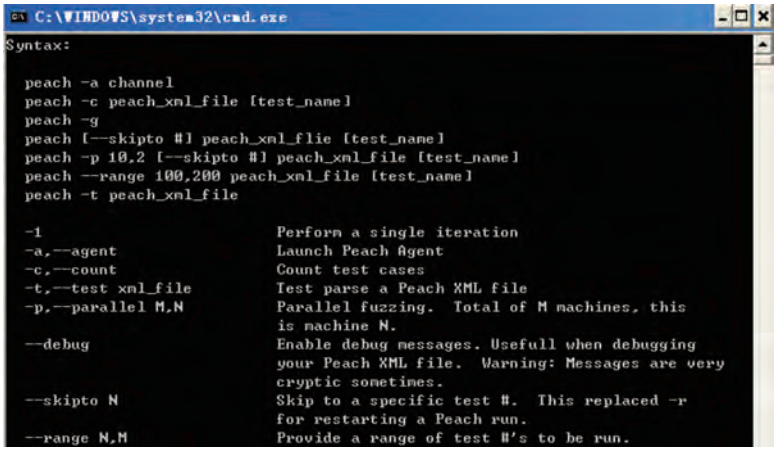
2.1 概述

Michael Eddington 等人开发的 Peach 是一个遵守 MIT 开源许可证的模糊测试框架，最初采用 Python 语言编写，发布于 2004 年，第二版于 2007 年发布，最新的第三版使用 C# 重写了整个框架。

Peach 支持对文件格式、ActiveX、网络协议、API 等进行 Fuzz 测试，Peach Fuzz 的关键是编写 Peach Pit 配置文件。

Windows 下使用 Peach3 需要预先安装 .net 4 和 windbg，Linux、OS X 下需要安装 Mono .net 开发框架。

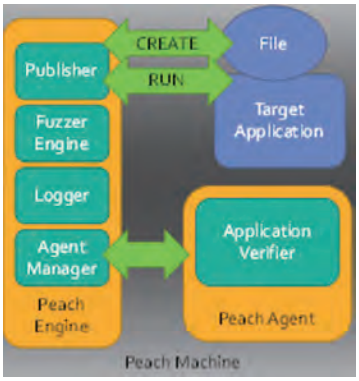
2.2 命令行参数



- -1：执行第 1 次测试。
- -a: 启动 Peach 代理。不指定 “channel” 默认为本地代理（默认支持, 无需显式启动）; “channel” 可以指定为 “tcp” 远程代理。
- -c：统计测试用例数。
- -t：验证 Peach Pit xml 文件正确性。
- -p：并行 Fuzz。运行 Peach 的机器总数为 M，这是第 N 个。
- --debug：调试信息开关。
- --skipto：指定 Fuzz 跳过的测试用例数。
- --range：指定 Fuzz 的测试用例范围。

3 Peach 文件 Fuzz

3.1 Peach 文件 Fuzz 流程图



3.2 Peach Pit

在使用 Peach 进行 Fuzz 之前需要编写被称为 “Peach Pit” 的 xml 配置文件，其中包含着如何进行 Fuzz 的关键信息，如下图:

```
<?xml ...版本，编码之类...>
<Peach ...版本，作者介绍之类...>
<Include ...包含的外部文件... />
<DataModel >原始数据结构定义，可嵌套</DataModel>
<StateModel >测试逻辑，状态转换定义</StateModel>
<Agent >监测被测目标的反应，如crash等</Agent>
<Test >指定使用哪个StateModel 、Agent 、Publisher、
Strategy、 Logger等</Test>
</Peach>
```

其主要元素包括：

- DataModel

一个 Pit 文件至少会包括一个或多个 DataModel, 描述数据类型信息, 关系（大小、

数量、偏移量），和其它允许智能 Fuzz 的信息。如下图：

```
<DataModel name="Template">
  <String name="Key" />
  <String value=": " token="true" />
  <String name="Value" />
  <String value="\r\n" token="true" />
</DataModel>

<DataModel name="Customized" ref="Template">
  <String name="Key" value="Content-Length" />
  <String name="Value">
    <Relation type="size" of="HttpBody" />
  </String>
  <Blob name="HttpBody" />
</DataModel>
```

其属性包括：

（1）name：数据模型的名字 [必须]。

（2）ref: 引用模版数据模型 [可选]。DataModel 有 ref 属性时，与被引用 DataModel 类似于子类与基类的关系，基类数据会被子类继承，子类子元素会覆盖基类同名子元素，

（3）mutable：数据元素可变异性 [可选，默认 true]。

其主要子元素：Blob、Block、Choice、Flags、String、Number、Relation 等。

1)Blob：常用于表示没有类型定义和格式的数据，如下图：

```
<Blob name="Unknown1" valueType="hex" value="01 06 22 03"/>
```

其主要属性包括：

•value：Blob 默认值。

•length：blob 的字节长度，blob 长度判断会根据后续有 token 元素的位置计算。

•token：这个元素解析是否作为 “标记”，默认 false。

2)Block：用来组合一个或者多个的其他元素。Block 和 DataModel 是很类似的，一个重要区别在于它们的位置，DataModel 是顶级元素，Block 是其子元素。

其不同于 DataModel 的属性包括：

•minOccurs：这个 Block 所必须出现的最低次数 [可选]。

•maxOccurs：这个 Block 可能会出现的最高次数 [可选]。

3)Choice: 每次选择其中一个元素，类似 switch 语句。如下图:

```
<DataModel name="ChoiceExample1">
 <Choice name="Choice1" minOccurs="3" maxOccurs="6">

  <Block name="Type1">
    <Number name="Str1" size="8" value="1" token="true" />
    <Number size="32"/>
  </Block>

  <Block name="Type2">
    <Number name="Str2" size="8" value="2" token="true" />
    <Blob length="255" />
  </Block>

  <Block name="Type3">
    <Number name="Str3" size="8" value="3" token="true" />
    <String length="8" />
  </Block>
 </Choice>
</DataModel>
```

minOccurs 为最小生成 Choice 数；maxOccurs 为最大生成 Choice 数，-1 为无上限；occurs 为必须产生的次数，如果不能达到这个次数，异常退出。具体匹配实现按照 Choice 中 Block 顺序，crack（解析）数据时根据 token 匹配一个 Block 后，数据位置后移匹配 Block 大小，继续按照 Choice 中 Block 顺序从头匹配。

4)Flags: Flag 元素定义包含在 Flags 容器中的位字段，如下图:

```
<Flags name="options" size="16">
  <Flag name="compression" position="0" size="1" />
  <Flag name="compressionType" position="1" size="3" />
  <Flag name="opcode" position="10" size="2" value="5" />
</Flags>
```

其主要属性包括 :

•size : 大小，以位数为单位 [必须] 。

•position : flag 的起始位置 (以 0 为基准）[必须]。

5)String : 定义一个或者双字节的字符串，如下图 :

```
<String value="Null terminated string" nullTerminated="true" />
```

其主要属性包括 :

•nullTerminated : 字符串是以 null 结尾 [可选] 。

•type : 字符编码类型，默认 “ascii”，可用选项有 ascii, utf7, utf8, utf16, utf16be, utf32 [可选]。

•padCharacter : 填充字符串，来填充达到 length 的长度，默认是 0x00[可选]。

6)Number : 定义了长度为 8，16，24，32 或者 64 位的二进制数字，如下图 :

```
<DataModel name="NumberExample7">
 <Number name="Hi5" value="AB CD" valueType="hex" size="16" signed="false" endian="little" />
</DataModel>
```

其主要属性包括 :

•size: Number 的大小, 以位为单位。有效的选择是 1-64 [可选]。

•endian : 数字的字节顺序，默认是小端字节 [可选] 。

•signed : 是否是有符号，默认是 true[可选]。

7)Relation: 用于连接两个大小、数据、偏移量相关元素，如下图:

```
<Number size="32" signed="false">
  <Relation type="size" of="Value"
  expressionGet="size/2" expressionSet="size*2" />
</Number>
<String name="Value" />
```

type 类型为 size 时，of 表示 Number 是 Value 字符串的字节数。expressionGet 用于 crack 过程，表示读 “Value” 多少字节。expressionSet 用于 publishing 过程, 为 Publisher 生成 Number 值。

StateModel

用于定义测试的逻辑，实际上相当于一个状态机。如下图 :

```
<!-- This is our simple wave state model -->
<StateModel name="TheState" initialState="Initial">
      <State name="Initial">
            <!-- Write out our wave file -->
            <Action type="output">
                  <DataModel ref="Wav"/>
                  <!-- This is our sample file to read in -->
                  <Data fileName="sample.wav"/>
            </Action>
            <Action type="close"/>
            <!-- Launch the target process -->
            <Action type="call" method="StartMPlayer" publisher="Peach.Agent" />
      </State>
</StateModel>
```

下级标签包括 State，每个 State 中又可以包含若干个 Action 标签。

1)State : 表示一个状态，不同的 State 之间可以根据一些判断条件进行跳转, 通常和 Action 的 when 属性联合使用。如 71 页左上图。

2)Action : 用于完成 StateModel 中的各种操作，是给 Publisher 发送命令的主要方式。Action 能发送输出、接收输入、打

```
<DataModel name="InputModel">
        <Number name="Type" size="32" />
</DataModel>
<DataModel name="OutputModelA">
        <Number name="Type" size="32" value="11 22 33 44" valueType="hex"
</DataModel>
<StateModel name="StateModel" initialState="InitialState">
        <State name="InitialState">
                <Action type="input">
                        <DataModel ref="InputModel" />
                </Action>
                <Action type="changeState" ref="State2" when="int
                (StateModel.states['InitialState']
                .actions[0].dataModel['Type'].InternalValue) == 2"/>
        </State>
        <State name="State2">
                <Action type="output">
                        <DataModel ref="OutputModelA" />
                </Action>
        </State>
</TheStateModel>
```

开连接，也能改变 State 等。主要属性：

•type：操作类型 [必须]。

主要类型：

start：启动 Publisher，隐含动作，一般不需要。

stop：停止 Publisher，隐含动作，一般不需要。

input：接收或者读取来自 Publisher 的输入，需要指定 DataModel，用于 crack 和包含输入数据。

ouput：通过 Publisher 发送或者写输出，需要一个 DataModel，包含可选 data，如下图：

```
<!-- Write out our wave file -->
<Action type="output">
        <DataModel ref="Wav"/>
        <!-- This is our sample file to read in -->
        <Data fileName="sample.wav"/>
</Action>
```

•when：如果提供的表达式为 true，完成操作；否则，跳过。

•ref：状态变更后的引用 [type=changeState] 。

•method：call 的方法 [必须 , type=call]，调用 Publisher 可选参数定义的方法，不是所有 Publisher 都支持。

Agent

是能够运行在本地或者远程的特殊的 Peach 进程，这些进程能够启动监视器监控被测目标，如附加调试器、检测 crash 等。如下图：

```
<Agent name="LocalAgent">
 <Monitor class="WindowsDebugger">
     <!-- The command line to run.  Notice the filename provided matched up
          to what is provided below in the Publisher configuration -->
     <Param name="CommandLine" value="C:\mplayer\mplayer.exe fuzzed.wav" />
     <!-- This parameter will cause the debugger to wait for an action-call in
          the state model with a method="StartMPlayer" before running
          program.
     -->
     <Param name="StartOnCall" value="StartMPlayer" />
 </Monitor>
 <!-- Enable heap debugging on our process as well. -->
 <!--Monitor class="PageHeap">
     <Param name="Executable" value="hmplayer.exe"/>
 </Monitor-->
</Agent>
```

远程 Agent 首先需要在远程目标机通过 peach –a tcp 启动远程代理，无需 pit 文件。本地 Peach pit 文件添加如下图 location，其中 ip 为目标机 ip。

```
<Agent name="RemoteAgent" location="tcp://192.168.1.1:9001">
  <!-- Monitors -->
</Agent>
```

可用 Monitor 如下图：

Windows Monitors
- Windows Debugger Monitor
- Cleanup Registry Monitor
- Page Heap Monitor
- Popup Watcher Monitor
- Windows Service Monitor

OS X Monitors
- OS X Crash Wrangler Monitor
- OS X Crash Reporter Monitor

Linux Monitors
- Linux Crash Monitor

Cross Platform Monitors
- CanaKit Relay Monitor
- Cleanup Folder Monitor
- IpPower9258 Monitor
- Memory Monitor
- Pcap Network Monitor
- Ping Monitor
- Process Launcher Monitor
- Process Killer Monitor
- Save File Monitor
- Socket Listener Monitor
- SSH Monitor
- SSH Downloader Monitor
- Vmware Control Monitor

Windows Debugger Monitor 通过 windbg 控制一个 windows 调试实例，主要参数：

•CommandLine：运行的命令行，如下图：

```
<Param name="CommandLine" value="c:\\mplayer\\mplayer.exe fuzzed.wav" />
```

文件 fuzz 时上述文件名 fuzzed.wav 需要与 Publisher 参数一致。如下图：

```
<Publisher class="File">
    <Param name="FileName" value="fuzzed.wav"/>
</Publisher>
```

•SymbolsPath：windbg 符号路径。

•StartOnCall：StateModel 有匹配调用时附加调试器。

•NoCpuKill：默认 false，表示当被测目标进程 cpu 占用为 0 时将其结束。

Peach3 对非内核目标使用的混合调试模式，首先通过 CreateProcess DEBUG_PROCESS 参数创建调试进程，当检测到被测目标有感兴趣 faults 产生时会使用 windbg 的 dbgeng.dll 进行重现调试，最后利用 windbg 插件 msec.dll 的 !exploitable 命令对漏洞的可利用性进行初步判断，记录结果。

Test

指定使用哪个 Agent、StateModel，Publisher 用什么方法发送数据，使用什么方法变异数据，日志文件路径等。可以有多个 Test，使用时通过 Peach 命令行指定要运行的 Test 名称，未指定默认运行名称为 “Default” 的 Test。如下图：

```
<Test name="Default" waitTime="5">
  <Strategy class="RandomDeterministic"/>
  <Agent ref="LocalAgent"/>
  <StateModel ref="TheState"/>
  <Publisher class="File">
      <Param name="FileName" value="fuzzed.wav"/>
  </Publisher>
  <Logger class="Filesystem">
      <Param name="Path" value="logs" />
  </Logger>
</Test>
```

Strategy（变异策略）包括：

•Random：默认会随机选择最大 6 个元素（可以通过参数 MaxFieldsToMutate 设置）利用随机 mutator（变异器）进行变异。

•Sequential：Peach 会对每个元素使用其所有可用的 Mutators 进行变异。

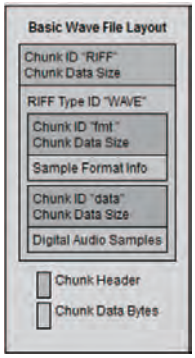
•RandomDeterministic：Peach 默认规则。这个规则对 pit xml 文件中元素根据 Mutators 生成的 Iterations 链表做相对随机（由链表中元素数目决定）的顺序混淆，所以每个 xml 文件每次运行生成的测试用例多少、顺序固定，这样才能保证 skipto 的准确性。

Peach3 包括元素增、删、改、交换，经验值，逐位、双字等 Mutators，见 73 页左上角图。

3.3 Fuzz Wav 文件

- ArrayNumericalEdgeCasesMutator
- ArrayRandomizeOrderMutator
- ArrayReverseOrderMutator
- ArrayVarianceMutator
- BlobBitFlipperMutator
- BlobDWORDSliderMutator
- BlobMutator
- DataElementDuplicateMutator
- DataElementRemoveMutator
- DataElementSwapNearNodesMutator
- FiniteRandomNumbersMutator
- NumericalEdgeCaseMutator
- NumericalVarianceMutator
- SizedDataNumericalEdgeCasesMutator
- SizedDataVarianceMutator
- SizedNumericalEdgeCasesMutator
- SizedVarianceMutator
- StringCaseMutator
- StringMutator
- UnicodeBadUtf8Mutator
- UnicodeBomMutator
- UnicodeStringsMutator
- UnicodeUtf8ThreeCharMutator
- ValidValuesMutator
- WordListMutator
- XmlW3CMutator

▸ Wav 文件格式



▸ Pit 文件

```
<?xml version="1.0" encoding="utf-8"?>
<Peach xmlns="http://phed.org/2012/Peach" xmlns:xsi="http://www.w3.org
  xsi:schemaLocation="http://phed.org/2012/Peach ../peach.xsd">

  <!-- Defines the common wave chunk -->
  <DataModel name="Chunk">
        <String name="ID" length="4" padCharacter=" " />
        <Number name="Size" size="32" >
                <Relation type="size" of="Data"/>
        </Number>
        <Blob name="Data" />
  </DataModel>

  <DataModel name="ChunkFmt" ref="Chunk">
        <String name="ID" value="fmt " token="true"/>
        <Block name="Data">
                <Number name="CompressionCode" size="16" />
                <Number name="NumberOfChannels" size="16" />
                <Number name="SampleRate" size="32" />
                <Number name="AverageBytesPerSecond" size="32" />
                <Number name="BlockAlign" size="16" />
                <Number name="SignificantBitsPerSample" size="16" />
                <Number name="ExtraFormatBytes" size="16" />
                <Blob name="ExtraData" />
        </Block>
  </DataModel>
```

```
省略ChunkData等其它DataModel

<!-- Defines the format of a WAV file -->
<DataModel name="Wav">
      <!-- wave header -->
      <String value="RIFF" token="true" />
      <Number size="32" />
      <String value="WAVE" token="true" />

      <Choice maxOccurs="30000">
            <Block ref="ChunkFmt"/>
            <Block ref="ChunkData"/>
            <Block ref="ChunkFact"/>
            <Block ref="ChunkSint"/>
            <Block ref="ChunkWavl"/>
            <Block ref="ChunkCue"/>
            <Block ref="ChunkPlst"/>
            <Block ref="ChunkLtxt"/>
            <Block ref="ChunkSmpl"/>
            <Block ref="ChunkInst"/>
            <Block ref="Chunk"/>
      </Choice>
</DataModel>
```

```
<!-- This is our simple wave state model -->
<StateModel name="TheState" initialState="Initial">
      <State name="Initial">
            <!-- Write out our wave file -->
            <Action type="output">
                  <DataModel ref="Wav"/>
                  <!-- This is our sample file to read in -->
                  <Data fileName="sample.wav"/>
            </Action>
            <Action type="close"/>
            <!-- Launch the target process -->
            <Action type="call" method="StartMPlayer" publisher="Peach.Agent" />
      </State>
</StateModel>

<Agent name="LocalAgent">
<!--Agent name="LocalAgent"  location="tcp://192.168.126.175:9001"-->
 <Monitor class="WindowsDebugger">
      <!-- The command line to run.  Notice the filename provided matched up
           to what is provided below in the Publisher configuration -->
      <Param name="CommandLine" value="C:\Program Files\Haihaisoft
        Universal Player\hmplayer.exe fuzzed.wav" />
      <!-- This parameter will cause the debugger to wait for an action=call in
           the state model with a method="StartMPlayer" before running
           program.
      -->


      <Param name="StartOnCall" value="StartMPlayer" />
  </Monitor>
</Agent>

<Test name="Default" waitTime="3">
  <Agent ref="LocalAgent"/>
  <StateModel ref="TheState"/>
  <Publisher class="File">
      <Param name="FileName" value="fuzzed.wav"/>
  </Publisher>
  <Logger class="Filesystem">
      <Param name="Path" value="logs" />
  </Logger>
</Test>

</Peach>
```

参考文献

www.peachFuzzer.com