Peach Fuzzer 3 官方文档

中文版

Xiao 白鸽
2017 年 3 月 31 日

目录

1 Peach 是什么

Peach 是一款智能模糊测试器（SmartFuzzer），它有两种模糊测试能力：基于生长的模糊测试和基于变异的模糊测试。Peach 需要创建一些 PeachPits 文件，该文件定义了被模糊测试数据中的结构、类型信息和相互关系。此外，它允许为进行中的模糊测试进行配置，包含选择一个数据通道（Publisher）和登录接口等。Peach 由 Deja vu Security 公司的 Michael Eddington 创造并开发，主要开发工作已经有 7 年了，主要有 3 个版本。

1.1 Peach 的历史

Peach V 1.0 是一个由 Python 框架创造的 fuzzer，于 2004 年完成开发，并公布于 ph-neutral 0x7d4 网站上。Peach 2.0 发布于 2007 年夏天，是第一款综合的开源 fuzzing 工具，包含进程监视和创建 fuzzer，其中创建 fuzzer 由 XML 语言实现。Peach 3 发布于 2013 年初，是一个全部重写的版本，它放弃了 Python 语言，改由 Microsoft.NET 框架来实现，主要是 C#。Mono 开源运行时促进了 Peach 的跨平台支持。

Peach 不是开源软件，而是遵循 MIT 许可证的免费软件。和 BSD 许可证一样，MIT 许可证在 Peach 的使用和修改上没有限制。

2 安装

2.1 安装 Peach 3

以下章节列出了在不同操作系统上安装 Peach 所需的步骤。Peach 3 利用 Microsoft.NET 和 C#语言对 Peach 2 进行了完全的重写。和 Peach 2 相比，新版本的可利用特性非常相似，主要变化在于引擎。初步测试显示，Peach 3 比 Peach 2.3 快 3 至 4 倍。

2.2 安装二进制发行版

Windows：

1、安装 Microsoft.NET 4 运行时。

2、安装 windows 调试工具（Windbg）。

3、将 Peach 二进制发行版解压到你个工作文件夹。

4、现在准备开始使用 Peach 3！

在 fuzzing 网络协议时如果你想让网络捕获可用，请安装 WireShark 或者 Winpcap。

OS X：

1、安装最新版的 Mono 包。

2、安装 Crash Wrangler。

3、将 Peach 二进制发行版解压到你个工作文件夹。

4、现在准备开始使用 Peach 3！

Linux：

1、安装最新的 Mono 包：Ubuntu/Debian 安装 mono 完整包；SUSE 安装请看 Mono 官网说明。

2、将 Peach 二进制发行版解压到你个工作文件夹。

3、现在准备开始使用 Peach 3！

2.3 由源码构建 Peach 3

Windows：

1、安装 Microsoft.NET 4 SDK。

2、安装 Visual Studio 2010 SP1。

3、安装最新版本的 Python 2。

4、将 Peach 源码解压到工作文件夹，或者用 GIT 工具下载最新的源码。

5、运行“waf.bat 配置”。

6、运行“waf.bat 安装”。

现在你将会看到一个包含有二进制文件的“output”文件夹。

OS X & Linux：

1、安装带编译器的 Mono。

2、将 Peach 源码解压到工作文件夹，或者用 GIT 工具下载最新的源码。

3、运行“waf.bat 配置”。

4、运行“waf.bat 安装”。

现在你将会看到一个包含有二进制文件的“output”文件夹。

3 教程

3.1 Peach 3 快速开始

以下教程为如何利用 Peach 3 开始 fuzzing 提供了信息。

- Dumb Fuzzing
- File Fuzzing

3.2 Dumb Fuzzing

欢迎来到 dumb fuzzing 教程。在这个教程中，我们将为 PNG 图像文件（.png）构建一个简单的 dumb fuzzer。利用位翻转和双字节移位等方法，dumb fuzzer 将会用一些样本文件（也叫种子文件）进行变异。 此次 fuzzer 的目标是 windows 系统中的 mspaint、linux 系统中的 feh 和 OSX 系统中的 Safari。主要有以下几个步骤：

3.2.1 开发环境

在开始构建我们自己的 Peach Fuzzer 之前，先花一些时间来讨论一下正常的 Peach 开发

环境。一个典型的开发环境如下所示：

（1）XML 编辑器

一个好的 XML 编辑器绝对是必须的。Windows 上一个好的免费 XML 编辑器是 Microsoft Visual Studio Express 编辑器。如果你已经拥有了一款智能感知 XML 编辑，那样最好。

（2）最新版 Peach

经常保持你使用的 Peach 版本为最新的。

（3）调试器和支持的工具

为了更好的进行 fuzzing 工作，需要以下工具的支持。Windows 系统需要 Windbg 支持（最新可利用版本）；OS X 系统需要 CrashWrangler 支持（需要苹果开发者账号）。

3.2.2 创建数据模型

首先复制一个 template.xml 文件的副本，并重命名为 png.xml，template.xml 位于你的 Peach 文件夹中。这个文件保存着关于 PNG dumb Fuzzer 的所有信息。同样地，你也需要一些 PNG 样本（至少 10 个）。对于我们的 dumb fuzzer，仅仅只需要一个数据模型，它保存着 PNG 文件的所有数据。这个数据模型并不知道 PNG 数据结构的任何信息，而是将所有的数据保持在一个“Blob”元素（二进制大对象或者字节数组）中。

创建 DataModel 元素

现在，用我们的 XML 编辑器打开 png.xml，开始编辑它。定位到“DataModel”的位置，如下图所示：

```
<!-- TODO: Create data model -->
<DataModel name="TheDataModel">
</DataModel>
```

想了解更多有关内容，请阅读 DataModel 章节。

在我们的数据模型中，将增加一个单独的数据元素，如下图所示：

```
<DataModel name="TheDataModel">
    <Blob />
</DataModel>
```

想了解更多有关内容，请阅读 DataModel 章节和 Blob 章节。

到现在为止，这就是我们的数据模型所需的所有东西，“Blob”元素最终将保持所有的 PNG 数据。

3.2.3 创建状态模型

现在已经创建好了数据模型，接下来创建状态模型。对于文件 fuzzing 来说，状态模型非常简单。所有需要我们做的仅仅只是写文件和启动目标进程，我们将使用三个 Action 元素来实现这些功能，如下所示：

- Output---------写文件
- Close---------关闭文件
- Call---------启动应用程序

在 png.xml 文件中，定位到 StateModel（名字为 TheState），展开这个元素，并添加我

们所需的三个 Action，如下所示：

```
<!-- This is our simple png state model -->
<StateModel name="TheState" initialState="Initial">
        <State name="Initial">

                <!-- Write out our png file -->
                <Action type="output">
                        <DataModel ref="TheDataModel"/>

                        <!-- This is our folder of sample files to read in -->
                        <Data name="data" fileName="samples_png/*.png"/>
                </Action>

                <Action type="close"/>

                <!-- Launch the target process -->
                <Action type="call" method="LaunchViewer" publisher="Peach.Agent"/>
        </State>
</StateModel>
```

更多有关内容请阅读 StateModel、State、Action、DataModel、Data 等章节。

请注意，最后的“call”动作，有一个“publisher”属性，它的值为“Peach.Agent”。它的作用是给任何一个被配置的代理发送一个含有“LaunchViewer”的调用消息，这就是为什么调试监视器知道如何启动进程的原因。

非常好，状态模型已经全部配置好了，接下来需要做的是配置调试器和 Publisher。

3.2.4 配置 Publisher

进行 fuzzing 之前的最后一步是配置一个 Publisher。Publishers 是 Peach 的 I/O 连接，它是实现输出、输入和调用等操作之间的管道。对于文件 fuzzer 来说，将使用一个称之为文件的 Publisher，它允许我们对一个文件进行写操作。配置一个 Publisher 是很容易的，它位于 png.xml 的底部附近，是 Test 标签的一个子标签。现在，这个 Publisher 有一个名字为 FileName 的唯一参数，该参数的值是被 fuzzer 文件的文件名，如下图所示：

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

接下来，我们将配置一种方式来探测目标程序什么时候崩溃。同样地，也要收集一些便于以后查看的相关信息，比如堆栈踪迹等等。下一节，将学习如何配置代理和监视器。

3.2.5 添加代理和监视器

（1）代理和监视器

现在开始配置代理和监视器。代理是特殊的 Peach 进程，它能够运行在本地进程中，也可以通过网络在远程运行。这些代理拥有一个或者多个监视器，用来执行调试器附加和查看内存消耗等之类的操作。本教程将为每个目标平台配置一些特定的 Peach 监视器。Windows 将配置 Windbg 调试器和 Heap 调试器，Windbg 用来监视 mspaint.exe 程序的异常和其他常

见问题， Heap 调试器用来调试目标进程；Linux 将配置相应的调试器来监视系统中的核心文件；OSX 将配置 CrashWangler 来监视 Safari 程序的异常和其他常见问题。

（2）配置代理和监视器

首先，要找到样本文件中的 Agent 元素，一般来说，该元素会有一些相关的注释，其大致情况如下图所示：

```
<!-- TODO: Configure agent -->
<Agent name="TheAgent" location="http://127.0.0.1:9000"/>
```

接下来，取消这部分的注释，并删除“location”属性。当没有“location”属性时，Peach 会自动地开启一个本地 Peach 代理。本教程将配置三个代理，分别对应于 Windows、Linux 和 OSX 操作系统。Windows 代理由 Windows Debugger 和 PageHeap 组成；Linux 代理由 LinuxDebugger 组成；OSX 代理由 CrashWrangler 组成，详细情况如下图所示：

```
<Agent name="WinAgent">
   <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="mspaint.exe fuzzed.png" />

        <!-- This parameter will cause the debugger to wait for an action-call in
             the state model with a method="LaunchViewer" before running
             program.
        -->
        <Param name="StartOnCall" value="LaunchViewer" />

        <!-- This parameter will cause the monitor to terminate the process
             once the CPU usage reaches zero.
        -->
        <Param name="CpuKill" value="true"/>

    </Monitor>

    <!-- Enable heap debugging on our process as well. -->
    <Monitor class="PageHeap">
        <Param name="Executable" value="mspaint.exe"/>
    </Monitor>

</Agent>

<Agent name="LinAgent">
    <!-- Register for core file notifications. -->
    <Monitor class="LinuxDebugger"/>

        <!-- This is the program we're going to run inside of the debugger -->
        <Param name="Executable" value="feh"/>

        <!-- These are arguments to the executable we want to run -->
        <Param name="Arguments"  value="fuzzed.png"/>

        <!-- This parameter will cause the monitor to terminate the process
             once the CPU usage reaches zero.
        -->
        <Param name="CpuKill" value="true"/>

</Agent>
```

```
<Agent name="OsxAgent">
    <Monitor class="CrashWrangler">
        <!-- The executable to run. -->
        <Param name="Command" value="/Applications/Safari.app/Contents/MacOS/Safari" />

        <!-- The program arguments.  Notice the filename provided matched up
             to what is provided below in the Publisher configuration -->
        <Param name="Arguments" value="fuzzed.png" />

        <!-- Do not use debug malloc. -->
        <Param name="UseDebugMalloc" value="false" />

        <!-- Treat read access violations as exploitable. -->
        <Param name="ExploitableReads" value="true" />

        <!-- Path to Crash Wrangler Execution Handler program. -->
        <Param name="ExecHandler" value="/usr/local/bin/exc_handler" />

        <!-- This parameter will cause the monitor to wait for an action-call in
             the state model with a method="LaunchViewer" before running
             program.
        -->
        <Param name="StartOnCall" value="LaunchViewer" />

    </Monitor>
</Agent>
```

更多详细内容请阅读 Agent 章节、WindowsDebugger 章节和 PageHeap 章节。

（3）配置 Test 元素

接下来，需要启用我们刚刚配置的代理。找到 Test 元素，取消该行的注释，并修改我们的 Launcher publisher,如下图所示：

```
<Test name="Default">
    <Agent ref="WinAgent" platform="windows"/>
    <Agent ref="LinAgent" platform="linux"/>
    <Agent ref="OsxAgent" platform="osx"/>

    <StateModel ref="TheState"/>

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

更多详细内容请阅读 Test 章节。

（4）配置 Fuzzing 策略

由于要对许多文件进行 dumb fuzzing，因此要更改 Peach 使用的默认 fuzzing 策略，以便更能满足本次实验的需求。最好的 dumb fuzzing 策略是随机策略，可以通过在 Test 元素中添加 Strategy 元素的方式来配置它，如下图所示：

```
<Test name="Default">
    <Agent ref="WinAgent" platform="windows"/>
    <Agent ref="LinAgent" platform="linux"/>
    <Agent ref="OsxAgent" platform="osx"/>

    <StateModel ref="TheState"/>

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

    <Strategy class="Random"/>
</Test>
```

（5）配置记录

现在，监视器正在探测程序的各种错误，因此，还需要一个信息记录机制来捕获 fuzzer 运行的结果，在 XML 文件底部的 Test 元素中添加 logger 元素就可以实现该机制。如下所示：

```
<Test name="Default">
    <Agent ref="WinAgent" platform="windows"/>
    <Agent ref="LinAgent" platform="linux"/>
    <Agent ref="OsxAgent" platform="osx"/>

    <StateModel ref="TheState"/>

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

    <Strategy class="Random"/>

    <Logger class="Filesystem">
        <Param name="Path" value="logs" />
    </Logger>
</Test>
```

更多详细内容请阅读 Test 章节、Logger 章节和 File Logger 章节。

（6）运行 Fuzzer

现在让我开始正真的 fuzzer 之旅，每隔大约 200 次迭代，策略将切换一个不同的样本文件。在命令行中，运行“peach png.xml”开始 fuzzing。

（7）下一步做什么

Fuzzing 开始之后，我们下一步需要做以下几个工作：

- 收集额外的样本文件。
- 观察运行状态，删除导致重复代码路径的任何文件。
- 收集程序的 bug 信息

3.3 File Fuzzing

欢迎来到 File fuzzing 教程，在该教程中我们将构建一个 wave 文件（.wav）的模糊器。Wave 文件基于 RIFF 文件格式，这种格式并不十分复杂，而且能够展示出 Peach 的几个特性。本次 fuzzer 的目标是 mplayer，它是一款开源、跨平台和命令行形式的多媒体软件。

3.3.1 开发环境

在开始构建我们自己的 Peach Fuzzer 之前，先花一些时间来讨论一下正常的 Peach 开发环境。一个典型的开发环境如下所示：

（1）XML 编辑器

一个好的 XML 编辑器绝对是必须的。Windows 上一个好的免费 XML 编辑器是 Microsoft Visual Studio Express 编辑器。如果你已经拥有了一款智能感知 XML 编辑，那样最好。

（2）最新版 Peach

经常保持你使用的 Peach 版本为最新的。

（3）调试器和支持的工具

为了更好的进行 fuzzing 工作，需要以下工具的支持。Windows 系统需要 Windbg 支持（最新可利用版本）；OS X 系统需要 CrashWrangler 支持（需要苹果开发者账号）。

3.3.2 创建数据模型

首先复制一个 template.xml 的副本（在你的 Peach 文件夹中），并重命名为 wav.xml，它保存着 WAV 模糊器的所有信息。同样地，你需要一个 WAV 样本，该网址可以获得：http://www-mmsp.ece.mcgill.ca/documents/AudioFormats/WAVE/Samples/AFsp/M1F1-int32WE-AFsp.wav。

现在，你需要查看以下两个规范来了解一下 WAV 格式：

- http://www.sonicspot.com/guide/wavefiles.html
- RIFF 文件规格说明（微软）。

如果曾经留意过 WAV 文件格式，你会发现 WAV 文件由一个文件头组成，其后跟着很多的 Chunk，这种形式对于文件格式和数据包来说是非常普遍的。典型地来说，每个 Chunk 的格式相同，它遵循某种形式的 T-L-V 或类型长度值。事实上，WAV 文件 Chunk 也是这样，类型后紧跟着是长度，长度后紧跟着是数据。每个 Chunk 类型定义了其后紧跟的数据的含义。

基于以上信息,就能构建出所需的模糊器，它有几个顶层的”“DataModel”元素，名字分别为：Chunk、ChunkFmt、ChunkData、ChunkFact、ChunkCue、ChunkPlst、ChunkLabl、ChunkLtxt、ChunkNote、ChunkSmpl、ChunkInst、Wav。名为 Chunk 的“DataModel”元素是以下每种类型 Chunk 的模板，我们将它们放在一起。名为 Wav 的“DataModel”用来定义文件头。

（1）Number 元素的默认设置

WAV 中用到的大多数字是无符号型的。在 PIT 文件中添加 XML 元素来设置默认值。如下图所示：

```
<Defaults>
    <Number signed="false" />
</Defaults>
```

（2）创建 Wav DataModel

回到 wav.xml 文件，开始编辑相关的 XML 元素。找到名字为“TheDataModel”的 DataModel

元素，如下图所示。更多该元素的细节请阅读 DataModel 章节。

```
<!-- TODO: Create data model -->
<DataModel name="TheDataModel">
</DataModel>
```

将该元素的名字重命名为 Wav。根据 wav 文件规范，在 Peach 文件中定义文件头如下图所说。其中，wav 文件的文件头格式为：

- 文件魔术数：4 字符串，一般为“RIFF”。
- 文件长度：32 位无符号整数。
- RIFF 类型：4 字符串，一般为“WAVE”。

```
<!-- 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"/>
</DataModel>
```

（3）Chunk DataModel

Chunk DataModel 是 Peach Pit 文件中的第一个 DataModel，所以将它放在 Wav DataModel 之前，如下图所示。更多详细内容请阅读 DataModel 章节、String 章节和 Number 章节。

```
<!-- Defines the common wave chunk -->
<DataModel name="Chunk">
</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"/>
</DataModel>
```

Chunk DataModel 在 Wav DataModel 之前，这是非常重要的，稍后我们会引用这个 DataModel，而且使用它之前必须先定义。根据 wav 文件规范，Chunk 的格式如下：

- ID：4 字符串，用空格填充。
- Size：4 字节无符号整数。
- 数据：字节数据，大小取决于 Size 参数的值

```
<!-- 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" />
    <Padding alignment="16" />
</DataModel>
```

因此，我们可以进行相应的定义，如上图所示。 更多详细内容请阅读 DataModel 章节、String 章节、Number 章节、Relation 章节、Blob 章节和 Padding 章节。

现在，我们已经创建了 Size 参数和 Data 参数之间的大小关系，这样 Size 参数自动更新时，会产生相应大小的 Data 参数。当用一个样本文件作为默认值进行解析时，解析器根据 Size 参数的值就能找到相应的 Data 部分。现在我们可以用一个 Padding 类型来正常地填充自己的 DataModel。注意，alignment 属性的值为 16，这说明 Padding 元素能够自己调整大小，Chunk DataModel 以 16 位（2 字节）的边界结束。

（4）Format Chunk

现在来定义 Format Chunk 的细节。使用已经定义过的通用 Chunk 作为模板，只需要指定 Format Chunk 的细节和保存一些版式。根据 wave 规范，Format Chunk 的格式如下：

- ID:通常为“fmt”。
- Data：
  - 压缩代码：16 位无符号整数。
  - 声道数目：16 位无符号整数。
  - 采样率：32 位无符号整数。
  - 平均每秒所需字节数：32 位无符号整数。
  - 块对齐单位：16 位无符号整数。
  - 每个采样所需的位数：16 位无符号整数。
  - 附加信息：16 位无符号整数。

ChunkFmt DataModel 位于 Chunk DataModel 之后和 Wav DataModel 之前，如下图所示。更多详细内容请阅读 DataModel 章节、Block 章节、String 章节、Number 章节和 Blob 章节。

```
<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>
```

在这里，你会发现一些非常酷的事情。首先，查看 DataModel 元素，你会发现一个名字为 ref 的属性，它的值是 Chunk，这是告诉 Peach 复制 Chunk DataModel，并以它为基础来产生新的 ChunkFmt DataModel，Chunk DataModel 中的所有元素都默认出现在新的 ChunkFmt DataModel 中，这说明 Peach 中的 Chunk 可以重用。其次，还发现 ChunkFmt DataModel 中有两个元素和 Chunk DataModel 中的两个元素名字相同，分别为 ID 和 Data，这会让新元素替代旧元素。这就允许我们根据新格式 Chunk 类型的需要，对旧元素进行重写。

现在，你也许会问为什么要对元素 ID 进行重写？这是个不错的问题，重写元素 ID 是为了指定新格式 Chunk 需要的静态字符，稍后我们将会指定一个样本 wav 文件来使用，解析器需要提示怎样选择正确的 chunk。更多细节会在后面的 Choice 元素部分来介绍。

（5）Data Chunk

接下来是 Data Chunk。这个很容易，因为数据包的数据部分没有结构。我们可以这样定

义这个 Chunk，如下图所示。更多详细内容请阅读 DataModel 章节和 String 章节。

```
<DataModel name="ChunkData" ref="Chunk">
    <String name="ID" value="data" token="true"/>
</DataModel>
```

（6）Fact Chunk

好了，现在我们来看 Fact Chunk。这个块的格式如下：

- ID：“fact”，字符串，4 个字符。
- Data：
  - 样本数目：32 位无符号整数。
  - 未知？：未知字节。

又一个非常容易定义的块，如下图所示：

```
<DataModel name="ChunkFact" ref="Chunk">
    <String name="ID" value="fact" token="true"/>
    <Block name="Data">
        <Number size="32" />
        <Blob/>
    </Block>
</DataModel>
```

更多详细内容请阅读 DataModel 章节、Block 章节、String 章节、Number 章节和 Blob 章节。请注意，我比较懒，没有为 Number 元素和 Blob 元素取名字，Peach 不需要所有的元素都有名字，只要被引用的元素有名字就行。

（6）Wave List Chunk

Wave List Chunk 有些不同，它由一个列表里的 silent chunk 和 data chunk 交替组成。因此，在完成 wave list chunk 之前，我们需要先定义 silent chunk。Silent chunk 非常简单，它仅仅是一个 4 字节的无符号整数，数据模型定义如下：。更多详细内容请阅读 DataModel 章节、Block 章节、String 章节和 Number 章节。

```
<DataModel name="ChunkSint" ref="Chunk">
    <String name="ID" value="sInt" token="true"/>
    <Block name="Data">
        <Number size="32" />
    </Block>
</DataModel>
```

这就是我们获得 wave list chunk 的方式。Data 部分是一个由 silent chunk 和 data chunks 构成的列表。下图显示了我们是如何做的。更多详细内容请阅读 DataModel 章节、Block 章节和 String 章节。

```
<DataModel name="ChunkWavl" ref="Chunk">
    <String name="ID" value="wavl" token="true"/>
    <Block name="Data">
        <Block name="ArrayOfChunks" maxOccurs="3000">
            <Block ref="ChunkSint"/>
            <Block ref="ChunkData" />
        </Block>
    </Block>
</DataModel>
```

该定义介绍了列表或者重复元素的概念。请注意，block 元素有个 maxOccurs 属性。它告诉 Peach 该 block 可能发生至少 1 次，最多 3000 次变化。同时也要注意到，我们使用了 block 元素的 ref 属性，它和我们重用数据模型内部数据的原理是一样的，不是很难理解。

（7）Cue Chunk

这个块比较简单，也是一个列表。由以下几部分组成：

- ID：4 字节。
- 位置：4 字节无符号整数。
- 数据 Chunk ID：4 字节 RIFFID。
- Chunk 开始：4 字节无符号整数的数据块偏移。
- Block 开始：4 字节的无符号整数，偏移到第一个声道的采样。
- 采样偏移：4 字节无符号整数，偏移到第一张声道的采样字节。

不用担心最后 3 个数字是偏移的事实。这个数据已经在 wave list chunk 中解析，只需要读取它们就行了。定义如下：

```
<DataModel name="ChunkCue" ref="Chunk">
    <String name="ID" value="cue " token="true"/>
    <Block name="Data">
        <Block name="ArrayOfCues" maxOccurs="3000">
            <String length="4" />
            <Number size="32" />
            <String length="4" />
            <Number size="32" />
            <Number size="32" />
            <Number size="32" />
        </Block>
    </Block>
</DataModel>
```

更多详细内容请阅读 DataModel 章节、Block 章节、String 章节和 Number 章节。

不要有任何惊讶，我们只是在重用之前的例子。再说一次，不会给每个东西都取名字。

（8）Playlist Chunk

Playlist Chunk 的 Data 同样是由一个列表组成，但是列表的数目包含在列表之前。我们将使用一个数目的关系来构建模型。

```
<DataModel name="ChunkPlst" ref="Chunk">
    <String name="ID" value="plst" token="true"/>
    <Block name="Data">
        <Number name="NumberOfSegments" size="32" >
            <Relation type="count" of="ArrayOfSegments"/>
        </Number>
        <Block name="ArrayOfSegments" maxOccurs="3000">
            <String length="4" />
            <Number size="32" />
            <Number size="32" />
        </Block>
    </Block>
</DataModel>
```

更多详细内容请阅读 DataModel 章节、Block 章节、String 章节、Number 章节和 Relation 章节。

（9）Associated Data List Chunk

这个 Chunk 由一系列的 Label Chunks、Name Chunks 和 Text Chunks 组成。由于不知道它们出现的顺序，因此需要在任何顺序上都支持它们。这是比较容易的，但是在定义 Data List Chunk 之前需要先定义每个子 Chunk。

1、Label Chunk

首先来定义 Label Chunk，这部分数据包含一个以空字符结尾的字符串，这个字符串可能是一个单独的填充字节。

```
<DataModel name="ChunkLabl" ref="Chunk">
    <String name="ID" value="labl" token="true"/>
    <Block name="Data">
        <Number size="32" />
        <String nullTerminated="true" />
    </Block>
</DataModel>
```

更多详细内容请阅读 DataModel 章节、Block 章节、String 章节和 Number 章节。

我们能从 Chunk 中自动地获得填充字节。

2、Note Chunk

它和 Label Chunk 一模一样，我们只需要创建一个 Label Chunk 的别名即可，如下所示：

```
<DataModel name="ChunkNote" ref="ChunkLabl">
    <String name="ID" value="note" token="true"/>
</DataModel>
```

更多详细内容请阅读 DataModel 章节和 String 章节。是的，就是这样，非常容易。

3、Labeled Text Chunk

它和 Note Chunk、Label Chunk 非常相似，不同的是它多了一些数字。复制一个 Label Chunk 的副本，并修改如下：

```
<DataModel name="ChunkLtxt" ref="Chunk">
    <String name="ID" value="ltxt" token="true"/>
    <Block name="Data">
        <Number size="32" />
        <Number size="32" />
        <Number size="32" />
        <Number size="16" />
        <Number size="16" />
        <Number size="16" />
        <Number size="16" />
        <String nullTerminated="true" />
    </Block>
</DataModel>
```

更多详细内容请阅读 DataModel 章节、Block 章节、Number 章节和 String 章节。

4、回到 Associated Data List Chunk

我们把这些子 Chunks 组合进一个列表，如下图所示：

```
<DataModel name="ChunkList" ref="Chunk">
    <String name="ID" value="list" token="true"/>
    <Block name="Data">
        <String value="adtl" token="true" />
        <Choice maxOccurs="3000">
            <Block ref="ChunkLabl"/>
            <Block ref="ChunkNote"/>
            <Block ref="ChunkLtxt"/>
            <Block ref="Chunk"/>
        </Choice>
    </Block>
</DataModel>
```

更多详细内容请阅读 DataModel 章节、Block 章节、Number 章节、String 章节和 Choice 章节。

这里我们介绍一下 Choice 元素。它使每个 Blocks 找到到最佳匹配。你会发现列表的最后是 Chunk。Wave 文件规范表明这里可能会出现其他 Block 的字节。

（10）Sampler Chunk

Sampler Chunk 和我们看到的其他 Chunk 比较相似，它包含一些数字和一些值构成的一个列表。定义如下：

```
<DataModel name="ChunkSmpl" ref="Chunk">
    <String name="ID" value="smpl" token="true"/>
    <Block name="Data">
        <Number size="32" />
        <Number size="32" />
        <Number size="32" />
        <Number size="32" />
        <Number size="32" />
        <Number size="32" />
        <Number size="32" />
        <Number size="32" />
        <Number size="32" />
        <Block maxOccurs="3000">
            <Number size="32" />
            <Number size="32" />
            <Number size="32" />
            <Number size="32" />
            <Number size="32" />
            <Number size="32" />
        </Block>
    </Block>
</DataModel>
```

更多详细内容请阅读 DataModel 章节、Block 章节、Number 章节和 String 章节。

（11）Instrument Chunk

这是最后一个需要定义的 Chunk，而且非常简单，由 7 个 8 比特数字组成。这个超级简单，定义如下：

```
<DataModel name="ChunkInst" ref="Chunk">
    <String name="ID" value="inst" token="true"/>
    <Block name="Data">
        <Number size="8"/>
        <Number size="8"/>
        <Number size="8"/>
        <Number size="8"/>
        <Number size="8"/>
        <Number size="8"/>
        <Number size="8"/>
    </Block>
</DataModel>
```

更多详细内容请阅读 DataModel 章节、Block 章节、Number 章节和 String 章节。这部分的数字不是无符号的，它们取值的范围是负数到正数。

（12）完成 Wav 模型

到了结束这个模型的时候了。回到我们之前接触到的 Wav Chunk，它的定义如下：

```
<!-- 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"/>
</DataModel>
```

更多详细内容请阅读 DataModel 章节、Number 章节和 String 章节。我们将添加一些 Chunks，尽管不知道这些 Chunks 的顺序，这就用到了我们的朋友 Choice 元素。

```
<!-- 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>
```

更多详细内容请阅读 DataModel 章节、Block 章节、Number 章节、String 章节和 Choice 章节。看吧，不是那么难。所有难的都工作结束了，但是在进行 fuzzing 之前我们还有许多事情要做。

3.3.3 创建状态模型

现在我们已经创建了数据模型，接下来我们创建状态模型。对于文件 fuzzing 来说，状态模型非常简单，所有我们要做的就是写文件和启动目标进程。我们要做以下几个操作：

- Output----写文件。
- Close----关闭文件。
- Call-----启动应用程序。

回到 wav.xml 文件，找到名字为 TheState 的 StateModel。展开它，并包含以上 3 个操作（Action），如下所示：

```
<!-- 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>
```

更多详细内容请阅读 StateModel 章节、State 章节、Action 章节、DataModel 章节、Data 章节和 Field 章节。

3.3.4 配置 Publisher

在进行 fuzzing 之前，要做的最后一件事是配置 Publisher。Publishers 是 Peach 的 I/O 连接，它是实现输出、输入和调用等动作之间的管道。对于 File fuzzing，我们将使用一个名字为 File 的 Publisher。这个 publisher 允许写文件和通过 call 操作来启动一个进程。配置 publisher 比较简单，找到 wav.xml 文件的底部，它是 Test 元素的子元素。

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

现在，这个 Publisher 有一个名字为 FileName 的参数，它的值是被 fuzzing 文件的名字，这和在 call 操作中指定文件名字一样。现在，我们需要一种方式来探测目标什么时候崩溃和启动目标。同样地，也要收集一些相关信息便于以后查看，比如堆栈跟踪等。

3.3.5 添加代理和监视器

（1）代理和监视器

现在，准备配置自己的代理和监视器。代理是特定的 Peach 进程，它可以在本地和远程运行，进程拥有一个或者多个监视器，这些监视器能够执行调试器加载和查看内存消耗等操作。本教程准备配置 Microsoft Windbg 来监视 mplayer.exe 的异常和其他常见信息。另外，我们也需要目标进程的 Heap 调试可用。

（2）配置代理和监视器

首先，在样本文件中找到被注释的 Agent 元素，如下图所示：

```
<!-- TODO: Configure agent -->
<Agent name="TheAgent" location="http://127.0.0.1:9000"/>
```

取消注释，删除 Location 属性，这时 Peach 会自动的启动一个本地代理。我们要配置三个代理，分别对应 windows、Linux 和 OSX 三个操作系统。Windows 代理有 Windbg 和 PageHeap 组成；Linux 的代理由 LinuxDebugger 组成；OSX 的代理由 CrashWrangler 组成。如下所示：

```
<Agent name="WinAgent">
   <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" />

        <!-- This parameter will cause the monitor to terminate the process
             once the CPU usage reaches zero.
        -->
        <Param name="CpuKill" value="true"/>

    </Monitor>

    <!-- Enable heap debugging on our process as well. -->
    <Monitor class="PageHeap">
        <Param name="Executable" value="c:\\mplayer\\mplayer.exe"/>
    </Monitor>

</Agent>

<Agent name="LinAgent">
    <!-- Register for core file notifications. -->
    <Monitor class="LinuxDebugger">

        <!-- This is the program we're going to run inside of the debugger -->
        <Param name="Executable" value="mplayer"/>

        <!-- These are arguments to the executable we want to run -->
        <Param name="Arguments"  value="sample.wav"/>

        <!-- This parameter will cause the monitor to terminate the process
             once the CPU usage reaches zero.
        -->
        <Param name="CpuKill" value="true"/>
```

```
            once the CPU usage reaches zero.
        -->
        <Param name="CpuKill" value="true"/>

    </Monitor>

</Agent>

<Agent name="OsxAgent">
    <Monitor class="CrashWrangler">
        <!-- The executable to run. -->
        <Param name="Command" value="mplayer" />

        <!-- The program arguments.  Notice the filename provided matched up
             to what is provided below in the Publisher configuration -->
        <Param name="Arguments" value="fuzzed.wav" />

        <!-- Do not use debug malloc. -->
        <Param name="UseDebugMalloc" value="false" />

        <!-- Treat read access violations as exploitable. -->
        <Param name="ExploitableReads" value="true" />

        <!-- Path to Crash Wrangler Execution Handler program. -->
        <Param name="ExecHandler" value="/usr/local/bin/exc_handler" />

        <!-- This parameter will cause the monitor to wait for an action-call in
             the state model with a method="StartMPlayer" before running
             program.
        -->
        <Param name="StartOnCall" value="StartMPlayer" />

    </Monitor>
</Agent>
```

更多详细内容请阅读 Agent 章节、WindowsDebugger 章节和 PageHeap 章节。

（3）配置 Test 元素

接下来，需要启用我们刚刚配置的代理。找到 Test 元素，取消该行的注释，并修改我们的 Launcher publisher,如下图所示：

```
<Test name="Default">
    <Agent ref="WinAgent" platform="windows"/>
    <Agent ref="LinAgent" platform="linux"/>
    <Agent ref="OsxAgent" platform="osx"/>

    <StateModel ref="TheState"/>

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

更多详细内容请阅读 Test 章节。

（4）配置记录

现在，我们的监视器正在探测程序的各种错误，因此，还需要配置一个信息记录机制来捕获 fuzzer 运行的结果，在 XML 文件底部的 Test 标签中添加 logger 元素就可以实现该机制。如下图所示：

```
<Test name="Default">
    <Agent ref="WinAgent" platform="windows"/>
    <Agent ref="LinAgent" platform="linux"/>
    <Agent ref="OsxAgent" platform="osx"/>

    <StateModel ref="TheState"/>

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

    <Logger class="Filesystem">
        <Param name="Path" value="logs" />
    </Logger>
</Test>
```

更多详细内容请阅读 Test 章节、Logger 章节和 File Logger 章节。

（5）测试 Fuzzer

现在让我们开始进行 fuzzing。打开一个命令行窗口，运行命令如下：

```
c:\wav>c:\peach\peach.exe -t wav.xml

] Peach 3 Runtime
] Copyright (c) Michael Eddington

File parsed with out errors.
```

如果你看到这个输出内容，说明所有步骤的配置正常。如果出现问题，回到前面出问题的部分，并尝试找出问题和解决它。

（6）运行 Fuzzer

现在让我开始正真的 fuzzer 之旅。在命令行中，运行“peach wav.xml”开始 fuzzing。

3.3.6 优化测试计数

当开始进行 fuzzing 的时候，还有一些事情可以优化，它的目的是减少模糊器产生迭代的次数。比如，所有真实的 PCM/WAV 样本和音乐数据可能是我们不需要 fuzzing 的东西了，这些所有的改变会产生不悦耳的声音。因此，让我们调低产生它们的变异器。

3.3.7 并行运行

现在我们拥有了一个优化过的 fuzzer，但是我们仍然需要 fuzzing 过程更快一些，恰好我

们有一些进行 fuzzing 的额外机器。幸运地是 Peach 支持并行 fuzzing，而且非常简单。我们要做的就是添加一个命令行参数，并且在每一个机器上运行。具体如下：

（1）配置机器

首先，需要配置每一台机器，每台机器上必须要有 Peach 程序、目标应用程序、wav.xml 和 sample.wav。

（2）运行 Peach

现在，仅仅需要在每一台机器运行 fuzzer 即可。使用的语法如下：

机器 1：c:\peach\peach.exe -p3,1 wav.xml

机器 2：c:\peach\peach.exe -p3,2 wav.xml

机器 3：c:\peach\peach.exe -p3,3 wav.xml

请注意，我们在命令行中添加了带有两个数字的参数“-p”。第一个参数代表了要使用的机器的总数，第二个数字代表 Peach 运行在第几个机器上。非常简单。

4 方法论

本部分内容包含模糊测试的基本方法论。同时，也对 fuzzer 工具的开发步骤进行了介绍，网址如下：http://community.peachfuzzer.com/Development.html。

- 系统风险分析
  - 确定信任边界
  - 数据流（DFD）
  - 代码年代
- 文件 fuzzing
  - 分析 CRC’s、crypt 等的格式
  - 收集样本文件（大量的）
  - 执行 minset 覆盖率分析（peach/tools/minset）
  - 执行 fuzzing
    - 构建一个“dumb”fuzzing 模板
    - 根据需要进行智能模糊测试
    - 用 Microsoft SDL fuzzing 的需求便于知道什么时候停止
- 网络 fuzzing
  - 分析 CRC’s、crypto 等的格式
  - 收集样本用例
  - 执行代码覆盖
    - 确定接收 fuzzing 的代码区域
    - 扩展用例进而提高代码覆盖率
  - 执行 fuzzing
    - 构建一个“dumb”fuzzing 模板
    - 根据需要进行智能模糊测试
    - 用 Microsoft SDL fuzzing 的需求便于知道什么时候停止

5 介绍

5.1 Peach 介绍

Peach fuzzing 框架被设计用来加速 fuzzer 的开发，它的服务对象为安全研究员、安全团队、顾问和公司。Peach 通过分离数据模型和被 fuzzing 的状态系统以及 fuzzing 引擎来实现相关功能。同时，Peach 还提供了一个健壮的代理/监视器系统来监视 fuzzing 的运行和探测软件缺陷。Peach 的所有主要组件都是可扩展和可插拔的，具有无限的灵活性。对于安全研究员来说，Peach 提供了创建自定义 fuzzing 策略和数据模型的能力。这样的组合允许用户全面控制如何进行 fuzzing。Peach 有以下几个高级概念：

（1）Modeling---Peach 通过提供数据模型和状态模型的 fuzzing 来实现操作。数据模型和状态模型是 Peach 的重点。对于普通的 Peach 用户来说，这是最花费时间的。模型中的细节水平区分了盲模糊测试和智能模糊测试。

（2）Publisher---它是 I/O 接口，描述了输入、输出和调用等抽象概念。正如在状态模型中看到的，它提供了真实的通道或实现。Peach 包含有许多的 Publisher，它提供了许多能力，比如写文件、通过 TCP/UDP 或者其他协议连接、构造 web 请求，甚至是调用 com 对象。创建自定义的 Publisher 是非常容易的。

（3）Fuzzing 策略---fuzzing 策略是如何进行 fuzzing 的逻辑。比如，在同一时刻，我们要修改一个还是多个数据元素？我们使用哪个变异器？是不是要修改模型部分比其他部分多？是否要修改状态模型？Fuzzing 策略唯一不能做的是产生真实的数据，这部分是由变异器实现的。Peach 包含的一些 fuzzing 策略对于大部分使用者来说是非常重要的。

（4）变异器---变异器用于产生数据。它通过修改存在的默认数据值来产生新的数据。变异器常常包含一个简单的逻辑，进而完成一个简单的变异类型。比如，产生-50 到 50 之间的数据，或者产生 1 到 10000 长度的字符串，或者产生 0 到 32 之间的 500 个随机数。

（5）代理---代理是特殊的 peach 进程，它能够在本地或者远程运行，拥有一个或者多个监视器或者远程 Publisher。代理是 Peach 框架提供的具有健壮性的基础监视设备，它允许通过一个有许多层级的复杂系统来监视简单的 fuzzing 配置。一个 Peach fuzzer 可拥有 0 个或者多个代理。

（6）监视器---监视器运行在 Peach 代理进行中，它用来执行通用的任务，比如在 fuzzing 迭代中捕获网络流量、把调试器挂载到一个进程上来探测软件崩溃、网络服务崩溃或者停止时重启它。Peach 包含许多监视器，添加新的监视器并编辑它是非常容易的。

（7）记录器---用来保存崩溃和 fuzzing 运行信息，Peach 默认拥有一个文件系统记录器。

5.2 使用 Peach 进行 Fuzzing

Peach 提供了一个具有强大监视能力的引擎，然而还有一些工作是留给用户的。使用 Peach 进行 fuzzing 的主要步骤如下：

1、创建模型
2、选择/配置 Publisher
3、配置代理/监视器
4、配置记录

6 训练

6.1 Peach 训练

Peach 已经成长为一个复杂的套件，比如在一些安全会议上提供训练。同时，这个训练也提供给喜欢在网站上训练的公司，和让人员工参加会议训练相比，通常要节省不少成本。

6.2 会议训练

在以下安全会议上均提供有 Peach 训练：Blackhat Vegas、CanSecwest、PacSec。对于不同的会议和国家来说，训练的价格是不一样的，具体的价格请查看各自的网站。

6.3 在线训练

Peach 也提供了自定义内容的在线训练。更多信息请联系 Michael Eddington，邮箱地址为： mike@dejavusecurity.com。

最小班人数	5 个学生
最大班人数	20 个学生
讲师数目	少于 10 学生 1 个讲师，多于 10 个学生 2 个讲师
能自定义内容吗？	可以，但需要额外费用
公司提供的必备设施有哪些？	训练场地、每个学生的训练机器、投影仪、白板等

6.4 训练大纲

整个训练课程注重以学生为中心，动手实践和实验室加强。第一天，学生能学习到 Peach Fuzzing 框架的使用方法，从实践者的角度，学习在面对多种目标时如何选取 fuzzing 方式，包括网络协议解析器、ActiveX/COM 接口、文件解析器、API 和 web 服务。学生将以真实的应用程序为目标进行构建和 fuzzing 学习。第二天，学生将以开发者的角度学习 Peach 的内部细节原理。为了方便配备必要的扩展技能和适应学生的自定义需求，Peach 的架构和模块接口解释的非常详细。学生可以在实验室环境中编写自己的 Peach 插件进而加强相关概念。

完成该课程的学习，学生能够具备创建高效模糊器的能力，具体目标如下：

- 状态感知网络协议解析器。
- 多层架构应用程序
- COM 和 Active/x 组件

学生将具备在自己独一无二的环境中应用这些概念和工具的能力，进而运用并行 fuzzing 来提高 fuzzing 的效率。

（1）预备知识：

- 有能力使用 windows

- 有能力使用 wireshark
- XML 或者 HTML 语言的基本知识

（2）笔记本电脑配置需求：

- 能够运行两个 windows 虚拟机的笔记本电脑
- 双核 CPU，内存大于 2G
- 硬盘至少 20G
- 以下虚拟平台之一
  - VMware Server2.0
  - VMware Player 2.0（免费版）
  - VMware Workstation 6.x
- 以下设备之一
  - USB2.0 接口
  - 双层 DVD 光驱

7 Peach 3

7.1 Peach Pits 文件

Peach Pits 是一些 XML 文件，包含 Peach 执行 fuzzing 所需的所有信息。当你使用 Peach 进行 fuzzing 时，需要创建一个 Peach Pit 文件，它主要包含以下几项内容：通用配置、数据模型、状态模型、代理和监控以及测试配置。也许你也想知道怎么调试 pit 文件和怎么验证 pit 文件是否有效（目前这两部分内容正在建设中）。

7.2 通用配置

Peach Pits 文件的第一部分是通用配置。这里是包含其他 Peach Pits 文件的位置，比如外部引用接口、默认属性设置、python 模块路径配置和自定义代码导入等。

7.2.1 Include

Include 元素允许把其他 pit 文件包含到当前 pit 文件的名称空间中使用。当引用被包含的 Pit 文件时，用名称空间前缀和冒号的格式来命名。格式为：Foo：DataModel，如下所示：

```
<Include ns="foo" src="file:foo.xml" />

<DataModel name="NewDataModel" ref="foo:TheDataModel">
</DataModel>
```

属性：

Ns---必须的。名称空间前缀。

Src---必须的。源码 URL，用“file：”前缀来命名文件名。

7.2.2 Defaults

Defaults---用来设置 Data 元素的默认属性，比如字节顺序。（注：目前该元素的详细内容，官网打不开）。

7.2.3 PythonPath

PythonPath 元素是一个顶层元素，它添加一个被 python 模块搜索的路径，主要作用是延伸 Peach 和包含自定义代码的位置。

```
<PythonPath path="c:/peach/mycode">
```

属性：

Path---必须的。要添加的路径。

7.2.4 Import

Import 元素允许在 pit 文件中导入自定义的 python 模块来使用。它就像 python 的关键字 import 一样。（注意，目前 Peach 3 不支持 from 属性）

```
<Import import="MyCode" />
```

属性：

Import--必须的。和 python 的关键字 import 一样。

7. 3 数据模型

Peach Pit 文件包含至少一个 DataModel 元素，也可以更多。DataModel 描述的数据包括类型信息、关系信息（大小、数目、偏移）和其他让模糊器执行智能变异的信息。DataModel 能被其他的 DataModel 重用和引用，允许将复杂的定义分解为可读部分。

7.3.1 DataModel

DataModel 是 Peach 根元素的子元素之一，它通过添加子元素（比如 Number、Blob 或者 String）的方式定义了数据块的结构。

（1）属性：

Name---必须的。当引用模型或者调试时，友好的 DataModel 名字是非常有用的。

Ref---可选的。引用一个 DataModel 模板。

Mutable---可选的，默认为真。该元素是否可变异。

Constraint---可选的。确定一个表达式，它帮助 Peach 确定数据元素是否已被适当的消耗。

（2）子元素：

Block、Choice、Custom、Flag、Flags、Number、Padding、String、XmlAttribute、XmlElement、Relation、Fixup、Transformer、Placement。

（3）例子：

一个 Peach 文件中可以指定任意多个 DataModel 元素，但每个 DataModel 的名字必须唯一。通过 DataModel 可以将复杂的格式按照逻辑分解为更小的模型，使数据模型更易阅读、调试和重用。一个名字为“HelloWorld”的 DataModel 包含一个字符串和输出“Hello World！”如下所示：

```
<DataModel name="HelloWorld">
  <String value="Hello world!" />
</DataModel>
```

一个 DataModel 可以引用其他 DataModel，可以继承带有 ref 属性的子元素。如下所示：

```
<DataModel name="ParentModel">
  <String value="Hello " />
</DataModel>

<DataModel name="HelloWorldModel" ref="ParentModel" >
  <String value=" world!" />
</DataModel>
```

（4）引用（ref 属性）：

当一个引用（ref 属性）被提供时，被引用 DataModel 元素的内容将被复制，并已次为基础来创建新的 DataModel 元素。新 DataModel 的任何子元素将覆盖具有相同名字的基础元素。在这个例子中，自定义的子模块中包含一个名字为 Key 的字符串，它的值将会覆盖没有值的父字符串“Key”，如下所示：

```
<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>
```

```
The output of "Template"   is " : \r\n"
The output of "Customized" is "Content-Length: 100\r\n"
```

当一个 DataModel 被解析时，自定义 DataModel 看起来像是两个数据模型的组合，如下所示：

```
<DataModel name="Customized" ref="Template">
  <String name="Key" value="Content-Length" />
  <String value=": " token="true" />
  <String name="Value">
    <Relation type="size" of="HttpBody" />
  </String>
  <String value="\r\n" token="true" />
</DataModel>
```

7.3.2 Blob

Blob 元素是 DataModel 或 Block 的一个子元素。Blob 元素常常用于代表缺少类型定义或格式的数据。如下所示：

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

（1）属性（除非声明，所有的属性都是可选的）：

Name---必须的。Blob 的名字。

Value---含有 Blob 的默认值。

Length---Blob 的大小，单位为字节。

Ref---引用一个数据模型来作为 Blob 的模板。

valueType---默认格式的值，hex，string，或者 literal，默认为 string。

minOccurs---该 Blob 元素必须发生变化的最小次数，默认为 1。

maxOccurs---该 Blob 元素能够发生变化的最大次数，默认为 1。

Token---当解析时该元素应该作为一个令牌来信任，默认是假。

lengthType---长度的类型，指定长度。

Constraint---一个约束的形式表达，用于数据破解。

Mutable---Blob 元素是否可变异（是否能被 fuzzing），默认为真。

（2）子元素：

Anayzers

（3）例子：

一个简单的 Blob。这个 Blob 中，任何类型或长度的数据能破解。

```
<Blob name="Unknown1" />
```

包含有默认值的 Blob：

```
<Blob name="Unknown1" valueType="hex" value="AA BB CC DD" />
```

7.3.3 Block

Block 是 DataModel 或 Block 元素的子元素。Block 用于在一个逻辑结构中将一个或者多个数据元素（Number 或 String）组织在一起，它和 Datamodel 非常相似，仅有的差异是它们的位置。DataModel 是个顶层元素，Block 是 DataModel 的一个子元素，它们都可以作为其他 Block 或 DataModel 的模板。

```
<Block name="HelloWorld">
  <String value="Hello world!" />
</Block>
```

（1）属性（除非声明，所有的属性都是可选的）：

Name---Block 的名字。

Ref---引用一个数据模型来作为 Block 的模板。。

minOccurs---该 Block 必须发生变化的最小次数。

maxOccurs--该 Block 可以发生变化的最大次数。

Mutable---元素是否可变异，默认为真。

（2）子元素：

Blob、block、Choice、Custom、Fixup、Flag、Flags、Number、Padding、Placement、Relation、Seek、String、Transformer、XmlAttribute、XmlElement。

（3）例子：

空 Block。最简单的 Block 是空 Block，这个定义将没有输出。

```
<DataModel name="BlockExample1">
 <Block>
 </Block>
</DataModel>
```

嵌套的 Block。Block 可以根据需要多层的嵌套，它可以帮助创建逻辑结构而不改变数据包含的内容。

```
<DataModel name="BlockExample2">
<Block>
  <Block>
    <Block>
      <String value="1" />
    </Block>

    <Block>
      <String value="2" />
    </Block>

    <String value="3" />
  </Block>
  <String value="4" />
</Block>
</DataModel>
```

这个嵌套的 Block 定义产生的输出为：1 2 3 4 。

命名一个 Block。给 Block 取一个友好的名字使他们容易理解和调试。

```
<DataModel name="BlockExample2">
 <Block name="HeaderDef">
  <String name="Header" />
  <String name="Colon" value=":"/>
  <String name="Val"/>
 <Block>

 <Block name="DataDef">
   <Number name="Type"  size="8" value="4"/>
   <Number name="Data" size="8" value="32"/>
 </Block>
 </DataModel>
```

引用一个 Block。引用的内容将被复制，并作为基础来创建新的 Block。新 Block 中的任何子元素将覆盖那些已经存在的具有相同名字的元素。

```
<DataModel name="OtherDataModel">
   <String value="Hello World"/>
 </DataModel>

 <DataModel name="ThisDataModel">
    <Block name="MyName" ref="OtherDataModel"/>
 </DataModel>
```

名字为“MyName”的 Block 将覆盖被引用的“OtherDataModel”。当被解析时，它的数据结构如下所示：

```
<DataModel name="ThisDataModel">
  <Block name="MyName">
   <String value="Hello World"/>
  </Block>
</DataModel>
```

引用属性允许构建健壮的模板，如下所示模板的名字为“Key”，值为“\r\n”。

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

使用该模板作为一个引用

```
<DataModel name="OtherModel">
  <String value="Before Block\r\n" />

  <Block name="Customized" ref="Template">
    <String name="Key" value="Content-Length" />
    <String name="Value" value="55"/>
  </Block>
</DataModel>
```

输出为：

```
Before Block\r\n
Content-Length: 55\r\n
```

两个关键字符串在这里发生冲突。当解析时，自定义的 Block 将代替它的 DataModel 模板的结构。添加字符串值“：\r\n”。同时“customized”将覆盖 String 元素的“Key”和“Value”的值，用“Content-Length”和 55 代替。最终的 DataModel 将被解析如下：

```
<DataModel name="OtherModel">
  <String value="BeforeBlock" />

  <Block name="Customized" ref="Template">
    <String name="Key" value="Content-Length" />
    <String value=": " token="true" />
    <String name="Value" value="55" />
    <String value="\r\n" token="true" />
  </Block>
</DataModel>
```

7.3.4 Choice

Choice 是 DataModel 或者 Block 元素的的子元素之一。Choice 元素用于指示任何子元素是有效的，但是只应选择一个，很像编程语言中的 switch 语句。

```
<Choice name="ChoiceBlock">
  <Block name="Type1">
    <!-- ... -->
  </Block>
  <Block name="Type2">
    <!-- ... -->
  </Block>
  <Block name="Type3">
    <!-- ... -->
  </Block>
</Choice>
```

（1）属性（除非声明，所有的属性都是可选的）：

Name---choice 元素的名字。

minOccurs---该 Choice 必须发生改变的最小次数。

maxOccurs---该 Choice 能发生改变的最大次数。

Occurs---该 choice 能发生改变的迭代次数。

（2）子元素：

Block、Choice、String、Number、Blob、Flags、Fixup、Transformer、XmlAttribute、XmlElement。

（3）例子：

一个基本的 Choice。这个例子将破解或消耗 1,2,3 类型的数据，很像一个需要在令牌上做出决定的常规切换语句。它的前 8 个字节是 1，剩下的数据被视为一个 32 位数字。如果前 8 位是 2，剩下的数据被视为一个 255 字节的二进制数据。如果前 8 位是 3，剩下的数据被视为一个 8 字节字符串。当 fuzzing 时，Peach 将选择其中的 1 个类型并进行 fuzzing，它的输出为一个 8 位数字，后跟相应的类型。Peach 将会尝试所有的 3 个类型。

```
<DataModel name="ChoiceExample1">
 <Choice name="Choice1">

  <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>
```

一系列的 Choice。第一个例子适合构建单个 Choice，但如果有许多 Type1 、Type2 和 Type3 块都是彼此跟随的，该怎么做呢？。通过设置 minoccurs、maxoccurs 或者 occurs 属性，可以指定 Choice 应该被重复。这个例子尝试来破解至少 3 个，最多 6 不同的 Choice。

```
<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>
```

7.3.5 Custom

该元素的相关内容正在建设中。

7.3.6 Flag

Flag 元素在 Flags 容器中定义了一个特定的位区域。

```
<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>
```

（1）属性：

Name---可选的。元素的名字。

Size---必须的。大小，以位为单位为

Position---必须的。Flag 的开始位置（以 0 位基础）。

Value---带有 Blob 的默认值。

valueType---格式的默认值（hex，string，literal）。

Mutable---可选的。数据元素是否变异，默认为真（peach 2.3）。

（2）子元素：

Relation。

7.3.7 Flags

Flags 定义了一组 Flag 的大小。

```
<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>
```

（1）属性：

Name---可选的。元素的名字。

Size---必须的。大小，以位为单位。

Mutable---可选的。元素是否可以变异，默认为真。

（2）子元素：

Fixup、Flag、Placement、Relation、Transformer。

7.3.8 Number

该元素定义了长度为 8,16,24,32，或 64 位长度的二进制数。它是 DataModel、Block 或者 Choice 的子元素。

（1）属性：

Name---必须的。Number 的名字。

Size---必须的。Number 的大小，以位为单位。有效值为 1 到 64。

Value---分配给 Number 的默认值。

valueType---可选的。value 的表现方式。有效选项为 string（字符串）和 hex（十进制）。

Token---当解析的时候，该元素被视为一个令牌，默认值为假。有效选项为真和假。

Endian---Number 的字节顺序。默认为小端。有效选项为大端、小端和网络。网络一样是大端。

Signed---Number 是否为有符号数据。默认为真。有效选项为真和假。

Constraint---一个以 Python 表达式为形式的约束。用于数据破解。

Mutable---元素是否可改变（fuzzing 时是否可变异），默认为真。有效选项为真和假。

minOccurs---Number 必须发生改变的最小次数，默认为 1。有效选项为正整数值。

maxOcuurs---Number 能够发生改变的最大次数，没有默认值。有效选项为正整数值。

（2）有效子元素：

Anayzers、Fixup、Relation、Transformer、Hint。

（3）例子：

一个简单的 Number 例子，它将产生一个 32 位（4 字节）Number，默认值为 5。

```
<DataModel name="NumberExample1">
 <Number name="Hi5" value="5" size="32"/>
</DataModel>
```

为了只使用 16 位（2 字节），改变 size 的值为 16.

```
<DataModel name="NumberExample2">
 <Number name="Hi5" value="5" size="16"/>
</DataModel>
```

有符号。为了表明这是一个无符号数据元素，设置 signed 属性等于“false”。默认为真。

```
<DataModel name="NumberExample3">
 <Number name="Hi5" value="5" size="32" signed="false"/>
</DataModel>
```

Value 类型。值类型定义了怎么解释 Value 的属性。有效选项为 string 和 hex，默认为 string。将值 1000 分配给 Hi5。

```
<DataModel name="NumberExample4">
 <Number name="Hi5" value="1000" valueType="string" size="16" signed="false" />
</DataModel>
```

将 43981 以十六进制形式分配给 Hi5。

```
<DataModel name="NumberExample5">
 <Number name="Hi5" value="AB CD" valueType="hex" size="16" signed="false" />
</DataModel>
```

小端。为了改变 Number 的字节顺序，请设置 endian 属性。

```
<DataModel name="NumberExample6">
 <Number name="Hi5" value="AB CD" valueType="hex" size="16" signed="false" endian="big" />
</DataModel>
```

上图将产生如下字节顺序：AB CD。

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

上图将产生如下字节顺序：CD AB。

7.3.9 Padding

Padding 元素用来填充大小变化的块或数据模型。

（1）属性：

Name---必须的。Number 元素的名字。

Aligned---将父元素对齐到 8 字节边界，默认为假。

Alignment---对齐到这个位边界，比如（8、16 等），默认为 8。

alignedTo---基于我们要填充的元素名字。

Lengthcalc---计算结果为整数的脚本表达式。

Constraint---一个以 Python 表达式形式的约束。用于数据破解。

Mutable---元素是否可变异，默认为真，有效选项为真和假。

（2）有效子元素：

Fixup、Relation、Transformer、Hint。

（3）例子：

```
<DataModel name="NumberExample1">
 <String name="VariableSizeString" />

 <Padding aligned="true"/>
</DataModel>
```

7.3.10 String

该元素定义了一个单字节或者双字节的字符串，它是 DataModel 或者 Block 的子元素。为了指定这是一个数值的字符串，请用 NumericalString 元素。

```
<String value="Hello World!" />

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

（1）属性：

Name---可选的，数据模型的名字。

Length---可选的，字符串的长度。

lengthType---可选的，Length 属性的单位。

Type---可选的。字符编码类型，默认为“ASCII”，有效选项为 ASCII、utf7、utf8、utf6、utf6be、utf32。

nullTerminated---可选的。是否为以空字节结尾的字符串（真或者假）。

padCharacter---可选的。根据 length 参数填充字符串的字符，默认为（0x00）。

Token---可选的。当解析的时候，该元素应该被视为一个令牌，默认为假。

Constraint---一个脚本表达式形式的约束。用于数据破解。

Mutable---可选的。元素是否可变异，默认为真。

Minoccurs---可选的。这个块必须发生改变的最小次数，默认为 1。

Maxoccurs---可选的。这个块会发生改变的最大次数，默认为 1。

（2）有效子元素：

Analyzer、Fixup、Relation、Transformer、Hint。

（3）NumericalString：

该元素只能用于 String 来说明它的值是一个数字。当使用这个提示时，它激活所有的数字突变以及标准的字符串突变。请注意：如果默认情况下一个字符串的值是数字，NumericalString 元素被自动添加。

```
<String value="250">
   <Hint name="NumericalString" value="true" />
</String>
```

7.3.11 XmlAttribute

该元素定义了 XML 元素的属性。只有当父元素是 XmlElement 时，才有效。

```
<XmlElement name="example" elementName="Foo">
  <XmlAttribute attributeName="Bar">
    <String value="My Attribute!" />
  </XmlAttribute>
</XmlElement>
```

```
<Foo Bar="My Attribute!"/>
```

（1）属性：

Name---可选的。数据模型的名字。

Minoccurs---可选的。这个块必须发生变化的最小次数。

Maxoccurs---可选的。这个块会发生变化的最大次数。

isStatic---可选的。当解析的时候，该元素被视为一个令牌，默认为假。

Token---可选的。当解析的时候，该元素被视为一个令牌，默认为假（Peach 2.3）。

Mutable---可选的。该元素是否可变异，默认为真（Peach 2.3）。

attributeName---必须的。XML 元素的名字。

Ns---可选的。XML 名称空间。

（2）有效子元素：

Block、Choice、String、Number、Blob、Flags、Fixup、Hint。

7.3.12 XmlElement

定义一个 XML 元素，XML 文档的基本块构建。这用来 fuzzing 一个 XML 文档的内容，但是不包含 XML 解析器。XmlElement 和 XmlAttribute 产生的所有输出都将被格式化。请注意，XmlElement 和 XmlAttribute 元素不支持数据破解。如果需要破解 Xmlelement 和 XmlAttribute 中的 XML，请用挂载到一个 String 元素的 XmlAnalyzer。

```
<XmlElement name="example" elementName="Foo">
  <XmlElement elementName="Bar">
    <String value="Hello World!" />
</XmlElement>
```

```
<Foo><Bar>Hello World!</Bar></Foo>
```

（1）属性：

Name---可选的。数据模型的名字。

Minoccurs---可选的。这个块必须发生变化的最小次数。

Maxoccurs---可选的。这个块会发生变化的最大次数。

isStatic---可选的。当解析的时候，该元素被视为一个令牌，默认为假。

Token---可选的。当解析的时候，该元素被视为一个令牌，默认为假（Peach 2.3）。

Mutable---可选的。该元素是否可变异，默认为真（Peach 2.3）。

attributeName---必须的。XML 元素的名字。

Ns---可选的。XML 名称空间。

（2）有效子元素：

XmlElement、XmlAttribute、Block、Choice、String、Number、Blob、Flags、Fixup、Hint。

7.3.13 Hint

Hint 是变异器的扩展。它能附加到数据元素上，为 Peach 引擎提供更多关于被解析数据的信息。例如，当字符串包含一个数字时，只有包含在字符串 mutator 中的数字测试才会执行。如果你添加一个 NumericalString 提示到 String，它将附加所有的数字变异器。

```
<String value="250">
   <Hint name="NumericalString" value="true" />
</String>
```

可用的 Hints：

NumericalString、ArrayVarianceMutator-N、DWORDSliderMutator、BitFlipperMutator-N、NumericalVarianceMutator-N 、 Telecommunications-N 、 FiniteRandomNumbersMutator-N 、SizedVaranceMutator-N 、 SizedNumericalEdgeCasesMutator-N 、 SizedDataVaranceMutator-N 、SizedDataNumericalEdgeCasesMutator-N、type、ValidValues。

7.3.14 Relation

Peach 允许构建数据间的关系。关系是类似这样的东西“X 是 Y 的大小”、“X 是 Y 的数量”、或者“X 是 y 的偏移（字节单位）”。

（1）大小关系：

在这个例子中，Number 元素的值将确定名字为 TheValue 的 string 元素的大小。请注意，这也适用于多字节字符，如 wchar。在 Peach 未来的版本中，这将会改变，或者将包括新的类型长度关系，以便更好地支持 UTF-8 和其他 Unicode 编码。

```
<Number size="32" signed="false">
  <Relation type="size" of="TheValue" />
</Number>
<String name="TheValue" />
```

在这个例子中，我们将提供两个 python 表达式，它允许在获取或设置 size 属性的时候修改它的大小，有两个变量可用，分别为 self 和 size。Self 是 Number 元素的一个引用，size 是一个整数。获取操作和设置操作应该是彼此的数学逆操作。在破解过程中应用获取操作，在发布过程中应用设置操作。

expressionGet---该表达式的结果用于内部，它确定名字为 TheValue 的 String 元素读取多少字节。如果 Peach 取 10，它将在内部存储一个 5，然后 Peach 将读取 5 个字节到 String 中。

ExpressionSet---为 publisher 生成一个值。 在以下示例中，为 TheValue 存储的 Size 的值“5”（TheValue 的长度），因此 Peach 通过 publisher 输出的值将为“5 * 2”或 10。

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

（2）数量关系：

在这个例子中，Number 将会说明 String 列表的数目。

```
<Number size="32" signed="false">
  <Relation type="count" of="Strings" />
</Number>
<String name="Strings" nullTerminated="true" maxOccurs="1024" />
```

在这个例子中，我们将提供两个 python 表达式，它允许在获取或设置 size 属性的时候修改它的大小。有两个变量可用，分别为 self 和 count。Self 是 Number 元素的一个引用，count 是一个整数。这里的让 count 可用与前面的表达式不同。虽然 self 在表达式对中始终可用，但其他可用的变量的名字是 Relation 元素 type 属性的值。

expressionGet---该表达式的结果用于内部，它确定 String 元素将扩展到多少项。maxOccurs 是 Peach 循环计算中遇到的最大值，由于 maxOccurs = 1024 的限制，Peach 在 CountIndicator 元素中破解时遇到的最大值是 2048。

ExpressionSet---设置要生成的值。 以下示例中，count 根据读入的 String 元素数目确定。

```
<Number name="CountIndicator" size="32" signed="false">
  <Relation type="count" of="TheValue" expressionGet="count/2" expressionSet="count*2" />
</Number>
<String name="TheValue" nullTerminated="true" maxOccurs="1024" />
```

（3）偏移关系：

偏移关系是 Peach 最新增加的，它允许需修改格式，这些格式需要偏移的改变和输出变量元素的偏移。这里有一些元素，它们是各种 String 元素偏移量的 ascii 表示。

```
<DataModel name="TheDataModel">
        <String length="4" padCharacter=" ">
                <Relation type="offset" of="Offset0" />
        </String>
        <String length="4" padCharacter=" ">
                <Relation type="offset" of="Offset1" />
        </String>
        <String length="4" padCharacter=" ">
                <Relation type="offset" of="Offset2" />
        </String>
        <String length="4" padCharacter=" ">
                <Relation type="offset" of="Offset3" />
        </String>
        <String length="4" padCharacter=" ">
                <Relation type="offset" of="Offset4" />
        </String>

        <String length="4" padCharacter=" ">
                <Relation type="offset" of="Offset5" />
        </String>

        <String length="4" padCharacter=" ">
                <Relation type="offset" of="Offset6" />
        </String>

        <Block>
                <Block name="Offset0">
                        <Block>
                                <String name="Offset1" value="CRAZY STRING!" />
                                <String value="aslkjalskdjas" />
                                <String value="aslkdjalskdjasdkjasdlkjasd" />
                        </Block>
                        <String name="Offset2" value="ALSKJDALKSJD" />
                        <Block>
                                <String name="Offset3" value="1" />
                                <String name="Offset4" value="" />
                                <String name="Offset5" value="1293812093" />
                        </Block>
                </Block>
        </Block>

        <String name="Offset6" value="aslkdjalskdjas" />

</DataModel>
```

1、相对偏移量：

从 Peach 2.3 开始支持相对偏移的概念。 相对偏移来自附加到 relation 的数据元素。 请考虑以下示例。，当确定 StringData 的偏移量时，Peach 将根据需要，对 OffsetToString 的位置进行加上/减去它的值，以确定正确的偏移量。

```
<!-- Other data elements precede -->

<Number name="OffsetToString">
   <Relation type="offset" of="StringData" relative="true" />
</Number>

<String name="StringData" nullTerminated="true"/>
```

2、相对于偏移量：

Peach 还支持相对于另一个元素的偏移。它用于以下情况，一个元素包含另外一个元素的偏移，而被包含的偏移的元素是一个结构体的开头。 以下示例中，StringData 偏移量将通过添加 OffsetToString 的值到 Structure 位置的方式来计算。

```
<Block name="Structure">
   <!-- Other data elements precede -->

   <Number name="OffsetToString">
      <Relation type="offset" of="StringData" relative="true" relativeTo="Structure" />
   </Number>

   <String name="StringData" nullTerminated="true"/>
</Structure>
```

包含 expressionGet / expressionSet

当使用带有偏移关系的 expressionGet / Set 时，将提供两个变量：self 和 offset。 Self 引用了一个父元素的引用，offset 是一个整数。

包含 Placement 的偏移关系

这个模型中，将使用一个典型的模式。在这个模式中，一个偏移列表会给我们另一个元素的位置。 使用 Placement 元素将创建的 Data 字符串移动到 Chunks 块之后。

```
<DataModel name="TheDataModel">
  <Block name="Chunks">
    <Block name="ArrayOfChunks" maxOccurs="4">
      <Number size="8" signed="false">
        <Relation type="offset" of="Data"/>
      </Number>
      <String name="Data" length="6">
        <Placement after="Chunks"/>
      </String>
    </Block>
  </Block>
</DataModel>
```

7.3.15 Fixup

Fixup 是一些代码函数，它通常操作另一个元素数据来产生一个值。校验和算法就是这样一个示例。Peach 默认包含有以下 fixups。

- Utility Fixups
  - CopyValueFixup
  - SequenceIncrementFixup
  - SequenceRandomFixup
- Check-sum Fixups
  - Crc32DualFixup
  - Crc32Fixup
  - EthernetChecksumFixup
  - ExpressionFixup
  - IcmpChecksumFixup
  - LRCFixup
- Hashing Fixups
  - MD5Fixup
  - SHA1Fixup
  - SHA224Fixup
  - SHA256Fixup
  - SHA384Fixup
  - SHA512Fixup

7.3.16 Transformers

Transformers 在父元素上执行静态转换或编码。通常来说 Transformers 是两个方向：编码和解码，但也不全是。 比如 ZIP 压缩，Base64 编码，HTML 编码等。与 Fixups 不同，Transformers 对父元素操作，而 Fixups 引用另一个元素的数据。

```
<DataModel name="Base64TLV">
  <Number name="Type" size="8" signed="false" value="1" token="true" />
  <Number name="Length" size="16" signed="false">
    <Relation type="size" of="base64Block" />
  </Number>

  <Block name="base64Block">
    <Transformer class="Base64Encode" />
    <Blob name="Data" />
  </Block>
</DataModel>
```

上述数据模型的输出为 0x01 <len(b64(Data))> <b64(Data)>

Peach 3 中的默认 Transformers：

- Compression
  - Bz2CompressTransformer
  - Bz2DecompressTransformer
  - GzipCompressTransformer
  - GzipDecompressTransformer
- Crypto
  - Aes128Transformer
  - ApacheMd5CryptTransformer
  - CryptTransformer
  - CvsScrambleTransformer
  - HMACTransformer
  - MD5Transformer
  - SHA1Transformer
  - TripleDesTransformer
  - UnixMd5CryptToolTransformer
  - UnixMd5CryptTransformer
- Encode
  - Base64EncodeTransformer
  - Base64DecodeTransformer
  - HexTransformer
  - HexStringTransformer
  - HtmlEncodeTransformer
  - HtmlDecodeTransformer
  - HtmlEncodeAgressiveTransformer
  - Ipv4StringToOctetTransformer
  - Ipv4StringToNetworkOctetTransformer
  - Ipv6StringToOctetTransformer
  - JsEncodeTransformer
  - NetBiosEncodeTransformer
  - NetBiosDecodeTransformer
  - SidStringToBytesTransformer
  - UrlEncodeTransformer
  - UrlEncodePlusTransformer
  - Utf8Transformer
  - Utf16Transformer
  - Utf16LeTransformer
  - Utf16BeTransformer
  - WideCharTransformer
- Type
  - AsInt8Transformer
  - AsInt16Transformer
  - AsInt24Transformer
  - AsInt32Transformer
  - AsInt64Transformer

- IntToHexTransformer
- NumberToStringTransformer
- StringToFloatTransformer
- StringToIntTransformer

- Misc
  - EvalTransformer

7.3.17 Placement

Placement 元素告诉数据破解者，在输入流被解析之后，特定元素应该被移动。结合偏移关系是 Peach 支持的文件处理方式，它通过偏移来包含元素的引用。

```
<DataModel name="TheDataModel">
  <Block name="Chunks">
    <Block name="ArrayOfChunks" maxOccurs="4">
      <Number size="8" signed="false">
        <Relation type="offset" of="Data"/>
      </Number>
      <String name="Data" length="6">
        <Placement after="Chunks"/>
      </String>
    </Block>
  </Block>
</DataModel>
```

属性：

以下其中之一是必须的：

After---元素移动到之后。

Before---元素移动到之前。

7. 4 状态模型

Peach 中有两个创建 fuzzer 的模型，DataModel 和 StateModel。StateModel 重新创建测试一个协议所必须的基本状态机器逻辑。它定义了怎么给目标发送和接收数据。StateModel 的范围从非常简单到及其复杂。建议在开始时，保持状态模型简单，需要时再进行扩展。

7.4.1 state

state 封装了一个为 Peach 工作的逻辑单元，进而来执行一个大的状态模型。 一些状态模型仅需要一个状态，而另外一些状态则需要许多状态来创建复杂协议模型。state 由 action 组成。 每个 action 可以执行与单个状态如何封装逻辑相关的任务。

属性：

Name---必须的。State 的名字。

有效子元素：Action。

7.4.2 Action

Action 元素能在 StateModel 中执行多种操作。Action 是发送命令给 Publisher 的一种主要方式，它能发送输出，接收输入或打开一个连接。Action 也能在 StateModel 中改为其他状态，在 DataModel 之间移动数据，调用被代理定义的方法。它是 state 元素的子元素。

```
<StateModel name="StateModel" initialState="InitialState">
        <State name="InitialState">

                <Action name="SendData" type="output">
                        <DataModel ref="MyDataModel" />

                        <!-- Optional data element -->
                        <Data name="load defaults" fileName="template.bin" />
                </Action>

        </State>
</TheStateModel>
```

（1）属性：

Name---可选的。action 的名字。

Type---必须的。action 类型。

When---只有在提供的表达式求值为 true 时才执行操作。

onComplete---表达式在完成操作后运行。

onStart---表达式在开始操作时运行。

Method---必须的。类型为 call。用来调用的方法。

Property---必须的。类型为 setproperty 或 getproperty。获取或者设置属性。

Setxpath---必须的。类型为 slurp，设置 Xpath 的值。

Value---值。类型为 slurp。

valueXpath---Xpath 的值。类型为 slurp。

Ref---更改为状态的引用。类型为 changestate。

（2）有效子元素：

DataModel、Data、Param。

（3）Action 类型:

1、start（隐式）

启动 Publisher，这是一个隐式的动作，一般不需要。

```
<StateModel name="StateModel" initialState="InitialState">
        <State name="InitialState">

                <Action type="start" />

        </State>
</TheStateModel>
```

2、stop（隐式）

停止 Publisher，这是一个隐式的动作，一般不需要。

```
<StateModel name="StateModel" initialState="InitialState">
        <State name="InitialState">

                <Action type="stop" />

        </State>
</TheStateModel>
```

3、Open/connect（隐式的）

Open 和 connect 是彼此的别名，执行相同的操作，这个动作也是隐式的，对于文件来说，文件必须被打开或者创建。对于 sockets 来说，应该打开一个连接。只有当需要特殊控制的时候，才使用这个操作。

```
<StateModel name="StateModel" initialState="InitialState">
        <State name="InitialState">

                <Action type="open" />

                <Action type="output">
                        <DataModel ref="DataModelToWrite"/>
                </Action>

        </State>
</TheStateModel>
```

```
<StateModel name="StateModel" initialState="InitialState">
        <State name="InitialState">

                <Action type="connect" />

                <Action type="output">
                        <DataModel ref="DataModelToSend"/>
                </Action>

        </State>
</TheStateModel>
```

4、Close（隐式的）

Close 同样也是隐式的，一般不使用，除非需要特殊控制时使用。

```
<StateModel name="StateModel" initialState="InitialState">
        <State name="InitialState">

                <Action name="FileWrite" type="output">
                        <DataModel ref="FileHeader"/>
                </Action>

                <Action name="FileClose" type="close" />

        </State>
</TheStateModel>
```

5、Accept

接收一个连接。并不是所有的 Publisher 都支持这个操作类型。这个操作通常会堵塞直到一个连接来到时可用。

```
<StateModel name="StateModel" initialState="InitialState">
        <State name="InitialState">

                <Action name="AcceptConnection" type="accept" />

                <Action name="ParseIncomingPacket" type="input">
                                <DataModel ref="PacketModel"/>
                </Action>

        </State>
</TheStateModel>
```

6、Input

从 Publisher 中接收或者读取输入。需要一个指定 DataModel 来破解和包含到来的数据。

```
<StateModel name="StateModel" initialState="InitialState">
        <State name="InitialState">

                <Action type="input">
                        <DataModel ref="InputModel" />
                </Action>

        </State>
</TheStateModel>
```

7、Output

通过 Publisher 发送或写 output 操作。需要一个 DataModel，可随意地提供一个的 Data 设置。

```
<StateModel name="StateModel" initialState="InitialState">
        <State name="InitialState">
                <Action type="output">
                        <DataModel ref="SomeDataModel" />
                </Action>

                <Action type="output">
                        <DataModel ref="SomeDataModel" />
                        <Data name="SomeSampleData" fileName="sample.bin" />
                </Action>
        </State>
</TheStateModel>
```

8、Call

调用一个被 Publisher 定义的方法，这个 Publisher 含有可选参数。并不是所有的 Publisher 都支持。

```
<StateModel name="StateModel" initialState="InitialState">
        <State name="InitialState">
                <Action type="call" method="openUrl">
                        <Param name="p1" type="in">
                                <DataModel ref="Param1DataModel" />
                        </Param>
                        <Param name="p2" type="in">
                                <DataModel ref="Param2DataModel" />
                                <Data name="p2data">
                                        <Field name="value" value="http://foo.com" />
                                </Data>
                        </Param>
                </Action>
        </State>
</TheStateModel>
```

9、Setproperty

设置一个属性。并不是所有的 Publisher 都支持。

```
<StateModel name="StateModel" initialState="InitialState">
        <State name="InitialState">
                <Action type="setProperty" property="Name">
                        <DataModel ref="NameModel"/>
                </Action>
        </State>
</TheStateModel>
```

10、Getproperty

获取一个属性。并不是所有的 Publisher 都支持。

```
<StateModel name="StateModel" initialState="InitialState">
        <State name="InitialState">
                <Action type="getProperty" property="Name">
                        <DataModel ref="NameModel"/>
                </Action>
        </State>
</TheStateModel>
```

11、Slurp

Slurp 用于在两个 DataModel 之间移动数据。这些 DataModel 被分配给一个 StateModel 中的不同 action。一个标准的用例是在一个协议序列中。一个序列 ID 或者一个怀疑 ID 需要被发送回服务器。Slurp 将从一个 action 复制数据到另外一个 action。从一个服务器输入，输出到另外一个服务器。

```
<DataModel name="ReceiveChallenge">
  <String name="Challenge" />
</DataModel>

<DataModel name="SendChallenge">
  <String name="Challenge" />
</DataModel>


<StateModel name="StateModel" initialState="InitialState">
        <State name="InitialState">
                <Action name="ReceiveChallenge" type="input">
                        <DataModel name="TheReceiveChallenge" ref="ReceiveChallenge"/>
                </Action>

                <Action type="slurp" valueXpath="//TheReceiveChallenge//Challenge" setXpath="//TheSendChallenge//Challenge" />

                <Action name="SendChallenge" type="output">
                        <DataModel name="TheSendChallenge" ref="SendChallenge"/>
                </Action>
        </State>
</TheStateModel>
```

12、Changestate

改变一个不同的状态。它被经常用于带有 when 属性的结合中。

```
<StateModel name="StateModel" initialState="InitialState">
        <State name="InitialState">
                <Action type="input">
                        <DataModel ref="InputModel" />
                </Action>

                <Action type="changeState" ref="State2"/>
        </State>

        <State name="State2">
                <Action type="output">
                        <DataModel ref="OutputModel" />
                </Action>
        </State>
</TheStateModel>
```

13、When

执行一个基于表达式的 action。当表达式计算值为真时，action 被执行。

```
<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>

<DataModel name="OutputModelB">
        <Number name="Type" size="32" value="AA BB CC DD" 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"/>

                <Action type="changeState" ref="State3" when="int(StateModel.states['InitialState'].actions[0].dataModel['Type'].InternalValue) == 3"/>

        </State>

        <State name="State2">
                <Action type="output">
                        <DataModel ref="OutputModelA" />
                </Action>
        </State>

        <State name="State3">
                <Action type="output">
                        <DataModel ref="OutputModelB" />
                </Action>
        </State>
</TheStateModel>
```

7.4.3 状态模型例子

（1）文件 fuzzing

当进行文件 fuzzing 的时候，Peach 把数据写进一个文件，然后调用目标进程打开上述的文件。对于一个简单的文件模糊器，Peach 会用一个单独的 state 和三个 action。

```
<StateModel name="TheStateModel" initialState="InitialState">
        <State name="InitialState">

                <!-- Write out the contents of file.  The publisher attribute matches
                the name provided for the publisher in the Test section. -->
                <Action type="output">
                        <DataModel ref="TestTemplate" />
                </Action>

                <!-- Close the file -->
                <Action type="close" />

                <!-- Launch the file consumer -->
                <Action type="call" method="ScoobySnacks" publisher="Peach.Agent"/>

        </State>
</StateModel>
```

（2）简单网络状态模型：

在这个状态模型中，Peach 将会发生和接收来自一个 TCP 端口的一系列包。

```
<StateModel name="TheStateModel" initialState="InitialState">
        <State name="InitialState">

                <!-- Peach will automatically connect to the remote host -->

                <!-- Send data -->
                <Action type="output">
                        <DataModel ref="PacketModel1" />
                </Action>

                <!-- Receive response -->
                <Action type="input">
                        <DataModel ref="PacketModel2" />
                </Action>

                <!-- Send data -->
                <Action type="output">
                        <DataModel ref="PacketModel3" />
                </Action>

                <!-- Receive response -->
                <Action type="input">
                        <DataModel ref="PacketModel4" />
                </Action>
        </State>
</StateModel>

<Test name="Default">
        <StateModel ref="TheStateModel"/>

        <Publisher class="TcpClient">
                <Param name="Host" value="127.0.0.1" />
                <Param name="Port" value="4242" />
        </Publisher>
</Test>
```

7.5 代理

代理是特殊的 Peach 进程，它可以在本地或者远程运行。这些进程拥有一个或者多个监视器，这些监视器可以执行加载调试器，查看内存消耗或者探测错误等操作。代理中的监视器可以收集信息和代表 fuzzer 执行操作。通用的代理有：本地代理、TCP 远程代理、ZeroMQ、REST Json 代理。

（1）本地代理：

Peach 运行时支持一个运行在进程中的本地代理。如果不指定的话，这是一个默认的代理类型。配置一个远程代理如下：

```
<Agent name="LocalAgent">
   <!-- Monitors -->
</Agent>
```

（2）TCP 远程代理：

这个代理存活在本地或远程机器的一个单独的进程中，通过 TCP 远程完成连接，是一种被本地运行时支持的 RPC 形式。为了使用远程代理，代理进程必须首先运行起来。

Configuring a remote agent

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

Running remote agent

```
c:\peach3> peach.exe -a tcp

[[ Peach v3.0
[[ Copyright (c) Michael Eddington

[*] Starting agent server
 -- Press ENTER to quit agent --
```

（3）ZeroMQ 代理：

这个代理存活在本地或者远程机器的一个独立进程中。使用 ZeroMQ 完成连接。ZeroMQ 支持许多类型的语言。可以使用这个代理信道来执行一个非.net 代理（比如 Python 或者 ruby）。为了使用一个远程代理，代理进程必须首先运行起来。

Configuring a remote agent

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

Running remote agent

```
c:\peach3> peach.exe -a zmq

[[ Peach v3.0
[[ Copyright (c) Michael Eddington

[*] Starting agent server
 -- Press ENTER to quit agent --
```

（4）REST Json 代理：

这个代理被用来与用其他语言编写的自定义的远程代理进行通信。

Example configuration with remote publisher

```
<Agent name="TheAgent" location="http://127.0.0.1:9980">
        <Monitor class="WindowsDebugger">
                <Param name="CommandLine" value="mspaint.exe fuzzed.png" />
                <Param name="WinDbgPath" value="C:\Program Files (x86)\Debugging Tools for Windows (x86)" />
                <Param name="StartOnCall" value="ScoobySnacks"/>
        </Monitor>
        <Monitor class="PageHeap">
                <Param name="Executable" value="mspaint.exe"/>
                <Param name="WinDbgPath" value="C:\Program Files (x86)\Debugging Tools for Windows (x86)" />
        </Monitor>
</Agent>

<Test name="Default">
        <Agent ref="TheAgent"/>
        <StateModel ref="TheState"/>

        <Publisher class="Remote">
                <Param name="Agent" value="TheAgent"/>
                <Param name="Class" value="File"/>
                <Param name="FileName" value="fuzzed.png"/>
        </Publisher>

</Test>
```

Sample session

```
GET /Agent/AgentConnect
<< { "Status":"true" }

POST /Agent/StartMonitor?name=Monitor_0&cls=WindowsDebugger
>> {"args":{"CommandLine":"mspaint.exe fuzzed.png","WinDbgPath":"C:\\Program Files (x86)\\Debugging Tools for Windows (x86)","StartOnCall":"ScoobySnacks"}}
<< { "Status":"true" }

POST /Agent/StartMonitor?name=Monitor_1&cls=PageHeap
>> {"args":{"Executable":"mspaint.exe","WinDbgPath":"C:\\Program Files (x86)\\Debugging Tools for Windows (x86)"}}
<< { "Status":"true" }

GET /Agent/SessionStarting
<< { "Status":"true" }

GET /Agent/IterationStarting?iterationCount=1&isReproduction=False
<< { "Status":"true" }

GET /Agent/IterationFinished
<< { "Status":"true" }

GET /Agent/DetectedFault
<< { "Status":"true" }
// Status of true indicates a fault was detected. False for no fault.

GET /Agent/GetMonitorData
<< {
        "Results":[
                {
                        "iteration":0,
                        "controlIteration":false,
                        "controlRecordingIteration":false,
                        "type":0,  (0 unknown, 1 Fault, 2 Data)
                        "detectionSource":null,
                        "title":null,
                        "description":null,
                        "majorHash":null,
                        "minorHash":null,
                        "exploitability":null,
                        "folderName":null,
                        "collectedData":[
                                {"Key":"data1","Value":"AA=="}
                        ]
                }
        ]
}
```

```
GET /Agent/IterationStarting?iterationCount=1&isReproduction=True
<< { "Status":"true" }

GET /Agent/IterationFinished
<< { "Status":"true" }

GET /Agent/DetectedFault
<< { "Status":"true" }
// Status of true indicates a fault was detected. False for no fault.

GET /Agent/GetMonitorData
<< {
        "Results":[
                {
                        "iteration":0,
                        "controlIteration":false,
                        "controlRecordingIteration":false,
                        "type":0,  (0 unknown, 1 Fault, 2 Data)
                        "detectionSource":null,
                        "title":null,
                        "description":null,
                        "majorHash":null,
                        "minorHash":null,
                        "exploitability":null,
                        "folderName":null,
                        "collectedData":[
                                {"Key":"data1","Value":"AA=="}
                        ]
                }
        ]
}

GET /Agent/Publisher/stop
<< { "Status":"true" }

GET /Agent/SessionFinished
<< { "Status":"true" }

GET /Agent/StopAllMonitors
<< { "Status":"true" }

GET /Agent/AgentDisconnect
<< { "Status":"true" }
```

Sample session with remote publisher

```
GET /Agent/AgentConnect
<< { "Status":"true" }

POST /Agent/StartMonitor?name=Monitor_0&cls=WindowsDebugger
>> {"args":{"CommandLine":"mspaint.exe fuzzed.png","WinDbgPath":"C:\\Program Files (x86)\\Debugging Tools for Windows (x86)","StartOnCall":"ScoobySnacks"}}
<< { "Status":"true" }

POST /Agent/StartMonitor?name=Monitor_1&cls=PageHeap
>> {"args":{"Executable":"mspaint.exe","WinDbgPath":"C:\\Program Files (x86)\\Debugging Tools for Windows (x86)"}}
<< { "Status":"true" }

GET /Agent/SessionStarting
<< { "Status":"true" }

GET /Agent/IterationStarting?iterationCount=1&isReproduction=False
<< { "Status":"true" }

POST /Agent/Publisher/Set_Iteration
>> {"iteration":1}
<< { "error":"false", "errorString":null }

POST /Agent/Publisher/Set_IsControlIteration
>> {"isControlIteration":true}
<< { "error":"false", "errorString":null }

POST /Agent/Publisher/Set_IsControlIteration
>> {"isControlIteration":true}
<< { "error":"false", "errorString":null }

POST /Agent/Publisher/Set_Iteration
>> {"iteration":1}
<< { "error":"false", "errorString":null }

GET /Agent/Publisher/start
<< { "error":"false", "errorString":null }

GET /Agent/Publisher/open
<< { "error":"false", "errorString":null }

POST /Agent/Publisher/output
>> {"data":"SGVsbG8gV29ybGQ="}
<< { "error":"false", "errorString":null }
```

```
GET /Agent/Publisher/close
<< { "error":"false", "errorString":null }

POST /Agent/Publisher/call
>> {"method":"ScoobySnacks","args":[{"name":"p1","data":"SGVsbG8gV29ybGQ=","type":0}]}
<< { "error":"false", "errorString":null }

GET /Agent/IterationFinished
<< { "Status":"true" }

GET /Agent/DetectedFault
<< { "Status":"true" }
// Status of true indicates a fault was detected. False for no fault.

GET /Agent/GetMonitorData
<< {
        "Results":[
                {
                        "iteration":0,
                        "controlIteration":false,
                        "controlRecordingIteration":false,
                        "type":0,  (0 unknown, 1 Fault, 2 Data)
                        "detectionSource":null,
                        "title":null,
                        "description":null,
                        "majorHash":null,
                        "minorHash":null,
                        "exploitability":null,
                        "folderName":null,
                        "collectedData":[
                                {"Key":"data1","Value":"AA=="}
                        ]
                }
        ]
}

GET /Agent/IterationStarting?iterationCount=1&isReproduction=True
<< { "Status":"true" }
```

```
POST /Agent/Publisher/Set_Iteration
>> {"iteration":1}
<< { "error":"false", "errorString":null }

POST /Agent/Publisher/Set_IsControlIteration
>> {"isControlIteration":true}
<< { "error":"false", "errorString":null }

POST /Agent/Publisher/Set_IsControlIteration
>> {"isControlIteration":true}
<< { "error":"false", "errorString":null }

POST /Agent/Publisher/Set_Iteration
>> {"iteration":1}
<< { "error":"false", "errorString":null }

GET /Agent/Publisher/start
<< { "error":"false", "errorString":null }

GET /Agent/Publisher/open
<< { "error":"false", "errorString":null }

POST /Agent/Publisher/output
>> {"data":"SGVsbG8gV29ybGQ="}
<< { "error":"false", "errorString":null }

GET /Agent/Publisher/close
<< { "error":"false", "errorString":null }

POST /Agent/Publisher/call
>> {"method":"ScoobySnacks","args":[{"name":"p1","data":"SGVsbG8gV29ybGQ=","type":0}]}
<< { "error":"false", "errorString":null }

GET /Agent/IterationFinished
<< { "Status":"true" }

GET /Agent/DetectedFault
<< { "Status":"true" }
// Status of true indicates a fault was detected. False for no fault.
```

```
GET /Agent/GetMonitorData
<< {
        "Results":[
                {
                        "iteration":0,
                        "controlIteration":false,
                        "controlRecordingIteration":false,
                        "type":0,  (0 unknown, 1 Fault, 2 Data)
                        "detectionSource":null,
                        "title":null,
                        "description":null,
                        "majorHash":null,
                        "minorHash":null,
                        "exploitability":null,
                        "folderName":null,
                        "collectedData":[
                                {"Key":"data1","Value":"AA=="}
                        ]
                }
        ]
}

GET /Agent/Publisher/stop
<< { "Status":"true" }

GET /Agent/SessionFinished
<< { "Status":"true" }

GET /Agent/StopAllMonitors
<< { "Status":"true" }

GET /Agent/AgentDisconnect
<< { "Status":"true" }
```

7.6 监视器

主要有以下几种监视器：windows 监视器、OSX 监视器、Linux 监视器和跨平台监视器。

7.6.1 windows 监视器

（1）Windows Debugger Monitor

WindowsDebugger 监视器控制了一个 windows 调试句柄。主要有以下用途：进程调试、服务调试、内核调试。

必须参数：

- Commandline---用逗号分隔的窗口名字。
- Processname---当找到一个窗口的时候，触发错误，默认为假。

- Kernelconnectionstring---内核调试的连接字符串。
- Service---要挂载的 windows 服务名称。如果停止或者崩溃，服务将会被启动。
  可选参数：
- Symbolspath---符号表路径或者服务。默认为：
  “SRV*http://msdl.microsoft.com/download/symbols”
- Windbgpath---windbg 的安装路径。尽量在本地。
- Noncrystalline---直到从状态模型的匹配调用完成时，debugger 才会被挂载。
- Ignorefirstchanceguardpage---忽略第一个机会机会保护页面错误。这些有时是假阳性或反调试错误。默认为假。
- Ignoresecondchanceguardpage---忽略第二个机会保护页面错误。这些有时是假阳性或反调试错误。默认为假。
- Nocpukill---不要使用进程 CPU 使用率提前终止。默认为假。
- Faultonearlyexit---如果进程存在，触发错误。默认为假。
- Waitforexitoncall--如果时间间隔到了，-等待状态模型调用的进程退出和参数故障。
- Waitforexittimeout---等待退出，timeout 值单位为微秒。（-1 位无穷大）默认位 10000。
- Restaroneachtest---为每次迭代重启进程。默认为假。

Examples

Commandline Configuration

```
<Agent name="Local">
        <Monitor class="WindowsDebugger">
                <Param name="CommandLine" value="CrashableServer.exe 127.0.0.1 4244" />
                <!--<Param name="WinDbgPath" value="C:\Program Files (x86)\Debugging Tools for Windows (x86)" />-->
        </Monitor>
</Agent>
```

Kernel Configuration

```
<Param name="KernelConnectionString" value="npipe:server=Server, pipe=PipeName [,password=Password]" />
```

Service Configuration

```
<Param name="Service" value="WinDefend" />
```

Process Configuration

```
<Param name="ProcessName" value="CrashableServer.exe" />
```

StartOnCall Configuration

```
<StateModel name="TheState" initialState="initial">
    <State name="initial">
        <Action type="call" method="launchProgram" publisher="Peach.Agent"/>
    </State>
</StateModel>


<Agent name="Local">
    <Monitor class="WindowsDebugger">
        <Param name="CommandLine" value="CrashableServer.exe 127.0.0.1 4244"/>
        <Param name="StartOnCall" value="launchProgram"/>
    </Monitor>
</Agent>
```

Exit Configurations

```
<Agent name="Local">
    <Monitor class="WindowsDebugger">
        <Param name="CommandLine" value="CrashableServer.exe 127.0.0.1 4244"/>
        <Param name="NoCpuKill" value="true"/>
        <Param name="FaultOnEarlyExit" value="false"/>
        <Param name="WaitForExitTimeout" value="250"/>
    </Monitor>
</Agent>
```

WaitForExitOnCall Configuration

```
<StateModel name="TheState" initialState="initial">
    <State name="initial">
        <Action type="call" method="exitProgram" publisher="Peach.Agent"/>
    </State>
</StateModel>


<Agent name="Local">
    <Monitor class="WindowsDebugger">
        <Param name="CommandLine" value="CrashableServer.exe 127.0.0.1 4244"/>
        <Param name="WaitForExitOnCall" value="exitProgram"/>
    </Monitor>
</Agent>
```

（2）cleanup registry monitor

cleanup registry 监视器将移除一个注册表键或者它的子健。以下前缀键被用到：

- HKCU\---当前用户。
- HKCC\---当前配置。
- HKLM\---本地机器。
- HKPD\---性能数据。
- HKU\---用户。

参数：

Key---要移除的注册表键。

Childrenonly---仅有的清除子健。默认为假。

Examples

Clean up after office

```
<Agent name="Local">
        <Monitor class="CleanupRegistry">
                <Param name="Key" value="HKLM\SOFTWARE\Office13\Recovery" />
        </Monitor>
</Agent>
```

（3）pageheap 监视器（windows 系统）

Pageheap 监视器将使 pageheap 调试选项对可执行程序可用。

参数：

Executable---可执行程序名字。（没有路径）。

Windbgpath---windbg 安装路径。尽量在本地。

Examples

Enable for Notepad

```
<Agent name="Local">
        <Monitor class="PageHeap">
                <Param name="Executable" value="notepad.exe" />
        </Monitor>
</Agent>
```

（3）PopupWatcher 监视器（windows）

Popupwatcher 监视器能根据标题关闭窗口。

参数：

WindowNames---被逗号分隔的窗口名字。

Fault---当找到窗口时，触发错误。默认为假。

Examples

Close Notepad

```
<Agent name="Local">
        <Monitor class="PopupWatcher">
                <Param name="WindowNames" value="Notepad" />
        </Monitor>
</Agent>
```

Fault on Assert

```
<Agent name="Local">
        <Monitor class="PopupWatcher">
                <Param name="WindowNames" value="Assert" />
                <Param name="Fault" value="True" />
        </Monitor>
</Agent>
```

（4）windowsService 监视器。

WindowsService 监视器控制一个 windows 服务。

参数：

Service---系统服务的名字，也可以是服务显示的名字。

MachineName---服务存在的机器，可选，默认为本地机器。

FaultOnEarlyexit---如果服务提前存在，则为故障。

Restart---windows 服务要挂载的名字，如果停止或者崩溃，服务将重启。

StartTimeout---服务要挂载的名字，如果停止或者崩溃，服务将重启。

Examples

Start IIS

```
<Agent name="Local">
        <Monitor class="WindowsService">
                <Param name="Service" value="World Wide Web Publishing Service" />
        </Monitor>
</Agent>
```

7.6.2 OSX 监视器

（1）CrashWrangler 监视器

CrashWangler 监视器将启动一个进程和监视器感兴趣的崩溃。这个监视器采用苹果系统自带的 CrashWrangler 工具，这个工具能从开发者网站下载。为了该工具能够正常运行，它必须在每个机器上进行编译。

参数：

Command---要执行的命令。

Arguments---命令行参数，可选，默认没有。

StartOnCall---状态模型调用的启动命令。可选，默认没有。

UseDebugMalloc---使用 OSX Debug Malloc（比较慢），可选默认为假。

ExecHandler---Crash Wrangler 执行处理程序，可选，默认为 exc_handler。

ExploitableReads---读 a / v 被认为是可利用的？可选，默认为假。

NoCpuKill---通过 CPU 使用禁用进程杀死。可选，默认为假。

CwLogFile---CrashWrangler 记录文件。可选，默认为 cw.log。

CwLockFile---CrashWrangler 锁文件，可选，默认为 cw.lock。

CwPidFile---CrashWrangler PID 文件，可选，默认为 cw.pid。

Examples

Fuzzing Safari

```
<Agent name="Local">
        <Monitor class="CrashWrangler">
                <Param name="Command" value="/Applications/Safari.app/Contents/MacOS/Safari" />
                        <Param name="Arguments" value="fuzzed.bin" />

                        <Param name="UseDebugMalloc" value="false" />
                        <Param name="ExploitableReads" value="true" />
                        <Param name="ExecHandler" value="./exc_handler" />

                        <Param name="StartOnCall" value="ScoobySnacks" />
        </Monitor>
</Agent>
```

（2）CrashReporter Monitor（OSX）

CrashReporter 监视器将报告被 OSX 系统探测到的 Crash 报告。

参数：

ProcessName---需要监视的进程名字。可选，默认为所有。

Examples

Catch all crashes

```
<Agent name="Local">
        <Monitor class="CrashReporter"/>
</Agent>
```

7.6.3 Linux 监视器

LinuxCrash 监视器用一个脚本来捕捉错误进程，该脚本被内置在内核中。

参数：

Executable---目标可执行程序，被用于过滤崩溃，可选的，默认为所有。

LogFolder---记录文件的文件夹。可选默认为“/var/peachcrash”。

Mono---mono 执行=程序所需的运行时的全路径。可选，默认为“/usr/bin/mono”。

Examples

Example

```
<Agent name="Local">
        <Monitor class="LinuxCrashMonitor"/>

        <Monitor class="Process">
                <Param name="Executable" value="./CrashingProgram" />
                <Param name="StartOnCall" value="Start" />
                <Param name="Arguments" value="fuzzed.bin" />
        </Monitor>
</Agent>
```

7.6.4 跨平台监视器

（1）canakit relay 监视器

它控制一组能够打开/关闭 AC 和 DC 线路的继电器。这对于在模糊运行期间打开和关闭设备非常方便。 任何带有交流或直流电源线的设备都可以通过接线来控制。

参数：

Serialport---电路板上的串行端口。

RelayNumber---延迟触发。

ResetEveryIteration--在每次迭代时重启动电源，可选，默认为假。

OnOffPause---在开/关之间暂停，以微秒为单位，可选，默认为 0.5 秒。

ResetOnStart---在启动时，重置设备，可选，默认为假。

ReverseSwitch---当 NC 端口被用于电源时，切换命令的顺序。

Examples

Reset power on relay 1

```
<Agent name="Local">
        <Monitor class="CanaKitRelay">
                <Param name="SerialPort" value="COM2" />
                <Param name="RelayNumber" value="1" />
        </Monitor>
</Agent>
```

（2）Cleanup Folder 监视器

Cleanup Folder 监视器在每次迭代时将会移除文件夹的内容，这用于清理后的目标。

参数：

Folder---要清理的文件夹

Examples

Remove contents of a folder

```
<Agent name="Local">
        <Monitor class="CleanupFolder">
                <Param name="Folder" value="c:\temp" />
        </Monitor>
</Agent>
```

（3）IpPower9258 监视器

IpPower9258 监视器控制找一个联网的能量切换，它允许设备在 fuzzing 时被被关闭或者开启。这个模型是“IP Power 9258”，特别是在购买 eaby 和亚马逊时有用。

参数：

Host---主机或者 IP 地址（能包含 http 接口端口，比如 8080）。

User---用户名。

Password---密码。

Port---重写设置的端口。

ResetEveryIteration---每次迭代时重启电源，可选，默认为假。

powerOnoffpause---在电源打开和关闭时暂停，以微秒为单位，可选，默认为 0.5 秒。

Examples

Reset power on port 1

```
<Agent name="Local">
        <Monitor class="IpPower9258">
                <Param name="Host" value="192.168.1.1:8080" />
                <Param name="User" value="peach" />
                <Param name="Password" value="power" />
                <Param name="Port" value="1" />
                <Param name="ResetEveryIteration" value="true" />
        </Monitor>
</Agent>
```

（4）内存监视器

内存监视器检查一个进程的内存使用情况，当它达到设置极限时抛出一个错误，在探测无关的内存使用情况式非常有用。

参数：

Pid---监控的进程 ID。

Processname--监控的进程名字。

memoryLimit---发生错误时，内存的使用极限，可选，默认为 0MB。

Stoponfault---如果触发了一个错误，停止 fuzzing，可选的，默认为假。

Examples

Monitor memory via PID

```
<Agent name="Local">
        <Monitor class="Memory">
                <Param name="Pid" value="2387" />
                <Param name="MemoryLimit" value="1000" />
        </Monitor>
</Agent>
```

（5）Pcap 监视器

Pcap 监视器在迭代时有一个网络捕获器，被捕获的数据是废弃的，在每次迭代时重写捕获。如果发生错误，被捕获的数据被记录为一个 pcap 文件，pcap 文件 wireshark 和 tcpdump 兼容。

参数：

Device---捕获的设备名字。

Filter---pcap 类型过滤器，可选。

Examples

Capture port 80

```
<Agent name="Local">
        <Monitor class="Pcap">
                <Param name="Device" value="Local Area Connection" />
                <Param name="Filter" value="port 80" />
        </Monitor>
</Agent>
```

（6）Ping 监视器

Ping 监视器将阻塞,直到超时是打击。此监视器对于验证目标是否仍然正常并等待目标重新启动非常有用。

参数：

Host---主机或者 IP 地址。

Timeout---时间间隔，微秒为单位，可选，默认为 1000。

Data---ping 包中药发生的数据，可选。

Faultonsuccess---如果 ping 成功了，出现错误，可选，默认为假。

Examples

Ping Host

```
<Agent name="Local">
        <Monitor class="Ping">
                <Param name="Host" value="www.google.com" />
        </Monitor>
</Agent>
```

（7）进程监视器

进程监视器在 fuzzing 时控制了一个进程，它由许多特性：

- 当进程死掉时，重启一个进程。
- 如果一个进程已经退出了，记录一个错误。
- 每次迭代时，重启一个进程。
- 终止一个 cpu 使用率很低的进程。
- 和 StateModel 相互影响，允许等待进程退出。
- 和 StateModel 相互影响，允许延迟开始一个进程。

参数：

Executable---要激活的可执行程序。

Arguments---命令行参数，可选。

Restartoneachtest---每次迭代时重启进程，可选，默认为假。

Faultonearlyexit---如果进程退出，触发错误，可选，默认为假。

Nocpukill---当 cpu 使用率接近 0 时，禁用进程杀死，可选，默认为假。

Startoncall---当从 StateModel 中调用一个动作时，开始进程。这个值和方法值必须匹配，可选。

Waitforexitoncall---在状态模型调用时，等待进程退出。这个值和方法值必须匹配，可选。

Waitforexittimeout---等待超时值。当超时时，触发错误，可选，默认禁用。

Examples

Start process

```
<Agent name="Local">
        <Monitor class="Process">
                <Param name="Executable" value="notepad.exe" />
                <Param name="Arguments" value="README.txt" />
        </Monitor>
</Agent>
```

Restart process on each test

```
<Agent name="Local">
        <Monitor class="Process">
                <Param name="Executable" value="notepad.exe" />
                <Param name="Arguments" value="README.txt" />
                <Param name="RestartOnEachTest" value="true" />
        </Monitor>
</Agent>
```

Start process from state model

```
<StateModel name="TheStateModel">
        <State name="initial">
                <Action type="call" method="ScoobySnacks" />
        </State>
</StateModel>

<Agent name="Local">
        <Monitor class="Process">
                <Param name="Executable" value="notepad.exe" />
                <Param name="Arguments" value="README.txt" />
                <Param name="StartOnCall" value="ScoobySnacks" />
        </Monitor>
</Agent>
```

Wait for process to exit in state model

```
<StateModel name="TheStateModel">
        <State name="initial">
                <!-- This action will block until process exits -->
                <Action type="call" method="ScoobySnacks" />
        </State>
</StateModel>

<Agent name="Local">
        <Monitor class="Process">
                <Param name="Executable" value="notepad.exe" />
                <Param name="Arguments" value="README.txt" />
                <Param name="WaitForExitOnCall" value="ScoobySnacks" />
        </Monitor>
</Agent>
```

（8）Processkiller 监视器

在每次迭代后，Processkiller 监视器将会杀死（终结）指定进程。

参数：

Processnames---要杀死的进程名字，逗号分隔。

Examples

Terminate two processes

```
<Agent name="Local">
        <Monitor class="ProcessKiller">
                <Param name="ProcessNames" value="nc.exe,foo.exe" />
        </Monitor>
</Agent>
```

（9）保存文件监视器

当错误发生时，保存文件监视器将会保存一个指定的文件作为记录数据的一部分。

参数：

Filename---保存到记录数据的 文件。

Examples

Save a file when fault occurs

```
<Agent name="Local">
        <Monitor class="SaveFile">
                <Param name="Filename" value="c:\temp\output.log" />
        </Monitor>
</Agent>
```

（10）socket 监视器

Socket 监视器等待一个 TCP 或者 UDP 的连接。如果连接被接受了，错误被触发。

参数：

Host---远程主机的 IP 地址，可选默认为空。

Interface---监听的 IP 地址接口，可选，默认为 0.0.0.0。

Port---监听端口，可选，默认为 8080。

Protocol---要监听的协议类型，可选，默认为 tcp。

Timeout---等待连接的时长，可选，默认为 1000ms。

Faultonsuccess---如果没有连接被记录，发生错误，可选，默认为假。

Examples

Listen for incoming connection

```
<Agent name="Local">
        <Monitor class="Socket">
                <Param name="Port" value="53" />
        </Monitor>
</Agent>
```

（11）ssh 监视器

Ssh 监视器通过 ssh 连接一个远程主机。监视器支持密码、键盘和私有密钥身份验证方法。监视器在一个远程系统上运行一个给定的命令。如果发生错误，一个正则表达式能被提供给命令结果来确定。监视器在模糊运行过程中保持对系统开放的持久连接。

参数：

Host---ssh 连接的主机。

Usename---ssh 用户名。

Command---检查错误的命令。

Password---ssh 账户的密码，可选，默认为空。

Keypath---ssh key 的路径，可选默认为空。

checkValue---正则表达式匹配命令响应，可选，默认为空。

Faultonmatch---如果正则表达式匹配，触发错误，可选，默认为真。

Example

```
<Agent name="LocalAgent">
        <Monitor class="Ssh">
                <Param name="Host" value="my.target.com" />
                <Param name="Username" value="tester" />
                <Param name="Password" value="Password!" />
                <Param name="Command" value="ls /var/cores/*.core" />
                <Param name="CheckValue" value="target.*?.core" />
                <Param name="FaultOnMatch" value="true" />
        </Monitor>
</Agent>
```

（12）ssh 下载器监视器

Ssh 下载器监视器可以通过 ssh sftp 从远程主机上下载一个文件或者文件夹。这个监控器支持密码、键盘、私有秘钥认证方法，有能力删除它下载的文件。

参数：

Host---ssh 主机。

Username---ssh 用户名

Password---ssh 账户密码，可选，默认为空。

Keypath---ssh key 路径，可选，默认为空。

File---要下载的文件，可选，默认为空。

Folder---下载的文件夹，可选，默认为空。

Remove---下载后，移除远程文件，可选，默认为真。

Example

```
<Agent name="LocalAgent">
        <Monitor class="SshDownloader">
                <Param name="Host" value="my.target.com" />
                <Param name="Username" value="tester" />
                <Param name="Password" value="Password!" />
                <Param name="File" value="/var/cores/core" />
                <Param name="Remove" value="false" />
        </Monitor>
</Agent>
```

（13）vmware 监视器

Vmware 监视器能控制一个 wmware 虚拟机。监视器能启动一个虚拟机，在每次迭代时重新设置一个快照。当一个错误发生时，它会重新设置一个指定的快照。

参数：

Vmx---虚拟机路径

Host---主机名字，可选。

Login---远程主机上认证用户名。可选。

Pasword---远程主机上的认证密码，可选。

Hosttype---远程主机类型，可选，默认为“default”。

Hostport---远程主机上的 TCP/IP 端口，可选。

Default---default。

VIserver---vCenter Server，ESX/ESXi 主机，VMWare Server 2.0。

Workstation---VMWare Workstation

WorkstationShared---VMWare Workstation（shared 模式）。

Player---VMWare player。

Server---VMWare Server 1.0.x。

snapshotIndex---VM snapshot 索引，可选。

Snapshotname---VM snapshot 名字，可选。

ResetEveryIteration---每次迭代时，重新设置 VM 快照，可选，默认为假。

ResetonfaultBeforecollection---在数据收集期间，当探测到一个错误之后，重新设置 VM。

Waitfortoolsinguest---等待工具在客户机中启动，可选，默认为真。

Waittimeout---等待客户机工具多少秒，可选，默认为 600。

Examples

Start Virtual Machine

```
<Agent name="Local">
        <Monitor class="Vmware">
                <Param name="Vmx" value="D:\VirtualMachines\OfficeWebTest\OfficeWebTest.vmx" />
                <Param name="HostType" value="Workstation" />
                <Param name="SnapshotName" value="Fuzzing" />
        </Monitor>
</Agent>
```

Start Virtual Machine hosted on ESXi

```
<Agent name="Local">
        <Monitor class="Vmware">
                <Param name="Vmx" value="[ha-datacenter/datastore1] guest/guest.vmx" />
                <Param name="SnapshotName" value="Fuzzing" />
        </Monitor>
</Agent>
```

7.7 测试

Test 元素被用于配置一个指定的 fuzzing 测试，这个配置通过一个 Publisher 和其他选项（比如 including/excluding 等被变异的元素、Agents、fuzzing 策略）组合成一个 StateModel。多个 test 元素是支持的，在 Peach 命令行使用中，只需简单提供 test 元素的名字即可。

```
<Test name="Default">

  <!-- Optionally exclude some elements from mutation -->
  <Exclude xpath="//Reserved" />
  <Exclude xpath="//Magic" />

  <!-- Optional agent references -->
  <Agent ref="LocalWindowsAgent" platform="windows" />
  <Agent ref="LocalOsxAgent" platform="osx" />
  <Agent ref="LocalLinuxAgent" platform="linux" />

  <Agent ref="RemoteAgent" />

  <!-- Indicate which state model to use (required) -->
  <StateModel ref="TheState" />

  <!-- Configure the publisher to use (required) -->
  <Publisher class="Tcp">
     <Param name="Host" value="127.0.0.1" />
     <Param name="Port" value="9001" />
  </Publisher>

  <!-- Use a different fuzzing strategy -->
  <Strategy class="Random" />

  <!-- Log output to disk -->
  <Logger class="File">
    <Param name="Path" value="logs" />
  </Logger>
</Test>
```

属性：

Name---必备的，test 元素的名字，默认为“Default”。

Waittime---每次测试之间的等待时间，默认为 0。

Faultwaittime---在开始下一次迭代时等待错误的时间，默认为 0。

controlIneration---我们只需控制迭代的频率，默认为 0。

有效子元素：

Agent（可选）

StateModel（必须）

Publisher（必须）

Include（可选）

Exclude（可选）

Strategy（可选）

Logger（可选，推荐）

7.8 Publishers

Publishers 是 Peach 发送和接收数据的 I/O 接口，Publisher 支持基于流和操作的调用。当 fuzzer 运行时，除了 slurp 之外，所有的 action 使用一个 Publisher 来执行操作。不同的 Publishers 支持不同的 action 类型。比如，文件 Publisher 支持从一个文件读取数据作为输入，把数据写入一个文件作为输出，但是不支持接收或者调用。这和 com Publisher 不同，它支持调用，但不支持输入、输出或接收。所有的 fuzzing 定义必须使用至少一个 Publisher，如果需要的话也可以选择多个 Publishers。当使用多个 Publisher 的时候，每个 action 必须指定它引用了哪个 Publisher，通请参考 action 的 publisher 属性中的 name 属性。如果 Publisher 属性丢失了，action 将执行 Test 中定义的第一个 Publisher。

7.8.1 网络 Publishers

当进行网络协议 fuzzing 的时候，Publisher 常常被用于包含目标协议的协议。比如，当进行 http 协议 fuzzing 的时候，使用 Tcp Publisher。当进行 TCPfuzzing 的时候，使用 IPv4 或 IPv6 Publisher。当进行 IPv4 fuzzing 的时候，使用 ethernet Publisher。

7.8.2 自定义 Publishers

Peach 支持创建自定义 Publisher。建议先对一些现有 Publisher 代码进行审核，以了解 Publisher 怎么运行。创建一个自定义的 Publisher 不需要改变 Peach 的源代码。相反，代码放被一个新的.NET 程序集（dll）中，然后把 dll 添加进 Peach bin 文件夹中，Peach 会自动地识别新的 Publisher，并使它可用。

7.8.3 Publishers

（1）Com

Com Publisher 允许调用方法和 com 对象上的属性。

参数：clsid---com clsid。

操作：Call---调用一个方法。Getproperty---获取属性值。Setproperty---设置属性值。

Examples

Calling a method

```
<StateModel name="TheState">
        <State name="initial">
                <Action type="call" method="DoCoolThings">
                        <Param name="Name">
                                <DataModel ref="NameDataModel"/>
                        </Param>
                </Action>
        </State>
</StateModel>

<Test name="Default">
    <!-- ... -->
        <Publisher class="COM">
                <Param name="clsid" value="{d20ea4e1-3957-11d2-a40b-0c5020524153}" />
        </Publisher>
</Test>
```

（2）Console

Console Publisher 将会输出数据到标准输出。

参数：没有。**操作：**Output---需要显示的数据。

Examples

Display data to console

```
<DataModel name="Data">
   <!-- ... -->
</DataModel>

<StateModel name="TheState">
  <State name="initial">
    <Action type="output">
      <DataModel ref="Data" />
    </Action>
  </State>
</StateModel>

<Test name="Default">
  <!-- ... -->
  <Publisher class="Console" />
</Test>
```

（3）consoleHex

consoleHex Publisher 将会输出数据到一个标准输出。数据将会以十六进制格式显示。

参数：Bytesperline---每行文本字节数，可选，默认为 16。**操作：**Output---要示的数据。

Examples

Display data to console

```
<DataModel name="Data">
   <!-- ... -->
</DataModel>

<StateModel name="TheState">
  <State name="initial">
    <Action type="output">
      <DataModel ref="Data" />
    </Action>
  </State>
</StateModel>

<Test name="Default">
  <!-- ... -->
  <Publisher class="ConsoleHex" />
</Test>
```

Display data with custom bytes per line

```
<DataModel name="Data">
   <Blob/>
</DataModel>

<StateModel name="TheState">
  <State name="initial">
    <Action type="input">
      <DataModel ref="Data" />
    </Action>
  </State>
</StateModel>

<Test name="Default">
  <!-- ... -->
  <Publisher class="ConsoleHex">
    <Param name="BytesPerLine" value="8" />
  </Publisher>
</Test>
```

（4）File

File Publisher 将会打开一个文件来读和写。

参数：

Filename---要打开文件的名字。

Overwrite---覆盖存在的文件，可选，默认为真。

Append---向存在的文件附加数据，可选默认为假。

操作：

Output---数据被写进一个文件。

Input---从文件读取数据。

Examples

Write to file

```
<DataModel name="Data">
   <!-- ... -->
</DataModel>

<StateModel name="TheState">
  <State name="initial">
    <Action type="output">
      <DataModel ref="Data" />
    </Action>
  </State>
</StateModel>

<Test name="Default">
  <!-- ... -->
  <Publisher class="File">
    <Param name="FileName" value="fuzzed.bin" />
  </Publisher>
</Test>
```

Read from file

```
<DataModel name="Data">
   <Blob/>
</DataModel>

<StateModel name="TheState">
  <State name="initial">
    <Action type="input">
      <DataModel ref="Data" />
    </Action>
  </State>
</StateModel>

<Test name="Default">
  <!-- ... -->
  <Publisher class="File">
    <Param name="FileName" value="fuzzed.bin" />
  </Publisher>
</Test>
```

（5）FilePerIteration

FilePerIteration Publisher 在每次迭代 fuzzer 执行时，将会创建一个输出文件。当进行预先产生 fuzzing 事件时，非常有用。

参数：

Filename---创建文件的名字，文件名字必须包含“{0}”，它将被迭代次数代替。

操作：

Output---写数据到文件。

Examples

```
<DataModel name="Data">
   <!-- ... -->
</DataModel>

<StateModel name="TheState">
  <State name="initial">
    <Action type="output">
      <DataModel ref="Data" />
    </Action>
  </State>
</StateModel>

<Test name="Default">
  <!-- ... -->
  <Publisher class="FilePerIteration">
    <Param name="FileName" value="fuzzed_{0}.bin" />
  </Publisher>
</Test>
```

（6）Http

Http Publisher 通过你选择的方法发送 HTTP 数据。这个 Publisher 支持以下特性：

- 通过 Basic、Digest 或者 windows 集成来认证。
- 定义方法类型。
- Fuzzing 和动态头设置（键和值）。
- Fuzzing 和动态查询字符串设置。
- 可选的 cookie 支持。
- SSL

参数：

Method---HTTP 方法类型（Get，Post 等）。

Url---目标 url。

Baseurl---一些认证类型使用的基本 url，可选。

Username---认证用户名，可选。

Domain---认证域名，可选。

Cookies---可用的 cookie 支持，可选，默认为真。

Cookiesacrossiterations---跨迭代跟踪 cookies，可选，默认为假。

Timeout---等待多久数据连接，单位为微秒，可选，默认为 3000。

Ignorecerterrors---忽略 cert 状态，允许 http，默认为假。

操作：

Call---为了 fuzzing 查询字符串或者头，支持指定的方法名。

Query---为一次调用操作指定方法名字，第一个参数为查询字符串。

Header---为一次调用操作指定方法名字，第一个参数是头名字，第二个是值。

Output---通过 output 发送数据，通过输出发送的数据作为 HTTP 主体提供。

Examples

Post data to a URL

```
<DataModel name="PostBody">
   <!-- ... -->
</DataModel>

<StateModel name="TheState">
  <State name="initial">
    <Action type="output">
      <DataModel ref="PostBody" />
    </Action>
  </State>
</StateModel>

<Test name="Default">
  <!-- ... -->
  <Publisher class="Http">
    <Param name="Method" value="POST" />
    <Param name="Url" value="http://foo.com/user/create" />
  </Publisher>
</Test>
```

Fuzz querystring

```
<DataModel name="QueryModel">
   <String value="key"/>
   <String value="=" token="true" />
   <String value="value"/>
</DataModel>

<StateModel name="TheState">
  <State name="initial">
    <Action type="call" method="Query">
      <Param>
        <DataModel ref="QueryModel" />
      </Param>
    </Action>
  </State>
</StateModel>

<Test name="Default">
  <!-- ... -->
  <Publisher class="Http">
    <Param name="Method" value="GET" />
    <Param name="Url" value="http://foo.com/user/create" />
  </Publisher>
</Test>
```

Fuzz header

```
<DataModel name="HeaderKey">
   <String value="Content-Type" />
</DataModel>
<DataModel name="HeaderValue">
   <String value="html" />
</DataModel>

<StateModel name="TheState">
  <State name="initial">
    <Action type="call" method="Header">
      <Param>
        <DataModel ref="HeaderKey" />
      </Param>
      <Param>
        <DataModel ref="HeaderValue" />
      </Param>
    </Action>
  </State>
</StateModel>

<Test name="Default">
  <!-- ... -->
  <Publisher class="Http">
    <Param name="Method" value="GET" />
    <Param name="Url" value="http://foo.com/user/create" />
  </Publisher>
</Test>
```

（7）NULL

所有发送到 null Publisher 的数据将被丢弃，这个 Publisher 被单元测试使用。

参数：没有。

Examples

```
<Test name="Default">

  <Publisher class="Null" />

</Test>
```

（8）RawEther

RawEther Publisher 允许发送带有 IP 头的 raw IPv6 包。

参数：

Interface---要绑定的接口 IP，可选。

Protocol---要使用的 Ethernet 协议，可选，默认为 ETH_P_ALL。

Timeout---数据/连接等待的时间，单位为微秒，可选，默认为 3000。

操作：

Output---发送数据给远程主机。

Input---从远程主机接收数据。

Examples

Sending data

```
<StateModel name="TheState">
        <State name="initial">
                <Action type="output">
                        <DataModel ref="Frame"/>
                </Action>

                <Action type="output">
                        <DataModel ref="IPHeader"/>
                </Action>

                <Action type="output">
                        <DataModel ref="ProtocolPacket"/>
                </Action>
        </State>
</StateModel>

<Test name="Default">
    <!-- ... -->
        <Publisher class="RawEther">
                <Param name="Interface" value="192.168.1.1" />
                <Param name="Protocol" value="42" />
        </Publisher>
</Test>
```

（9）RawIPv4

RawIPv4 Publisher 允许发送带有 IP 头的 Raw IPv4 包。

参数：

Host---主机名或者远程主机 IP 地址。

Interface---要绑定的接口 IP 地址。

Protocol---要使用的 IP 协议。

Timeout---数据/连接要等待的时间，单位为微秒，可选，默认为 3000。

maxMTu---最大允许可用的 MTU 属性值，可选，默认为 131070。

minMTU---最小允许可用的属性值，可选，默认为 1280。

操作：

Output---发生数据给远程主机。

Input--从远程主机接收数据。

Examples

Sending data

```
<StateModel name="TheState">
        <State name="initial">
                <Action type="output">
                        <DataModel ref="IPHeader"/>
                </Action>

                <Action type="output">
                        <DataModel ref="ProtocolPacket"/>
                </Action>
        </State>
</StateModel>

<Test name="Default">
    <!-- ... -->
        <Publisher class="RawIPv4">
                <Param name="Host" value="192.168.1.1" />
                <Param name="Protocol" value="42" />
        </Publisher>
</Test>
```

（10）RawIPv6

官网内容不可访问。

（11）Rawv4

Rawv4 Publisher 允许发送带有 IP 头的 Raw IPv4 包。

参数：

Host---主机名或者远程主机 IP 地址。

Interface---要绑定的接口 IP 地址。

Protocol---要使用的 IP 协议。

Timeout---数据/连接要等待的时间，单位为微秒，可选，默认为 3000。

maxMTu---最大允许可用的 MTU 属性值，可选，默认为 131070。

minMTU---最小允许可用的属性值，可选，默认为 1280。

操作：

Output---发生数据给远程主机。

Input--从远程主机接收数据。

Examples

Sending data

```
<StateModel name="TheState">
        <State name="initial">
                <Action type="output">
                        <DataModel ref="ProtocolPacket"/>
                </Action>
        </State>
</StateModel>

<Test name="Default">
    <!-- ... -->
        <Publisher class="RawV4">
                <Param name="Host" value="192.168.1.1" />
                <Param name="Protocol" value="42" />
        </Publisher>
</Test>
```

（12）RawV6

RawV6 Publisher 允许发送带有 IP 头的 IPv6 包。

参数：

Host---主机名或者远程主机 IP 地址。

Interface---要绑定的接口 IP 地址。

Protocol---要使用的 IP 协议。

Timeout---数据/连接要等待的时间，单位为微秒，可选，默认为 3000。

maxMTu---最大允许可用的 MTU 属性值，可选，默认为 131070。

minMTU---最小允许可用的属性值，可选，默认为 1280。

操作：

Output---发生数据给远程主机。

Input--从远程主机接收数据。

Examples

Sending data

```
<StateModel name="TheState">
        <State name="initial">
                <Action type="output">
                        <DataModel ref="ProtocolPacket"/>
                </Action>
        </State>
</StateModel>

<Test name="Default">
    <!-- ... -->
        <Publisher class="RawV6">
                <Param name="Host" value="192.168.1.1" />
                <Param name="Protocol" value="42" />
        </Publisher>
</Test>
```

（13）Remote

Remote Publisher 从一个 Peach Agent 进程中运行另外一个 Publisher。

参数：

Agent---运行 Publisher 的代理名字。

Class---运行的 Publisher 名字。

Other parameters---远程 Publisher 所需的参数。

操作：

被远程 Publisher 支持的任何操作。

Examples

Remoting TCP Publisher

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

<Test name="Default">
    <!-- ... -->
        <Agent ref="RemoteAgent" />

        <Publisher class="Remote">
                <Param name="Agent" value="RemoteAgent" />
                <Param name="Class" value="TcpClient"/>

                <!-- Parameters for TcpClient -->
                <Param name="Host" value="127.0.0.1" />
                <Param name="Port" value="8080" />
        </Publisher>
</Test>
```

Starting Peach Agent Process

```
peach.exe -a tcp
```

（14）TCPClient Publisher

TCPClient Publisher 连接一个远程的 TCP 服务。

参数：

Host---主机名或者远程主机 IP 地址。

Port---目的端口数字。

Timeout--数据的等待时间，单位为微秒，可选，默认为 3000。

Connecttimeout---一个新连接的等待时间，单位为微秒，可选，默认为 10000。

操作：

output---发送数据到远程主机。

Input---从远程主机接收数据。

Examples

Sending and receiving data

```
<DataModel name="TheDataModel">
   <String name="value" length="4" />
</DataModel>

<StateModel name="TheState">
        <State name="initial">
                <Action type="output">
                        <DataModel ref="TheDataModel"/>
                        <Data>
                                <Field name="value" value="mike" />
                        </Data>
                </Action>

                <!-- receive 4 bytes -->
                <Action type="input">
                        <DataModel ref="TheDataModel"/>
                </Action>
        </State>
</StateModel>

<Test name="Default">
    <!-- ... -->
        <Publisher class="TcpClient">
                <Param name="Host" value="127.0.0.1" />
                <Param name="Port" value="8080" />
        </Publisher>
</Test>
```

（15）udp Publisher

Udp Publisher 能发送和接收 UDP 包。

参数：

Host---主机或者 ip 地址或者远程主机。

Port---目的端口数字，当第一个包被目标发送时可选。

Srcport---源端口，可选。

Interface---绑定的接口 IP

Timeout---数据/连接的等待时间，单位为微秒，可选，默认为 3000。

maxMTU---最大允许可用 MTU 属性值，可选，默认为 131070

minMTU---最小允许可用 MTU 属性值，可选，默认为 1280。

操作：

Output---发送数据到远程服务器。

Input---从远程主机接收数据。

Examples

Sending and receiving data

```
<DataModel name="TheDataModel">
   <String name="value" length="4" />
</DataModel>

<StateModel name="TheState">
        <State name="initial">
                <Action type="output">
                        <DataModel ref="TheDataModel"/>
                        <Data>
                                <Field name="value" value="mike" />
                        </Data>
                </Action>

                <!-- receive 4 bytes -->
                <Action type="input">
                        <DataModel ref="TheDataModel"/>
                </Action>
        </State>
</StateModel>

<Test name="Default">
    <!-- ... -->
        <Publisher class="Udp">
                <Param name="Host" value="127.0.0.1" />
                <Param name="Port" value="53" />
        </Publisher>
</Test>
```

（16）webservice Publisher

Webservice Publisher 能调用基于 web 服务的 SOAP 和 WCF。默认情况下，Publisher 将会尝试定位一个服务定义或者，被提供的服务定义。

参数：

Url---webService URL

Service---服务名字。

Wsdl---web 服务的 WSDL 路径或 url，可选。

Erroronstatuscode---当状态代码不是 200 时，发生错误，可选，默认为真。

Timeout---数据/连接的等待时间，单位为微秒，可选，默认为 3000。

Throttle---连接之间的等待时间，单位为微秒，可选，默认为 0。

操作：

Call---web 服务要调用的方法属性。

Examples

Example calling web service

```
<DataModel name="TheDataModel">
   <String name="value" />
</DataModel>

<StateModel name="TheState">
        <State name="initial">
                <Action type="call" method="Login">
                        <Param name="user">
                                <DataModel ref="TheDataModel"/>
                                <Data>
                                        <Field name="value" value="mike" />
                                </Data>
                        </Param>
                        <Param name="pass">
                                <DataModel ref="TheDataModel"/>
                                <Data>
                                        <Field name="value" value="Password!" />
                                </Data>
                        </Param>
                </Action>
        </State>
</StateModel>

<Test name="Default">
    <!-- ... -->
        <Publisher class="WebService">
                <Param name="Url" value="http://localhost:5903/TestService.svc" />
                <Param name="Service" value="TestService" />
        </Publisher>
</Test>
```

7.9 Loggers

Peach 有一个可扩展的记录系统，它允许使用者存储他们想要的记录。默认情况下，Peach 使用一个单独的文件系统记录器。

文件系统记录器会指定一个路径，它会创建一个包含运行名字和时间标记的文件夹，文件夹中是真实的记录。除非探测到错误，否则非常少的记录信息会被保存在磁盘空间中。

```
<Logger class="File">
    <Param name="Path" value="logfolder" />
</Logger>
```

参数：

Path---创建记录文件的相对或绝对路径。

7.10 变异策略

一般来说，Peach 的 Datamodel 是在自上而下的。它保证每个数据元素都都能 fuzzing 到各种事例，但对于能够产生大量事例的大的复制系统来说不是最优的。此外，需要一种能能容易改变如何执行 fuzzing 的机制，进而以最优的方法和策略来进行研究。这就需要变异策略。通过实现单个类，用户可以完全控制 Peach 如何 fuzzing 包括状态转换的目标。

7.10.1 随机

随机策略会一直运行，此策略将一次选择最多 MaxFieldsToMutate 个元素进行变异。对于每个选择的元素，其相应的突变体之一是随机选择的。 桃源从随机生成的种子数中导出这些选择的随机性。可以通过将相同的种子值作为 Peach 命令行选项与--seed 命令行选项传递来重复相同的测试运行。这对于重放模糊迭代以重现先前的故障很有用。

参数：

MaxFieldsToMutate---一次变异的最大区域，默认为 6。

Switchcout---进行切换数据集之前执行的迭代次数，默认为 200。

Examples

```
<Test name="Default">
        <StateModel ref="TheStateModel"/>

        <Publisher name="writer" class="File">
                <Param name="FileName" value="fuzzed.tmp"/>
        </Publisher>

        <Strategy class="Random">
                <Param name="MaxFieldsToMutate" value="15" />
                <Param name="SwitchCount" value="100" />
        </Strategy>
</Test>
```

7.10.2 有顺序的

在同一时刻，Peach 将会按照顺序对 DataModel 中的每个元素进行 fuzzing。Peach 将从 Datamodel 的顶部开始，并对每个数据元素执行所有有效的突变，直到所有可能的突变都已用尽。这种策略有一定数量的 fuzzing 迭代。此策略的种子不可配置，始终为 31337。

Examples

```
<Test name="Default">
  <!-- ... -->
  <Strategy class="Sequential" />
</Test>
```

7.10.3 随机确定性（默认）

这个 fuzzing 策略是确定性的（有开始和结尾）。除了突变顺序外，它和顺序策略类似。

7. 11 变异器

更多内容很快会补充。目前的变异器列表如下：

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

7.12 运行

Peach 3 是一个命令行运行时和很多工具和 GUI 组成的 fuzzer。

7.12.1 命令行

Command Line Syntax Help

```
peach.exe
```

Command Line Syntax Help, Non-Windows

```
./peach
```

Starting Agent Process (TcpRemoting)

```
peach.exe -a tcp
```

7.12.2 图形化程序

Peach fuzz bang

Peach fuzz bang 是一个 GUI 文件 fuzzing 工具。这个工具是跨平台的，允许快速 fuzzing 数据消耗。

Peach Validator

Peach Validator 是一个 GUI 程序，用于视觉验证数据加载到数据模型中。

7.13 最小集

该工具将通过目标程序运行每个样本文件并确定代码覆盖率。然后它将找到覆盖最多代码所需的最少文件数。这应该是在 fuzzing 时使用的最小文件集。

Syntax Help

```
> PeachMinset.exe

] Peach 3 -- Minset
] Copyright (c) Deja vu Security

Peach Minset is used to locate the minimum set of sample data with
the best code coverage metrics to use while fuzzing.  This process
can be distributed out across multiple machines to decrease the run
time.

There are two steps to the process:

  1. Collect traces        [long process]
  2. Compute minimum set   [short process]

The first step, collecting traces, can be distributed and the results
collected for analysis by step #2.

Collect Traces
 --------------

Perform code coverage using all files in the 'samples' folder.  Collect
the .trace files for later analysis.

Syntax:
  PeachMinset [-k] -s samples -t traces command.exe args %s

Note:
  %s will be replaced by sample filename.

Compute Minimum Set
 -------------------
```

```
Analyzes all .trace files to determin the minimum set of samples to use
during fuzzing.

Syntax:
  PeachMinset -s samples -t traces -m minset


All-In-One
 ----------

Both tracing and computing can be performed in a single step.

Syntax:
  PeachMinset [-k] -s samples -m minset command.exe args %s

Note:
  %s will be replaced by sample filename.


Distributing Minset
 -------------------

Minset can be distributed by splitting up the sample files and
distributing the collecting of traces to multiple machines.  The
final compute minimum set cannot be distributed.
```

Example Run

```
>peachminset -s pinsamples -m minset -t traces bin\pngcheck.exe %%s

] Peach 3 -- Minset
] Copyright (c) Deja vu Security

[*] Running both trace and coverage analysis
[*] Running trace analysis on 15 samples...
[1:15]   Converage trace of pinsamples\basn0g01.png...done.
[2:15]   Converage trace of pinsamples\basn0g02.png...done.
[3:15]   Converage trace of pinsamples\basn0g04.png...done.
[4:15]   Converage trace of pinsamples\basn0g08.png...done.
[5:15]   Converage trace of pinsamples\basn0g16.png...done.
[6:15]   Converage trace of pinsamples\basn2c08.png...done.
[7:15]   Converage trace of pinsamples\basn2c16.png...done.
[8:15]   Converage trace of pinsamples\basn3p01.png...done.
[9:15]   Converage trace of pinsamples\basn3p02.png...done.
[10:15]   Converage trace of pinsamples\basn3p04.png...done.
[11:15]   Converage trace of pinsamples\basn3p08.png...done.
[12:15]   Converage trace of pinsamples\basn4a08.png...done.
[13:15]   Converage trace of pinsamples\basn4a16.png...done.
[14:15]   Converage trace of pinsamples\basn6a08.png...done.
[15:15]   Converage trace of pinsamples\basn6a16.png...done.

[*] Finished
[*] Running coverage analysis...
[-]   3 files were selected from a total of 15.
[*] Copying over selected files...
[-]   pinsamples\basn3p08.png -> minset\basn3p08.png
[-]   pinsamples\basn3p04.png -> minset\basn3p04.png
[-]   pinsamples\basn2c16.png -> minset\basn2c16.png

[*] Finished
```